Priority Scheduling in Database Management Systems

by

Rajiv Jauhari

Computer Sciences Technical Report #959
August 1990

PRIORITY SCHEDULING IN DATABASE MANAGEMENT SYSTEMS

by

RAJIV JAUHARI

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN --- MADISON
1990

This research was supported in part by the National Scientific Foundation under grant IRI-8657323, by the Digital Equipment Corporation through its Initiatives for Excellence program, and by a subcontract from Xerox Advanced Information Technology. The Xerox subcontract was supported by the Defense Advanced Research Projects Agency and by the Rome Air Development Center under Contract No. F30602-87-C-0029. The views and conclusions contained in this thesis are those of the author and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Rome Air Development Center, or the U.S. Government.

ABSTRACT

The use of priority for scheduling the resources of a database management system (DBMS) is likely to become increasingly important as the set of applications for DBMS technology grows larger. Despite the attention that the concept of priority has received for CPU scheduling in operating systems, it has received relatively little attention in the database systems area. Priority scheduling in a DBMS differs from the problem of priority CPU scheduling due to the heterogeneity and multiplicity of resources that must be scheduled effectively in a DBMS; in addition to the CPUs, these include the disks, the main memory buffers, and the data itself.

This dissertation addresses the problem of scheduling each of these resources in an environment where each transaction is assigned one of a set of priority levels. The goal of a priority-oriented database system in this environment is to emulate a preemptive-resume server as closely as possible; that is, the goal is to insulate transactions of a given priority from the effects of any competing transactions of lower priority levels. The architectural consequences of adding priority to a DBMS are examined, and specific priority-based algorithms are proposed for scheduling each resource. The performance of these algorithms and the tradeoffs involved in priority scheduling are then studied via a simulation model. The results of this performance analysis indicate that, in spite of certain negative consequences of the use of priority, a DBMS can indeed be made to behave much like a preemptive-resume server through the use of appropriate priority scheduling algorithms.

ACKNOWLEDGEMENTS

It has been my privilege and good fortune to work with Professors Mike Carey and Miron Livny, for both of whom I have great respect. I will fondly remember the many meetings we have had to discuss our research, and how I almost always emerged from a meeting with far clearer insights than I had when it began. Both Mike and Miron have been extraordinarily patient and supportive during my development as a graduate student. I would like to take this opportunity to thank them both for all their encouragement and direction.

I would also like to thank Professor David DeWitt, who first encouraged me to step into the world of database research. Professor DeWitt has always been kind and helpful; his extraordinary enthusiasm for research has always been infectious. I am also grateful to the other members of my committee, Professors Jeff Naughton and Guri Sohi, for their constructive criticism.

The staff and secretaries of the Computer Sciences Department have been of great assistance to me during my graduate career. Without the help of the Systems Lab and the group responsible for Condor, my research would have been much harder to accomplish.

Many thanks go to all my friends who have made life fun during the last five years. Viranjit Madan, Pam Naber, Kamal Gupta, and Jaspal Kohli have been great sources of encouragement in difficult times. I would also like to thank Donovan Schneider, Shahram Ghandeharizadeh, Anoop Sharma, Murali Muralikrishna, Bob Gerber, Goetz Graefe, Beau Shekita, Mike Franklin, Sanjay Krishnamurthi, Jayant Haritsa, Scott Vandenberg, and other past and present members of the database gang at UW for their friendship; it has been a pleasure working with them. My co-residents at the Knapp House (Victor, Earl, Nancy, Jeff, Greg, Subra, Kang-Ro, John, Evan, Peter, Brita, and Masako) have also been very helpful in allowing me to maintain a sense of perspective during the last two years.

Finally, and most importantly, I would like to express my gratitude to my parents, who have always believed in me, even when I myself had doubts. From them, I have absorbed the strong commitment to the pursuit of academic excellence that has been the foundation of all my efforts in the past five years.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Motivation

Priority scheduling of computer systems is a concept that has been studied extensively for more than two decades [Coff68, Klei76]. The priority of a task is a measure of its "importance" to the system, and the objective of priority scheduling is to provide preferential treatment to tasks with higher priority values. A number of priority scheduling algorithms have been proposed for both uniprocessor and multiprocessor scheduling, and numerous theoretical results have been obtained for work-conserving priority scheduling disciplines [Jack60, Coff68, Jais68, Lamp68, Coff73, Coff76, Klei76, Buze83, Pete86, Chan87].

Despite the attention that the concept of priority has received for CPU scheduling in operating systems, it has received relatively little attention in the database management system (DBMS) area. This is somewhat surprising, as priority seems just as useful and applicable in a DBMS. In a transaction processing system, for example, it would seem desirable to ensure responsiveness for interactive DBMS users by giving their transactions high priority, while allowing batch jobs to be run in the background at low priority. In addition, one desired objective of database systems of the 1990s ("third generation" database systems, in the terminology of [CADF90]) is to provide DBMS support for rule processing. Data-intensive, rule-oriented, real-time applications such as stock trading, computer-integrated manufacturing, and command and control systems may also require the use of priority scheduling to achieve desired performance objectives [Abbo88, Abbo89, Hari90, SIGM88, Stan88b]. For example, tasks in such environments may have desired completion times or deadlines, and priority scheduling may be used to minimize the number of deadlines missed. Finally, priority scheduling may also be used to support "goal-oriented" scheduling, where the workload of a system is divided into classes, and each class is "promised" a certain level of performance (for example, a certain mean response time per class) by the system.

Priority scheduling in a DBMS differs from the problem of priority CPU scheduling due to the heterogeneity and multiplicity of resources that must be scheduled effectively in a DBMS and the fact that the use of priority introduces overheads at some of these resources. Important DBMS resources include the CPU, disks, and main memory, which are physical resources, and data items, which can be viewed as logical resources. It is common for each of these resources to be managed by an independent scheduler: The underlying operating system kernel schedules the CPU, the operating system or DBMS device drivers handle disk request scheduling, the buffer manager controls the main memory resource (i.e., buffer pool page frames), and the concurrency control manager schedules accesses to data items. As we will show, there are penalties involved with the use of priority, especially for disk scheduling, buffer management, and concurrency control; in each case, the total load on the system may increase purely as a result of priority scheduling.

In this dissertation, we will study the problem of scheduling DBMS resources when the workload consists of transactions of different priority levels. In the remainder of this chapter, we begin by providing a simple taxonomy of priority assignment policies. We then describe the preemptive-resume paradigm for making priority-based scheduling decisions. We conclude with an overview of the architecture of a priority-oriented database system, in which we identify the key components of a DBMS where priority scheduling can affect performance.

1.2. Priority Assignment

Priority may be assigned to a task either *internally* (within the system) or *externally* (by some agent outside the system). When priority is assigned internally, the system uses one or more measurable quantities to compute the priority of the task in order to achieve a desired objective. For example, the "Shortest-Job-First" CPU scheduling algorithm uses an estimate of the remaining amount of processing time required to finish a job to minimize the average waiting time per job. Alternatively, priorities may be assigned based on criteria that are external to the computer system, such as the type of usage of the system resources (e.g., "batch" vs. "interactive") or the perceived criticality of the task being scheduled (e.g., in an automated industrial control system, "watchdog" processes designed to shut down the system in emergency situations could be given high priority).

The key difference between external and internal priority assignment is that when it is assigned externally, priority is used to achieve *task-specific goals* (i.e., to provide different levels of service to different user tasks or task classes), while internal priority assignment is used to achieve *system-wide goals*. From the system's point of view, external priority

assignment represents the use of priority scheduling as an end in itself. In this case, the system is presented with a set of tasks with different priorities, and the sole objective of the system is to provide the best possible service to the task(s) with the highest priority. In contrast, internal priority assignment involves the use of priority as a means to some end. Here, the system has the added responsibility of assigning priorities to tasks with the system-wide goal in view. Thus, systems with internally-assigned priority are inherently more complex than systems where the responsibility of assigning priorities is left to external agents.

A second issue related to priority assignment is the variation of priority over time. If the priority of a task remains unchanged for the duration of the task, the task is said to have *static* priority. Alternatively, the priority of a task may change *dynamically*. For example, some operating system schedulers decrease the priority of processes as their time spent in the system increases. Both external and internal priority assignment may be either static or dynamic. Of course, whether the priority of a task is assigned statically or dynamically, its priority *relative* to that of other concurrently running tasks may change with time as the set of concurrent tasks changes. Figure 1.1 illustrates the resulting taxonomy of priority assignment policies.

In this dissertation, we will restrict ourselves to the problem of scheduling transactions whose priority has been determined externally and statically. Such a use of static priorities represents an intermediate step in the transition from conventional database systems, where no priority is used, to the full-fledged real-time database systems of the future, where priorities may be dynamic and internally assigned. An understanding of the basic performance tradeoffs in an environment

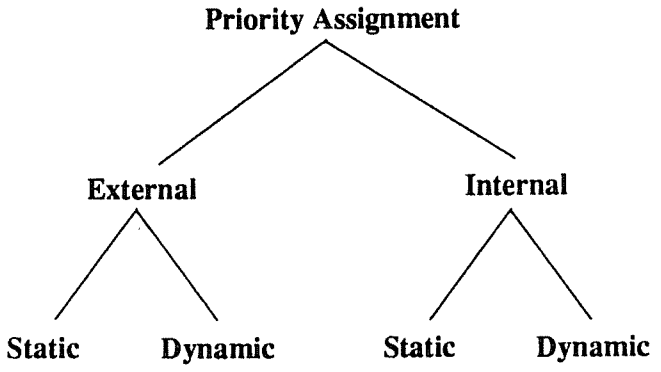

Figure 1.1: Priority Assignment Taxonomy.

where transaction priorities do not change will clearly be useful for the designers of database systems built to handle static priority assignment, and it is also likely to provide useful insights for the designers of real-time database systems.

1.3. The Preemptive-Resume Paradigm

In our work, we assume that the primary objective of a priority scheduler is to provide the best possible service to the tasks with the highest priority. That is, we would like the DBMS to behave as much like a *preemptive-resume* (PR) server [Klei76] as possible. An ideal preemptive-resume server ensures that no higher-priority task is ever made to wait while a lower-priority task receives service. Thus, if a low-priority task is being served and a high-priority task becomes ready for service, the low-priority task is preempted by the PR server and the high-priority task is given service. The low-priority task that has been preempted resumes its service at the point where it left off as soon as it becomes the task with the highest priority among those ready for service. An ideal preemptive-resume server is "work-conserving," in that no work (service requirement) is created or destroyed within the system [Klei76]. The PR server merely controls the sequence in which tasks are provided service based on task priorities, thus influencing the distribution of task waiting times and task response times while keeping the total task waiting time constant.

In a multiple-resource environment such as a DBMS, of course, it is clearly impractical to consider the entire DBMS as one PR server and to allow only the single highest-priority transaction into the system at a time. In addition, for performance as well as correctness reasons, certain database actions (such as acquiring a lock on a data item) have to be atomic or non-preemptable. Finally, it also turns out that the total amount of service required by a transaction may increase as a consequence of priority scheduling, thus violating the "work-conserving" property of ideal preemptive-resume servers. For example, if a high-priority transaction replaces a DBMS buffer page containing data that needs to be reaccessed by a lower-priority transaction, the lower-priority transaction may have to re-read the replaced page from disk later. For all these reasons, a DBMS using priority scheduling cannot be expected to behave exactly like an ideal preemptive-resume server; however, we use the underlying principles of the preemptive-resume paradigm to guide the design of our algorithms.

The fact that the use of priority in a DBMS may increase the total load on the system leads us to the formulation of a second goal for DBMS priority scheduling: the minimization of priority-induced penalties. Thus, while our primary objective is to provide the best possible service to high priority tasks like an ideal PR server, we would also like to reduce the

negative side-effects of priority scheduling on tasks of lower priority as far as possible.

Of course, approaches other than preemptive-resume can also be considered for priority scheduling. For example, one could attempt to "reserve" different fractions of the resource capacities of the DBMS for tasks of different priority. A CPU scheduler implementing this paradigm could try to enforce a rule such as the following: for every four high-priority processes given a time-slice at the CPU, one low-priority process must also be given a chance. This rule would attempt to provide a maximum of 80% of the CPU capacity of the system to high-priority tasks, and reserve 20% of the capacity for other tasks. Such reservation-based scheduling policies have several problems, however, especially in a DBMS context. For example, since data accesses and disk requests are for specific objects rather than a general-purpose resource such as a processor, the rule stated above for CPU scheduling cannot easily be translated to the context of concurrency control or disk scheduling. Furthermore, determining the appropriate fractions of system capacities for various priority levels would require some advance knowledge of the workload. For these reasons, we prefer the preemptive-resume paradigm, which represents a simple, intuitive approach to priority scheduling while requiring no advance knowledge of the workload.

1.4. Architecture of a Priority-Oriented DBMS

Figure 1.2 depicts the overall architecture of a DBMS from a resource perspective. From the figure, it can be seen that there are several places where a DBMS makes resource scheduling decisions that could be influenced by priority considerations. First, a DBMS often controls the total number of transactions allowed to be active at any one time, requiring additional transactions to wait outside the system until they can be admitted, in order to prevent thrashing due to buffer pool over-utilization [Chou85]. Second, when a transaction attempts to access or update a data item, the concurrency control algorithm of the DBMS may delay or reject such requests to ensure that the data is maintained in a consistent state. Third, as discussed earlier, the DBMS or the underlying operating system must make CPU scheduling decisions; similarly, disk scheduling decisions must be made. Finally, the buffer manager must make page replacement decisions when a non-resident page is requested and there are no free page frames.

In order to see how adding priority at some or all of these decision points might enable us to reduce the response time of high priority transactions, let us consider the composition of a transaction's response time (T_R). In order to keep this discussion simple, we assume that a locking scheme is used for concurrency control, and we also assume that the transaction under consideration does not get restarted due to concurrency control conflicts. Basically, there are two major components

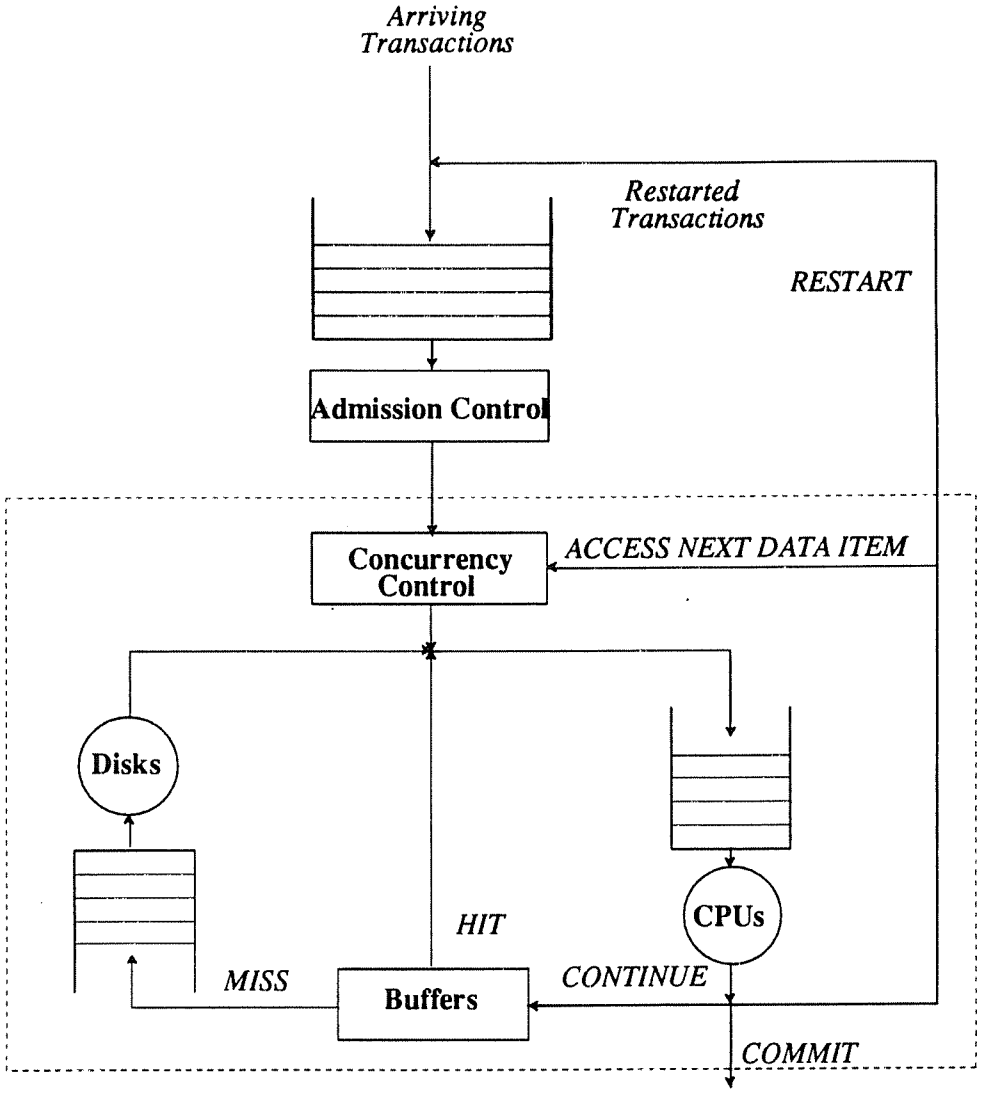

Figure 1.2: DBMS Resource Architecture.

of T_R: the external waiting time due to admission control (T_{W_EXT}), and the time spent inside the system (T_{SYS}). T_{SYS} itself has three components, lock waiting time (T_{CC}), CPU time (T_{CPU}) and disk time (T_{DISK}). T_{CC} is the time that the transaction spends blocked due to concurrency control conflicts, i.e., waiting for permission from the lock manager to access data items. The CPU time for a transaction consists of two sub-components, the waiting time due to CPU contention (T_{W_CPU}) and the actual CPU service time (T_{S_CPU}). The disk time for a transaction is the product of the number of disk requests that it makes (N_{DISK}) and the average disk access time; the disk access time also consists of a waiting time (T_{W_DISK}) and a service time (T_{S_DISK}). Thus, we see that:

$$T_R = T_{W\_EXT} + T_{CC} + T_{W\_CPU} + T_{S\_CPU} + N_{DISK}*(T_{W\_DISK} + T_{S\_DISK})$$

Incorporating priority into the admission control decision, by ordering waiting transactions according to priority and preempting lower-priority transactions in favor of higher-priority transactions, is a way of reducing T_{W_EXT} for high-priority transactions. Using priority in the concurrency control algorithm, such as by granting pending lock requests in priority order, is a way of reducing T_{CC} for high-priority transactions. In terms of CPU time, priority-based CPU scheduling can reduce T_{W_CPU} for high-priority transactions; T_{S_CPU} cannot be reduced, however. Similarly, T_{W_DISK} can be reduced through the priority-based scheduling of disk requests. Using priority to schedule disk requests instead of serving disk requests in positional order may actually lead to an increase T_{S_DISK}, however, due to an increase in the expected seek time per disk access. Finally, buffer replacement decisions can potentially reduce N_{DISK} for high-priority transactions. Note that, when a transaction gets restarted due to concurrency control conflicts, each of the terms constituting its response time will represent the accumulated effects of the original execution as well as each restarted execution. For example, if a transaction is restarted n times, its T_{W_EXT} will include the time spent blocked outside the system when it was initially submitted, as well as the time spent blocked after each of the n restarts. Clearly, priority-based concurrency control can affect the number of restarts as well. Of course, none of these reductions for high-priority transactions will come for free. Rather, we must expect to see corresponding increases in the response time components of transactions of lower priority. In addition, there may also be some overhead involved in implementing the priority-based scheduling algorithms, such as increased path lengths in the code for buffer management and the cost of priority-induced context switching at the CPU.

1.5. Organization of the Dissertation

The remainder of this dissertation is organized as follows: Chapter 2 reviews relevant earlier work in the area of resource scheduling and discusses new DBMS applications that may require priority scheduling. Our algorithms for priority-based scheduling of DBMS resources are described in Chapter 3. In Chapter 4, we describe a performance model of a priority-oriented DBMS. Chapter 4 also includes an overview of our experimental methodology and the workload used in our simulation experiments. The results of simulation experiments conducted to understand the performance implications of using priority scheduling in a DBMS are presented in Chapters 5, 6, and 7. Lastly, Chapter 8 summarizes the contributions of this work and presents our ideas for future research in related areas.

CHAPTER 2

SURVEY OF RELATED RESEARCH

While little has been published concerning DBMS scheduling in the context of static priorities, there is a significant amount of published research in certain related areas. For example, priority scheduling theory has been studied extensively for over two decades. In practice, priority scheduling has been considered for use in operating systems; in this area, most of the research has focused on processor scheduling, given various levels of information about job service time requirements. Recently, there has also been some work in the area of deadline scheduling for real-time database systems. The remainder of this chapter contains a brief summary of published research in each of these areas.

2.1. Priority Scheduling Theory

Priority scheduling provides preferential treatment to some customers over others in an effort to maximize system "profit" or minimize system "cost," where the profit and cost functions may vary from one system to another. In general, one of two approaches may be used to schedule the service of a high priority task that arrives when the server is busy with a low priority task. Under the "head-of-the-line" (HOL) or non-preemptive policy, the new arrival is inserted into the priority queue but does not receive service until the low priority task completes service. In contrast, under the "preemptive resume" (PR) policy, the low priority task is interrupted and deprived of the server so that the high priority task can begin service immediately; the interrupted task resumes service after the high priority task completes. Of course, in both approaches, waiting tasks are queued in priority order. Much of the research in the area of priority scheduling has assumed a collection of one or more identical CPUs for servers. [Jais68] contains an analysis of the HOL and PR disciplines for single-server systems with Poisson arrivals and arbitrary service time distributions (i.e., M/G/1 systems). [Buze83] provides response time results for PR scheduling in M/M/m systems with a centralized queue.

A general theoretical treatment of priority scheduling theory is contained in [Klei76]. In the queueing model used here, the priority of a task may be based on a combination of the following factors:

(1) the arrival time of the task relative to that of other tasks waiting for the resource;

(2) the task service time required or received so far;

(3) an externally assigned value or "group membership."

Examples of queueing disciplines that depend only upon arrival time are First-Come-First-Served (FCFS) and Last-Come-First-Served (LCFS); policies that discriminate based on service time include Shortest-Job-First (SJF), Shortest-Remaining-Time-First (SRTF), and Longest-Job-First (LJF). Both preemptive and non-preemptive scheduling is considered, and analytical solutions are provided for calculating the average waiting time for several priority scheduling disciplines.

Also provided in [Klei76] is a summary of priority scheduling disciplines for conservative systems (i.e., systems where there is no cost for preemption and no server is idle when jobs are present). Three possible information states are considered relative to task service times: (1) exact service time information is available *a priori*, (2) only the distribution of service times is known, and (3) no information at all is available. For each level of information, the optimal scheduling discipline for minimizing different cost functions is given, for both preemptive and non-preemptive cases. For example, among preemptive algorithms for G/G/1 queues where each job has an identical linear cost and job service times are known, the Shortest-Remaining-Time-First scheduling algorithm is known to be optimal. Finally, as an example of externally-assigned priority, a study of optimum bribing for queue position is presented. Each customer is allowed to "buy" a relative priority by means of a bribe given to the scheduler, and the goal of the scheduler is to maximize the total bribes received. Other sources that provide overviews of priority scheduling theory include [Coff68], [Lamp68], [Coff73], and [Coff76].

Of course, in real systems, there may be overheads incurred as a result of the use of priority, particularly when scheduling is preemptive. For example, context switches may be expensive, especially in shared-memory multiprocessor systems or vector machines. In an effort to minimize preemption costs, [Coff76] describes a policy where preemption is considered only if the remaining service time of the task in progress is above a certain threshold. Clearly, priority scheduling may also result in the starvation of low priority tasks. In order to avoid starvation, [Jack60] proposed a time-out priority rule that increases the priority of a task when its waiting time exceeds a threshold.

2.2. Scheduling for Real-Time Systems

If the correctness of a computing system depends not only on the logical results of the computation performed, but also on the timeliness with which the results are produced, the system is said to be a "real-time" or "time-constrained" system. More specifically, as explained in [Stan88b], the objective of real-time computing is to meet the individual time requirements of each of a set of tasks, rather than to minimize the average response time (or maximize the throughput) for the entire task set. Clearly, some form of priority scheduling is likely to be required for real-time systems.

A model of generalized time constraints for real-time systems is described in [Jens86]. In this model, the time constraint placed on a task is captured by a function mapping the completion time of the task to a "value." The value of a task at any given time t represents the relative importance to the system of having completed that task successfully at time t. Clearly, each task must have a positive value for some range of completion time if it is of any use at all to the system. If the value of a task abruptly drops to zero (or becomes negative) at some value t_h of completion time, the task is said to have a "hard" deadline at t_h. Even if a task does not have a hard deadline, it may still have a point of discontinuity t_s in its value function that represents a sharp downward change in the value of the task at that time. In this situation, the task is said to have a "soft" deadline or "critical time" at t_s. Alternatively, the value of a task may just taper off more gradually, indicating that there is no specific "critical time" or deadline for the task. This model of time constraints was developed for use in a real-time operating system, but it can be extended to other computing systems. For example, [Stan88a], [Abbo88], and [Buch89] include discussions of how appropriate value functions might be constructed for database transactions.

2.2.1. Scheduling in Real-Time Operating Systems

The focus of research in the area of real-time operating systems has been on providing simple, fast primitives and on developing scheduling heuristics for meeting hard deadlines. Recently, commercial operating systems such as UNIX have been enhanced to provide real-time support. For example, in Real-Time enhanced UNIX (RTU), as described in [Conc90], programs can be classified as either "time-sharing" or "real-time." Real-time programs have statically assigned priorities and are not given time slices like time-sharing programs. Instead, preemptive scheduling is used, where the highest-priority real-time program that is ready for execution runs until it exits, blocks, or is interrupted by a real-time program of even higher priority. In the absence of real-time programs, normal time-sharing programs are scheduled in the conventional manner; in effect, non-real-time programs are all assigned the same lowest priority in the system. Other features of

RTU include memory locking (to allow parts of a high-priority process to be locked in main memory to prevent paging), fast inter-process communication, and faster disk I/O. Special-purpose operating systems have also been built for specific real-time applications. For example, a flexible set of operating system primitives suited to high-performance robotics and real-time control systems is proposed in [Schw87]. Using these primitives, which include fast prioritized messages and low-cost processes, the operation of a complex robot (a six-legged, three-ton vehicle) requiring extensive real-time computations is supported.

More general scheduling problems are discussed in [Zhao87a] and [Zhao87b]. [Zhao87a] describes a heuristic approach to the problem of dynamically scheduling non-preemptable tasks in a distributed real-time environment, where tasks have hard deadlines and may have to wait for resources such as CPUs and I/O devices. A heuristic function is used to select the task to be scheduled next. The function consists of three weighted factors, representing the resource requirements of the task, its laxity[1], and its worst-case computation time. By adjusting the weights associated with each of these three factors and using backtracking techniques, the performance of the heuristic approach can be made quite close to that of exhaustive search. In [Zhao87b], heuristic algorithms for scheduling preemptable tasks under time and resource constraints are developed and their performance is analyzed. Resources can be utilized in either "exclusive" or "shared" modes in this study. Limited backtracking in the construction of schedules again results in significant performance improvements.

A study of dynamic scheduling algorithms for distributed soft real-time systems is presented in [Chan87]. In this study, the scheduler has two objectives: first, it attempts to minimize the percentage of jobs missing their deadlines; second, it attempts to ensure that the N jobs being served in an N-processor distributed system are those having the highest priorities among all the jobs in the system. A number of protocols are proposed for transferring jobs among processors in order to achieve these goals. For example, a receiver-initiated protocol, where a processor examines its queue after each job completion, polling other processors to offer help if it finds itself underloaded, is found to be quite robust for minimizing deadline misses.

---

[1]Informally, the laxity of a task is a measure of the maximum amount of time that the scheduler can delay the execution of the task while still meeting the task's deadline.

The need to synchronize tasks that share logical or physical resources in a real-time processing environment can lead to situations where high-priority jobs are forced to wait indefinitely for low-priority jobs. This phenomenon, called *priority inversion*, is discussed in [Sha87]. As an example, consider three jobs J_1, J_2, and J_3, where J_2 has a priority greater than J_1, and J_3's priority exceeds that of J_2. Assume that J_3 and J_1 share a data structure guarded by a semaphore S, and that J_2 does not require to access this data structure. Let J_1 begin execution first and lock S at time t_1. Before J_1 releases its lock on S, J_3 arrives, preempts J_1, and tries to use the shared data. In this situation, J_3 will be blocked on S. Ideally, J_3 (having the highest priority of the three jobs) should be blocked no longer than the time needed for J_1 to complete its critical section. However, the duration for which J_3 is blocked is in fact unpredictable, because any job of intermediate priority, such as J_2, can preempt J_1 while the latter is in its critical section. A solution to this problem, called "priority inheritance," is suggested in [Sha87]; the basic idea is that whenever a job J blocks higher priority jobs, J is made to execute at the highest priority level among the blocked jobs. In the above example, J_1's priority would be made equal to that of J_3 until it releases its lock on S, ensuring that J_1 cannot be preempted by an intermediate-priority job before it completes its critical section.

2.2.2. Scheduling in Real-Time Database Systems

The problem of scheduling real-time database activities is somewhat more complicated. The amount of disk and processor use required by a transaction depends on such factors as the amount of data accessed, the buffer management algorithm used, and the concurrency control algorithm employed.

Algorithms for scheduling transactions under deadline constraints in a single-processor, main-memory database system were analyzed in [Abbo88], and extensions for an environment with disk-resident data were presented in [Abbo89]. A transaction was modeled simply as a set of alternating data and CPU requests; each transaction had a start time, a deadline, and a run time estimate associated with it. Three aspects of transaction scheduling were investigated — eligibility criteria, priority assignment, and concurrency control. Eligibility criteria were used to determine whether a transaction should be allowed to begin or continue execution. Priorities were assigned to transactions according to different protocols: FCFS, earliest deadline, or least estimated slack. The slack of a transaction (like the laxity of a task, described earlier) is a measure of the amount of time that the scheduler can delay the execution of the transaction and still meet the transaction's deadline. Finally, data consistency was enforced either by strict serial execution of transactions, by concurrency control

algorithms based on prioritized variations of the Wound-Wait algorithm [Rose78], or by an algorithm based on priority-inheritance. The performance of different combinations of eligibility, deadline assignment, and concurrency control algorithms was compared. The main performance metric considered was the number of missed deadlines. The use of eligibility tests for screening out transactions that are deemed unlikely to meet their deadlines was found to result in improved performance. For priority assignment, the least slack algorithm worked the best. For concurrency control, the priority-inheritance protocol, called "Wait-Promote," was found to offer the best performance. It was also shown in [Abbo89] that when the I/O system is heavily loaded, using priorities to schedule I/O requests yields significant performance gains over scheduling I/O requests in a FIFO manner. However, the disk scheduling algorithms studied did not take the physical (track) addresses of disk requests into account. While this was an excellent first study of real-time scheduling for database systems, the study's transaction and buffer pool models limit the validity of their conclusions somewhat. For example, buffer replacement policies were not included in the model, and transactions consisted of sequences of page accesses chosen at random from a global "database" rather than more structured accesses to relations via relational operators such as joins and selections.

Several other studies of concurrency control for real-time databases have recently been published as well. In [Hari90], it was demonstrated that, under a policy that discards transactions whose deadlines are not met, optimistic concurrency control may outperform locking algorithms in a real-time DBMS where priority is used to schedule physical resources. Another argument used in discussions of concurrency control in real-time systems is that real-time applications are concerned more with timely responses to important requests than with maintaining perfectly consistent data; that is, it may be advisable to sacrifice consistency to some degree in order to improve responsiveness. In [Sha88], it was suggested that the database be divided into "atomic data sets" (ADSs) such that the consistency of each ADS can be maintained independently of the others; transactions can then be decomposed into "elementary transactions" such that each elementary transaction maintains the consistency of the ADS accessed by it. Each transaction, along with its elementary transactions, had a priority assigned to it. A locking protocol based on priority inheritance was used to maintain serializability of elementary transactions with respect to each ADS. The idea of hierarchically decomposing databases in order to allow increased concurrency is also discussed in [Hsu83] and [Hsu86]. For real-time distributed databases, another potential method of enhancing timeliness is through extensive replication of data; a quorum-based concurrency control algorithm that supports three different levels of consistency is described in [Lin88]. An alternative model of real-time database

processing, where deadlines are associated with consistency constraints rather than explicitly with transactions, is provided in [Kort90]. A predicate-based approach to transaction scheduling is proposed in order to detect inconsistencies and take corrective actions.

Recently, an empirical evaluation of real-time transaction processing on a testbed system was presented in [Huan89]. The testbed consisted of a centralized DBMS implemented as a set of server processes on top of the VAX/VMS operating system. Several algorithms for handling CPU scheduling and concurrency control conflicts based on transaction values and deadlines were analyzed. In addition to demonstrating the value of deadline-based CPU scheduling empirically, this study also showed that the costs of locking and message communication can be significant overheads in real-time systems. The testbed was limited, however, in that it did not support deadline-based disk scheduling, and buffer management issues were also not examined in this study.

CHAPTER 3

ALGORITHMS FOR PRIORITY SCHEDULING

As described in Chapter 1, there are four key resources in a DBMS where priority scheduling can make a difference: the processors, the disks, the main memory buffers, and the data itself. In this chapter, priority scheduling algorithms for each of these resource types are presented. A known algorithm for CPU scheduling is presented, while the algorithms for disk scheduling, buffer management, and concurrency control are essentially extensions or combinations of conventional algorithms, modified to enable them to handle priorities. A brief discussion of our assumptions precedes the description of the algorithms themselves.

3.1. Assumptions

We assume that arriving transactions belong to one of a set of P different priority classes, indexed by the subscript p ($p = 1, 2, ..., P$). In the following discussion, we adopt the convention that the *larger* the value of the index associated with the priority group, the *higher* is its priority; that is, transactions from priority group p are given preferential treatment over transactions of priority group p-1. For each resource, we assume that the scheduler's guiding philosophy is to emulate the Preemptive-Resume paradigm as far as possible. That is, transactions of group p should "see" competition only from transactions of equal or higher priority; the scheduling algorithm should "hide" the existence of transactions of groups p-1, p-2, ..., 1 from transactions of group p. Of course, the Preemptive-Resume discipline applies only across priority groups. Within a priority group, our approach is to use the conventional service discipline that is most appropriate in the absence of priority; for example, a positional disk scheduling algorithm such as the Elevator algorithm should be used to schedule a set of disk requests of equal priority.

3.2. CPU Scheduling

A number of options for priority CPU scheduling have been discussed in the operating systems literature [Coff68, Klei76, Pete86]. Straightforward preemptive-resume priority scheduling, where a high-priority job preempts a low-priority job, and the low-priority job resumes its CPU processing after all high-priority jobs have completed their processing, does not seem entirely suitable for database systems. This is because preempting transactions while they hold short-term locks or latches may lead to the convoy problem, as described in [Blas79]. It seems more reasonable, then, to use a non-preemptive form of priority scheduling, while ensuring that high-priority requests do not suffer greatly as a result of the non-preemptability of low-priority requests. In order to reconcile priority with non-preemptability, CPUs can be scheduled according to a priority-based round-robin scheme where the length of a CPU slice is determined by the transaction and not by the scheduler. In this scheme, a transaction gives up the CPU at a "safe point" after a short burst of CPU use. Once a CPU is released, it is assigned to the transaction at the head of the CPU queue. The queue is managed in priority order, with requests of the same priority being served in FCFS order. When there are several processors in the system, the actual CPU where a request is processed is selected at random from among the idle CPUs, if any.

3.3. Disk Scheduling

Classical disk scheduling schemes such as the *shortest seek time first* and *elevator* algorithms [Teor72, Pete86] are *positional* in that they attempt to minimize the average seek distance instead of servicing disk requests in FCFS order. Since the elevator algorithm performs especially well under high disk loads and has good fairness characteristics, it is a good candidate to serve as the basis of a priority scheduling algorithm at the disk.

In the elevator algorithm, the disk head is either in an inward-seeking phase or an outward-seeking phase. While seeking inward, it services any requests that it passes until there are no more requests ahead. The disk head then changes direction, seeking outward and servicing all requests in that direction as it reaches their tracks. Figure 3.1 shows an example of the operation of the elevator algorithm where the disk under consideration has 1000 tracks. At the top of the figure is a set of outstanding disk requests arranged in order of their track positions. In this example, the head of the disk is initially at track 503 and is in an outward seeking phase; i.e., the head is moving towards higher track addresses. Five disk requests are serviced in order of increasing track address; then the head reverses direction, serving requests in reverse order until all thirteen requests have been served. The disk head has to move across a total of 1300 tracks in serving thirteen requests.

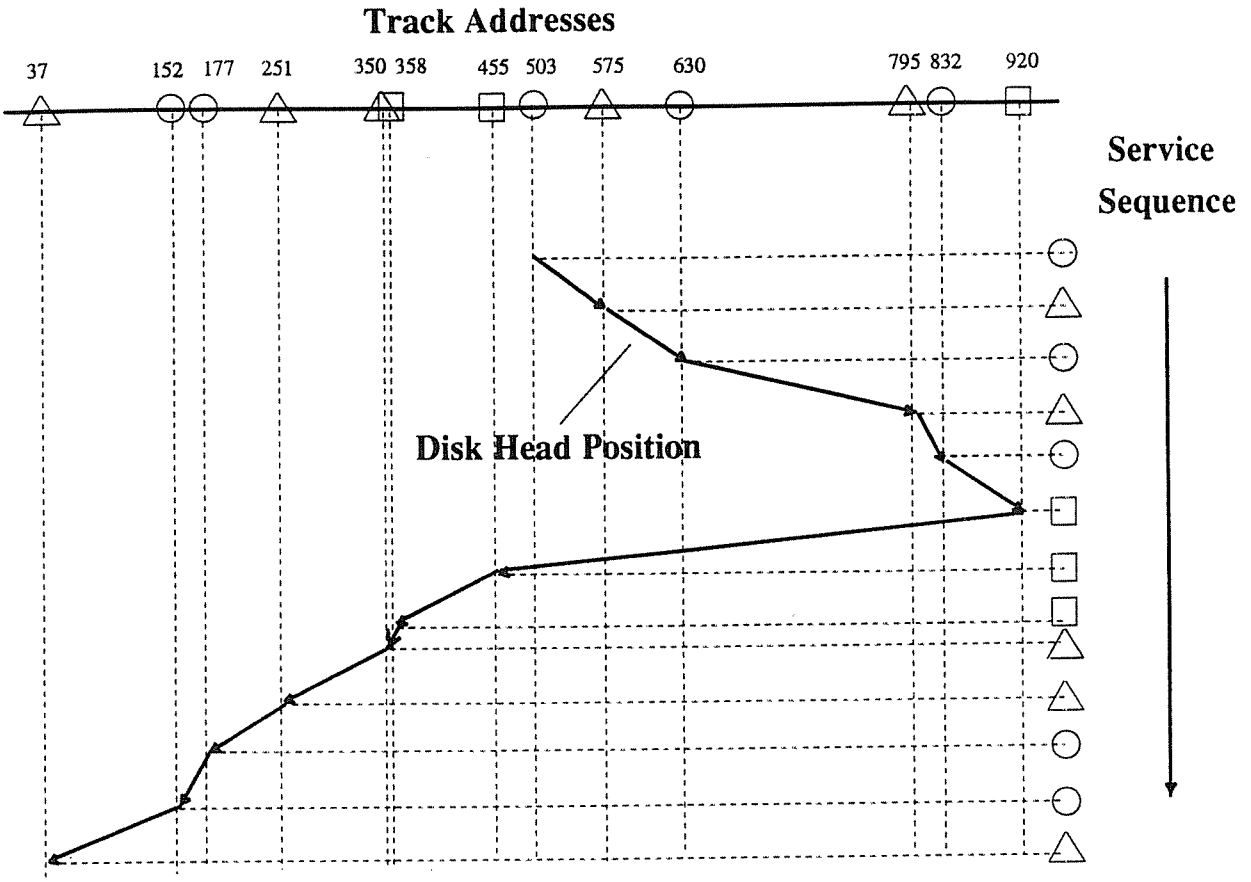

Figure 3.1: Elevator Disk Scheduling.

In order to support priority, the elevator algorithm can be modified in the following way: disk requests are grouped on the basis of their priority, and the elevator algorithm is used within each group. There is one queue per priority level for buffering outstanding disk requests. Within each queue, requests are arranged in order of their physical (track) addresses. While seeking inward or outward, the disk services any requests that it passes in the currently-served priority queue until there are no more requests ahead. On the completion of each disk request, the scheduler checks to see whether a disk request of a higher priority is waiting for service. If such a request is found, the scheduler switches to the queue that contains the request of the highest priority among those waiting and starts serving that queue. When it switches to a new queue, the request in the queue with the shortest seek distance from the head's current position is used to determine the direction in which the head will move. Figure 3.2 illustrates the working of the Prioritized Elevator algorithm. As in Figure 3.1, there are thirteen disk requests, but now they are grouped into three priority levels. The scheduler first services the

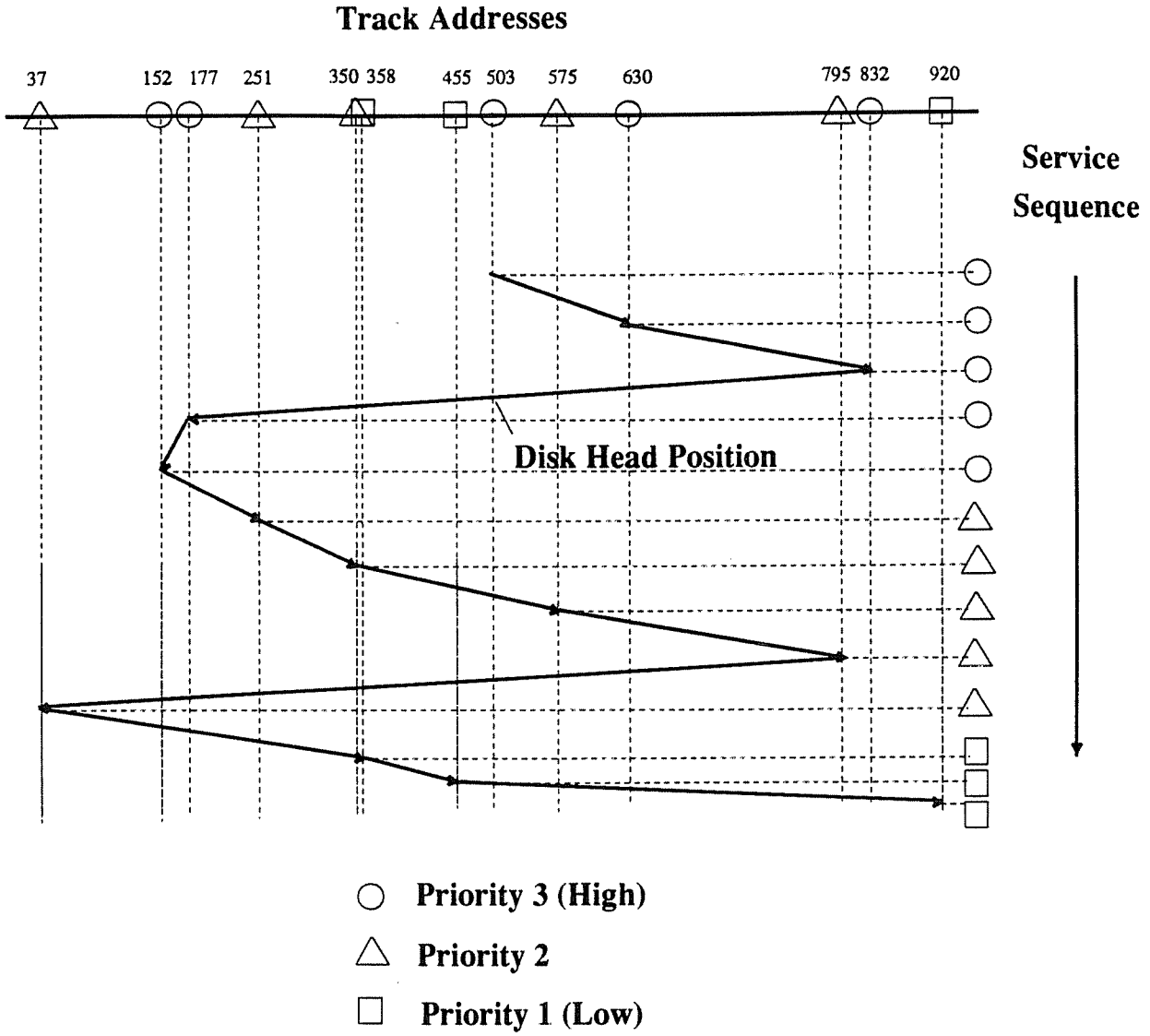

Figure 3.2: Prioritized Elevator Disk Scheduling.

five requests having the highest priority, using the elevator algorithm within this group. After the priority 3 requests have been served, the scheduler uses the elevator algorithm among the requests of priority 2, and then it finally serves the requests of the lowest priority.

An important side-effect of introducing priority in disk scheduling in this fashion is that the average seek time can worsen as the number of priority levels increases. Although the sets of requests serviced in Figures 3.1 and 3.2 are identical in terms of track positions, the total seek distance has increased from 1300 tracks to 3293 tracks as a result of the use of

priority. Since seek time is proportional to the square root of seek distance [Bitt88], this represents a significant increase in the average seek time over Figure 3.1. In the worst case, if each request had a different priority, the prioritized elevator algorithm would degenerate into straight priority scheduling, losing the advantages of a positional disk scheduling policy. This suggests that, in order to provide reasonable I/O performance, it will be preferable to map disk requests into a small number of priority levels at the disk scheduler (even if each transaction has a distinct priority in other parts of the system).

3.4. Buffer Management

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. 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 DB2 buffers (sequential or random). 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.

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 could lead to situations of priority inversion. Furthermore, the set of concurrently active transactions in these operating regions may include transactions of different priority levels. Priority-based buffer replacement policies are thus required in order to provide preferential service to high-priority transactions. We 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]

In this section, we describe a new buffer management algorithm called *Priority-Hints* that makes use of hints provided by the database access methods (as in Starburst [Haas90] and DB2 [Teng84]). Priority-Hints uses these hints to make priority-based buffer management decisions while trying to minimize the priority-induced overhead on the system. In addition to Priority-Hints, we also present two other new priority-based buffer management algorithms, *Priority-LRU* and *Priority-DBMIN*. Priority-LRU is a modification of the commonly used Global LRU buffer management policy [Effe84], extended to handle priority; similarly, Priority-DBMIN represents an extension of the DBMIN policy of [Chou85]. Note that DBMIN is a sophisticated policy that uses information about data access patterns to provide significant performance gains over Global LRU and other simpler algorithms that do not use such information. We chose to extend these two existing buffer management policies because they represent two ends of the buffer management algorithm spectrum; thus, their prioritized versions would seem to be logical candidates against which the performance of Priority-Hints can be compared. As will become clearer as the algorithms are described, Priority-Hints falls somewhere in between Priority-DBMIN and Priority-LRU in the amount of data access pattern information required by the buffer manager; consequently, a comparison of the three algorithms will illustrate the tradeoffs between algorithm complexity and buffer management performance.

Our assumptions about buffer management are outlined first. We then describe Priority-Hints, Priority-LRU and Priority-DBMIN. Our scheme for handling dirty data, which is common to the three algorithms, is described next. We conclude the section with a summary of the key differences between the three algorithms.

3.4.1. 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

[1]This is in contrast to current technology 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 almost irrelevant.

executing transactions that have accessed the page since it was last 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, discussed in Section 3.4.5) 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 owned by the transaction are freed. The buffer manager considers *reactivating* suspended transactions at the same decision points 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 to waiting transactions of higher priority.

3.4.2. Priority-Hints

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:

---

[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.

(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 3.3, there are three transactions, three priority levels, and no free buffers. Priority decreases from the top to the bottom of the figure. Transaction T1 is at priority 3, T3 is at priority 2, and T2 is at priority 1.

A transaction set consists of two kinds of buffers: the buffers currently fixed by the owner (marked by the letter "F" in Figure 3.3), and buffers containing unfixed favored pages of the owner (marked by the letter "U" in Figure 3.3). The unfixed favored pages are maintained in MRU order with the help of buffer timestamps. Note that a transaction set

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

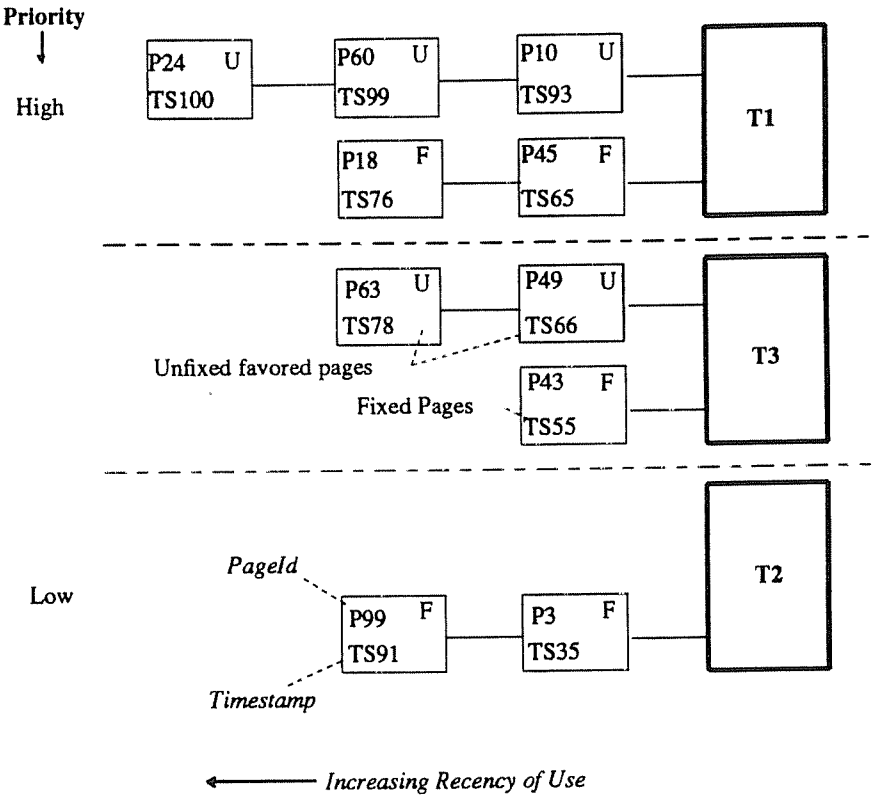

Figure 3.3: Example of Priority-Hints Buffer Pool Organization.

contains no unfixed normal pages; whenever a normal page is unfixed, it is placed on the global free list.

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

[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 or suspended transactions of the same priority) that should first attempt to enter the system.

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 3.3, if T1 makes a buffer request for page P37, 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 P63 as the replacement victim. Had there been no unfixed pages of priority 1 or 2, then P24 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 normal pages. (Recall that normal pages are placed on the free list at unfix time, and that the free list is maintained in LRU order.)

3.4.3. Priority-LRU

Global LRU [Effe84] is a buffer replacement policy based on the assumption that there is a temporal locality of data references in relational database operations. Thus, when a buffer frame is required by a transaction, and no free frame is available, the frame with the least-recently-accessed data (from among all the frames in the buffer pool) should be selected for replacement. The policy is global in that all the frames in the buffer pool are treated according to a single criterion

(recency of usage) for allocation as well as for replacement; thus, there is no difference between the policy used for replacement and the policy used for allocation.

Buffer Pool Organization

In Priority-LRU, our prioritized version of Global LRU, the buffer pool is organized dynamically into priority levels in the following way. At system startup time, all the buffer frames are free, and are arranged in a free list. When a transaction with priority *p* is allocated a frame that was previously free, the frame is inserted in an LRU queue of frames whose owners have priority *p*. Thus, if there are transactions having *p* different priority levels at any given time, the buffer pool consists of *p* LRU queues (one per priority level), the free list, and the dirty list. Figure 3.4 shows an example of the organization of a buffer pool for Priority-LRU; there are three priority levels, and thus three LRU queues, and there are no free frames in the example. Priority increases as we move from the bottom to the top of the figure, and the least recently-used page of each queue is in the rightmost frame in the figure.

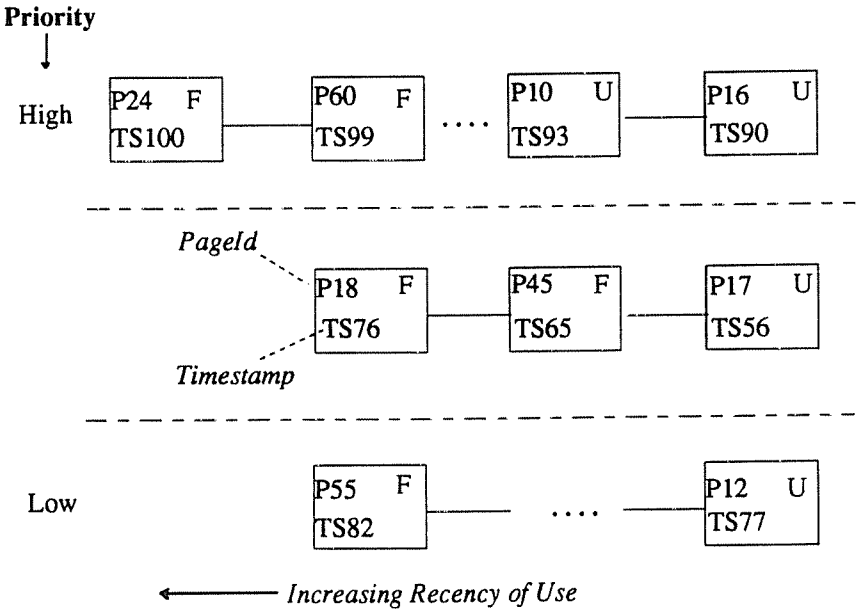

Figure 3.4. Example of Priority-LRU Buffer Pool Organization.

Transaction Admission

Priority-LRU's transaction admission policy is exactly the same as that of Priority-Hints.

Buffer Replacement and Allocation

As for the conventional Global LRU policy, the buffer allocation and replacement policies coincide in our algorithm. The key idea of the replacement policy is that the least recently used page of the lowest priority should be chosen as the victim. In the example of Figure 3.4, when a transaction of priority 3 needs to read a page into the buffer pool, P12 would be the victim chosen; had there been no unfixed pages at priority 1, then P17 would have been chosen.

3.4.4. Priority-DBMIN

As discussed in [Chou85], the primitive operations (e.g., selections, joins) 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, and these patterns are known to the query optimizer. The DBMIN buffer management policy makes use of this information using the following key ideas:

(1) Buffers should be allocated to transactions on a "per file instance" basis: i.e., since the pattern of accessing each file used by a transaction can be different, a different set of buffers (called a "locality set") should be allocated to a transaction for each file that it opens.

(2) For each file instance *Fi*, there is an optimum number of buffers ($OptBufs_{Fi}$) and an optimum replacement policy ($RepPol_{Fi}$). As long as the number of buffers actually allocated to file instance *Fi* is less than $OptBufs_{Fi}$, the admission policy (see (4)) guarantees that there will be at least one free buffer available for *Fi*; when the number of buffers allocated to *Fi* is equal to $OptBufs_{Fi}$, $RepPol_{Fi}$ is used to choose a victim from *Fi*'s locality set when replacement is required. Thus, the replacement policy in DBMIN is local rather than global.

(3) The query optimizer can inform the buffer manager of $OptBufs_{Fi}$ and $RepPol_{Fi}$ for each file instance *Fi*. The buffer manager can then ensure that the maximum number of buffers allocated to a file instance is $Optbufs_{Fi}$.

(4) The buffer manager ensures that no transaction is allowed to begin running unless it can be guaranteed to get the optimum number of buffers for each of its file instances.

By using reference locality information *before* allowing a transaction to begin execution, DBMIN ensures that the transaction's buffer requirements will be satisfied once it actually begins to run. This is why DBMIN was found to perform so well in [Chou85].

Buffer Pool Organization

As in the original DBMIN algorithm, the buffer pool is organized into "locality sets," where the data pages in each set are all part of the same file instance and have the same owner. If a page is accessed by more than one concurrent transaction, its owner is the transaction with the highest priority among them. Within a locality set, pages are arranged according to the replacement policy prescribed by the optimizer. The buffer manager maintains the sum of the *OptBufs* values for each priority. Thus, for any priority level p, it is easy to compute the sum of the *OptBufs* values for all higher priority transactions that are running.

Transaction Admission

As discussed earlier, DBMIN relies heavily on its Transaction Admission policy. When transactions have priorities, however, the guarantee that any transaction allowed to run will always find *OptBufs* buffers available must be made conditional in the following way. Let the combined size of the locality sets of transaction T be $OptBufs_T$, and let its priority be p_T. Let the sum of the optimum sizes of the locality sets of running transactions with priority $\geq p_T$ be $OptBufs_{HIGHER}$, and the sum of the optimum sizes of the locality sets of transactions with priority $< p_T$ be $OptBufs_{LOWER}$. Then, if

$$(1)\quad (N - OptBufs_{HIGHER}) \geq OptBufs_T,$$

where N is the total number of buffers in the buffer pool, the buffer manager admits T. The idea is that if there are sufficient buffers for all transactions of priority p_T or higher currently running, as well as for T itself, then T should be allowed into the system. Note that if

$$(2)\quad (N - OptBufs_{HIGHER} - OptBufs_{LOWER}) < OptBufs_T,$$

then some buffers may have to be deallocated from running transactions in order to satisfy T's buffer requirements. When both (1) and (2) are true, the buffer manager will successively suspend transactions (starting with the transaction with the minimum priority) until (2) becomes false.

Buffer Replacement and Allocation

Buffer allocation and replacement are the same as in the original DBMIN algorithm.

3.4.5. Dealing with 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 special priority, higher than that of any read requests. 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]

---

[5]The reason that the engine checks whether each page is old enough to flush, rather than just 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.

3.4.6. Discussion of Buffer Management Algorithms

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 (as opposed to favored), 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 the 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.

(4) Priority-Hints allows a transaction to execute even when it does not have an optimum number of favored pages, as does Priority-LRU, while Priority-DBMIN does not.

(5) 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 by other transactions, including those of higher priority, as long as the latest arrival has some unfixed favored buffers. Thus, the performance degradation caused by stealing favored buffers is limited to one transaction at a time in Priority-Hints.

---

[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.

(6) 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 in the access method code 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.

(7) 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 support a local replacement search strategy. Secondly, unfixed buffers are arranged in MRU order in Priority-Hints, unlike in DB2 or Starburst. These two factors can have a significant impact on performance, as will become clear in Chapter 6.

3.5. Concurrency Control

Concurrency control has long been a fruitful area for database research; a large number of algorithms have been developed for both centralized and distributed databases. Most of these algorithms are based on one of three basic mechanisms: *locking* [Mena78, Rose78, Gray79, Lind79], *timestamps* [Reed78, Bern80], and *optimistic* concurrency control [Casa79, Kung81]. A performance analysis of different mechanisms is presented in [Agra87]. There it is shown that blocking algorithms, i.e., algorithms that tend to conserve physical resources by making transactions wait rather than restarting them, are likely to perform better than restart-oriented algorithms in an environment where the physical resources are limited. In an environment where physical resource utilizations are low, however, a restart-oriented algorithm may be the better choice.

In a priority-oriented DBMS that strives to behave like a preemptive-resume server, high priority transactions should not be forced to wait for transactions of lower priority. In fact, if priority scheduling is used at the physical resources, forcing high priority transactions to wait for locks held by transactions of low priority could result in significant performance degradation due to priority inversion; low priority transactions holding locks would make little progress due to contention for physical resources, while high priority transactions could be blocked due to data contention. One solution to this priority inversion problem is to increase the priority of the lock holder to that of the requester via priority inheritance [Sha87]. This may have negative side-effects, however. Consider the case of a long, low priority transaction L that acquires an exclusive lock early in its execution. If a high priority transaction H attempts to acquire this exclusive lock, and the priority

of L is increased to that of H to avoid priority inversion, then L will execute at high priority until it commits. This will lead to greater contention for resources among transactions at H's priority, and L may also block other high priority transactions outside the system. Furthermore, in an environment with high data contention, priority inheritance could potentially lead to the entire set of concurrent transactions being run at high priority, thereby defeating the purpose of priority scheduling. These considerations led to the design of the *Wound-Wait/Two-Phase Locking* (WW/2PL) algorithm for priority-based concurrency control.

We precede our discussion of the WW/2PL algorithm by a review of two existing algorithms that provided the key ideas on which WW/2PL is based. Each algorithm employs the locking mechanism and supports two conflicting lock modes: *read* locks, which may be shared by a number of transactions, and *write* locks, which are held by one transaction at a time.

3.5.1. Two-Phase Locking and Prioritized-Wound-Wait

In [Gray79], the basic *Two-Phase Locking* (2PL) protocol is described for concurrency control in a conventional DBMS. In 2PL, if a data item is locked by one transaction, any transaction that requests a conflicting lock on the item blocks until the original lock is released. A "waits-for" graph of the blocked transactions is maintained, and deadlock detection is performed whenever a transaction blocks. When a deadlock is discovered, the youngest transaction in the deadlock cycle is aborted. In order to choose deadlock victims, transactions are assigned unique timestamps based on submission time.

An algorithm called *High Priority* (HP) was introduced for priority-based concurrency control in [Abbo88]. The key idea of the HP policy is that lock conflicts are always resolved in favor of the transaction with the higher priority.[8] That is, if a lock is held by a transaction of lower priority than the requesting transaction, the lock holder is restarted and releases its locks. If the requester's priority is less than or equal to that of the holder, however, the requester blocks, as in 2PL. If each transaction has a unique priority, deadlocks cannot occur in the HP policy. In the static priority environment, however, deadlocks among transactions of a given priority level are a possibility; in order to prevent them, our version of HP

[8]Actually, several variations of HP were introduced in [Abbo88] and [Abbo89], including one based on priority inheritance. Here we describe the simplest version.

enforces uniqueness by using transactions' ages to break priority ties. Since HP is essentially a priority-flavored extension of the Wound-Wait algorithm of [Rose78], we will refer to our version of HP as the *Prioritized-Wound-Wait* (PWW) algorithm in subsequent discussions.

3.5.2. Wound-Wait/Two-Phase Locking

The Wound-Wait/Two-Phase Locking (WW/2PL) concurrency control algorithm is a hybrid of the 2PL protocol and the PWW algorithm. The basic idea, as in PWW, is to restart a transaction if it holds a lock requested by a transaction of higher priority in a conflicting mode. The key difference between WW/2PL and PWW is that WW/2PL anticipates that many transactions may have the same priority, while PWW was designed for a deadline environment where transactions typically have unique priorities. Therefore, WW/2PL uses two-phase locking with deadlock detection for conflicts within a priority level, whereas PWW uses timestamp-based deadlock prevention. Note that in WW/2PL, deadlocks can occur only among transactions of a single priority level, as the PWW flavor of this algorithm prevents the occurrences of deadlocks across priority levels.

3.5.3. Suspensions and Concurrency Control

As discussed in Section 3.4.1, transactions may be temporarily suspended in a priority-based DBMS. Recall that suspended transactions wait outside the DBMS until sufficient buffer space becomes available to allow them to be reactivated. If a suspended transaction holds locks, it may block other transactions for the duration of its suspension. Thus, suspension may have negative performance consequences due to concurrency control conflicts, especially since the duration for which a transaction remains suspended depends on system load. In fact, if priority is not used for concurrency control, as in 2PL, suspensions can actually lead to deadlocks. To illustrate this problem, consider an extreme example where the buffer pool can accommodate just one transaction at a time. Assume that transaction S is running and that S has a write lock on data item d. A new transaction T of a higher priority than S arrives. To allow T to begin execution, S is suspended. If T now requests a write lock on d, a deadlock occurs between S and T; S waits to be readmitted to the system, while T waits for S to release its lock on d. In general, detection of deadlocks involving multiple resource types is more complicated than detecting deadlocks due to locking alone, as it involves searching for "knots" rather than cycles in waits-for graphs [Knap87].

In order to avoid these problems, we suggest that concurrency control algorithms be extended to include the notion of *suspended locks*. When a transaction S is suspended, the lock manager marks its locks as being such. Then, instead of blocking transactions when there is a conflict over a suspended lock, the lock manager restarts the suspended transaction. Note that if a transaction is blocking other transactions at the time it is suspended, it will be restarted immediately in this approach. An alternative approach would be to automatically restart a suspended transaction if it holds locks; this approach would be unduly pessimistic, however, leading to unnecessary restarts.

CHAPTER 4

MODEL AND METHODOLOGY

In this chapter, we describe a performance model of a priority-oriented database system. In subsequent chapters, this model will be used to study the tradeoffs involved in priority scheduling at each of the four resources in the system (the CPUs, the disks, the main memory buffers, and the data), and to examine the relative performance of alternative priority scheduling algorithms. After describing the model, we discuss the experimental methodology and metrics that will be common to all our experiments. Finally, we conclude this chapter with a description of the workloads that will be used to drive our simulations.

4.1. Modeling a Priority-Oriented Database System

Our performance model of a priority-oriented DBMS is an extension of the model of a single database site from the distributed database study described in [Care88]. It has been implemented using the DeNet simulation language [Livn88].

In our model, the DBMS has 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 the main memory resources of the system; and a *Concurrency Control Manager*, which implements the details of a particular concurrency control algorithm. Figure 4.1 presents the overall structure of this model and summarizes the key interactions between the components. While the model described in [Care88] contained these components, buffer management was not included in the Resource Manager in that study. In addition, each of the components has been modified to capture the priority scheduling algorithms described in Chapter 3. In this section, we first describe the model of the data. This is followed by a description of the remaining components, each of which is realized by a DeNet module, along with their parameters.

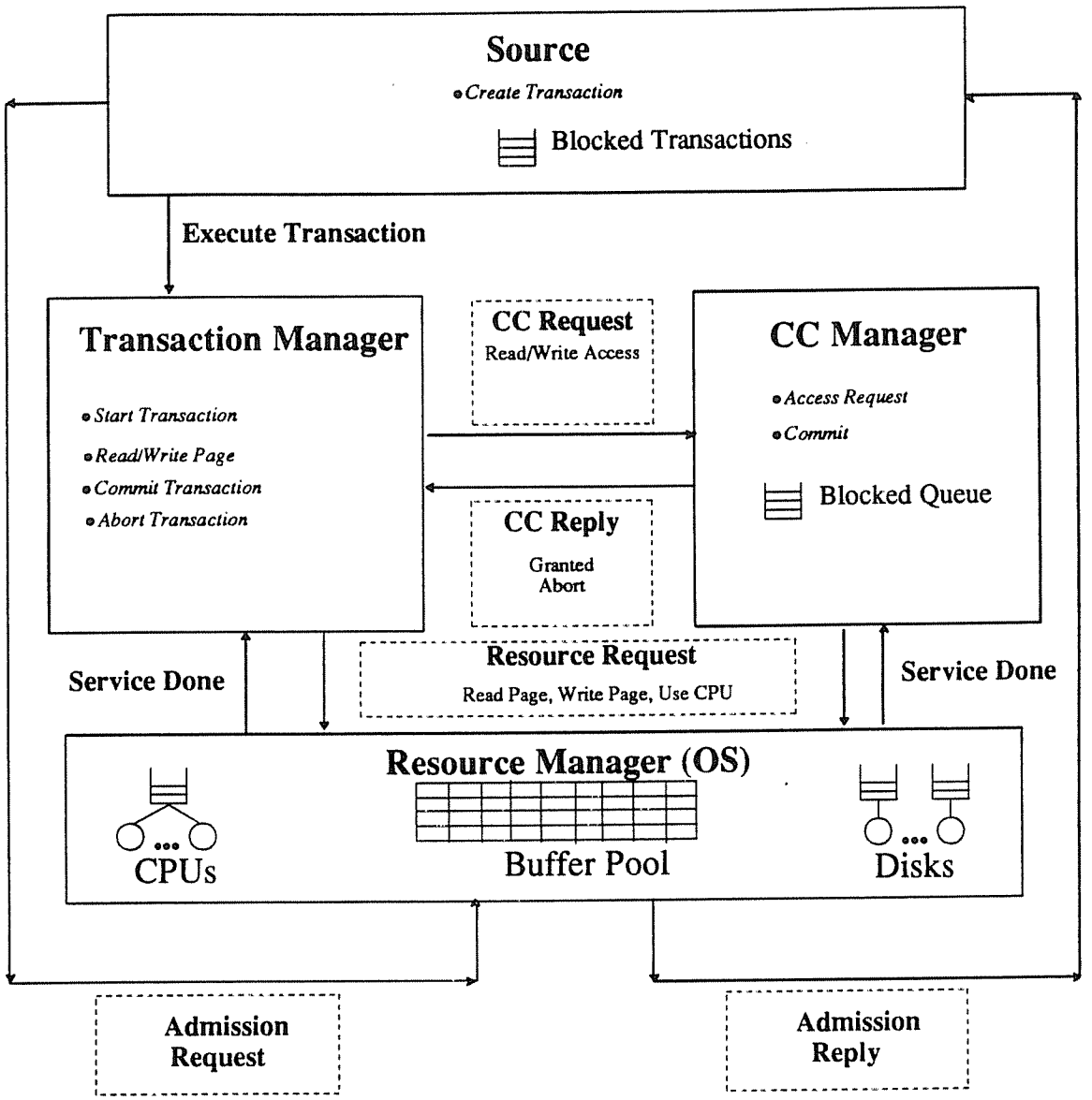

Figure 4.1: Components of the DBMS Model.

4.1.1. Modeling the Data

The database is modeled as a collection of *relations*. In turn, each relation is modeled simply as a collection of pages. As subsequent sections will show, page-level granularity was used consistently across all components of the DBMS model; for example, CPU and disk costs for processing the data were modeled on a per page basis, and page level locks

were used for concurrency control. Modeling the data at the tuple level would have increased the complexity of our simulations without adding significantly to our understanding of the issues related to priority scheduling.

In addition to relations, the database model assumes the presence of indices to help transactions access the relational data. An index may be structured either as a clustered or as a non-clustered B+ Tree, and each relation may have one or both types of indices. Table 4.1 summarizes the key parameters of the database model. The number of relations in the database is *NumRelations*. For each relation i ($1 \leq i \leq$ *NumRelations*), $RelSize_i$ is the relation size in pages, and $NumIndices_i$ indicates the number of indices available for the relation. If the relation has j indices, $IndexType_{ij}$ indicates whether each index is clustered or non-clustered, $Fanout_{ij}$ indicates the fanout of the internal index nodes, and $Levels_{ij}$ determines the number of levels of the B+ Tree.

4.1.2. The Source Module

The Source module is the component responsible for modeling the workload of the DBMS. Table 4.2 summarizes the key parameters of the workload model. A transaction may belong to any one of *NumClasses* classes, where classes can differ from one another in the specific relations accessed or the type of queries or updates involved. Each transaction may have any one of *NumPriorities* priority levels. The model is that of an open queueing 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 <class i, priority j> transaction is controlled by $TransType_{ij}$. Selections can be performed via sequential scans or index scans, and we model two basic join methods: nested-loops joins and classic hash joins. Since repetitive reaccess behavior can affect

Parameter	Meaning
NumRelations	Number of relations
$RelSize_i$	Number of pages in relation i
$NumIndices_i$	Number of indices on relation i
$IndexType_{ij}$	Types of index (clustered/non-clustered)
$Fanout_{ij}$	Fanout of internal nodes of indices
$Levels_{ij}$	Number of levels of index B+ Trees.

Table 4.1: Database Model Parameters.

buffer hit ratios significantly, the particular transaction types supported were chosen to provide both transactions that exhibit a repetitive pattern of data accesses, such as nested loop joins and hash joins, as well as transactions that do not reaccess any data.

For each transaction class i of a particular priority level j, an execution plan is provided in the form of a set of parameters. For selections, the relation, the access path and the mean selectivity are provided as parameters. Consistent with our page-level model, selectivity is also defined at page granularity; for example, a 1% selection of a 1000-page relation accesses 10 data pages. The actual selectivity for relation k is varied uniformly over the range [$Selectivity_{ijk}$/2, 3*$Selectivity_{ijk}$/2]. For select-updates, the probability of updating a page is additionally specified via the parameter $UpdateProb_{ijk}$. For select-joins, the join method and the inner and outer relations are provided in addition to the selectivity parameters. Finally, times spent at the CPU for processing or updating pages are uniformly distributed: the CPU time spent per data page varies uniformly over the range [$DataPageCPU_{ijk}$/2, 3*$DataPageCPU_{ijk}$/2], and similar distributions are used for $IndexPageCPU_{ijk}$ and $UpdateCPU_{ijk}$. Given an execution plan, the Source module generates a list of page accesses that models the sequence in which pages will be accessed by the transaction.

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 transactions
$TransType_{ij}$	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 kth relation
$Selectivity_{ijk}$	Fraction of kth relation selected
$UpdateProb_{ijk}$	Probability of page update
$IndexPageCPU_{ijk}$	CPU time for processing an index page
$DataPageCPU_{ijk}$	CPU time for processing a data page
$UpdateCPU_{ijk}$	CPU time for updating a data page

Table 4.2: Workload Model Parameters.

4.1.3. 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 first requests permission to access the page from the Concurrency Control Manager. Then, if the access is granted, 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. A request to update a given page is always preceded by a request to read the page. The Resource Manager also 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. The Transaction Manager is also informed by the Concurrency Control Manager when a transaction is aborted.

4.1.4. 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 to support the Priority-LRU, Priority-DBMIN, and Priority-Hints buffer management algorithms, respectively. In addition, versions supporting Global-LRU and Global-Hints, two non-priority-based buffer management algorithms, have also been implemented. The Global-Hints algorithm is introduced in Chapter 6. The parameters of the Resource Manager are summarized in Table 4.3.

4.1.4.1. CPU and Disk Models

The DBMS has *NumCPUs* CPUs and a single priority queue for recording outstanding CPU requests. The actual CPU for processing a given request 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; as advocated in Chapter 3, each transaction voluntarily gives up the CPU after processing or updating one page. There are *NumDisks* disks in the system, and requests at each disk are scheduled according to the prioritized elevator algorithm described in Chapter 3. We model the data as being uniformly distributed across all of the disks and across all tracks within a disk. The total time required to complete a disk access is computed by the following formula:

Disk Access Time = Seek Time + Rotational Latency + Transfer Time

The rotational latency and transfer time are together modeled as a single parameter called *DiskConst*. As in [Bitt88], the time required to seek across n tracks is computed using the formula:

$$\text{Seek Time}(n) = \text{SeekFactor} * \sqrt{n}$$

In addition, a key feature of the model is that priority scheduling can be switched "on" or "off" at both the CPUs and the disks. This allows the tradeoffs of priority scheduling at each resource to be studied in isolation and in combination with priority scheduling at other resources. For example, in order to examine the impact of disk priority scheduling alone on system performance, we can switch off priority scheduling at the CPUs and use non-priority-based algorithms for both buffer management and concurrency control. *CPUPrio* and *DiskPrio* are the parameters used to control priority at the CPUs and the disks, respectively.

4.1.4.2. 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*. In order to isolate the effects of admission control from those of buffer replacement, we also provide a parameter called *AdmPrio* at the buffer manager. *AdmPrio* is used to switch priority "on" or "off" in the admission control decision, just as *CPUPrio* and *DiskPrio* are used at the CPUs and the disks. The asynchronous write engine, which is common to all of the algorithms studied, is activated automatically after *EngineSleepTime* seconds. *FlushThreshold* is the threshold that it uses to choose "sufficiently old" dirty pages to be flushed to disk. As mentioned earlier, a separate buffer manager component has been implemented for each

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
CPUPrio	Switch to turn priority on/off at CPUs
DiskPrio	Switch to turn priority on/off at disks
NumBuffers	Number of page frames in buffer pool
AdmPrio	Switch to turn priority on/off at Admission Control
EngineSleepTime	Period after which write engine wakes up
FlushThreshold	Threshold used to choose pages that are "sufficiently old" to be written to disk.

Table 4.3: Parameters of the Resource Manager.

buffer management algorithm studied.

4.1.5. The Concurrency Control Manager Module

The Concurrency Control Manager is responsible for handling concurrency control requests, i.e., requests for read locks and write locks, made by the Transaction Manager. A parameter called *CCReqCPU* specifies the amount of CPU time required to process a read or write access request. Versions of the Concurrency Control component have been implemented for the Wound-Wait/Two-Phase Locking (WW/2PL) algorithm, for the Prioritized-Wound-Wait (PWW) algorithm, and for standard Two-Phase Locking (2PL) without priority.

4.2. Methodology and Metrics

Our performance analysis of a priority-based DBMS consists of three sets of experiments; each set has a separate chapter devoted to it. Chapter 5 focuses on the use of priority for CPU scheduling, disk scheduling, and admission control. Chapter 6 deals with priority-based buffer replacement and allocation. Finally, in Chapter 7, the role of priority in concurrency control is examined.

In each chapter, the following methodology is used: First, we identify the specific resources that form the focal point of the analysis and the main issues that we intend to address. Then, we describe the simulation parameters used for a "base" experiment; these parameters are selected to force the system to operate in a region where there is substantial contention for the resource being studied. The results of the base experiment are then presented and analyzed in detail. Subsequently, we present experiments where we vary some of the key parameters in order to demonstrate the sensitivity of the results of the base experiment to these variations. Note that the range of operation of interest to us is determined primarily by the *utilizations* of the resources being studied, rather than by the system configuration. From a simulation viewpoint, there are several different ways to make any given resource the critical resource in the system for a particular workload. For example, the CPUs can be made the bottleneck by making most of the data fit in memory, increasing the per-page CPU time requirement, increasing the speed of the disks, or increasing the number of disks in the system. In order to keep as many factors consistent across our experiments as possible, our policy is to change the system configuration (i.e., the numbers of CPUs and disks) in preference to the other parameters. Thus, in order to study a CPU-bound system, we configure the system with a number of disks and just one CPU so that the utilization of each disk is small compared to that

of the CPU.

For most experiments, we consider just two priority levels ("low" vs. "high" priority). This simplifies the analysis and the presentation of our results; multiple priority levels are considered in Section 5.5, where we examine the extent to which an increase in the number of priority levels affects the basic conclusions of our study. In the experiments with two priority levels, low priority transactions are assumed to be running in the background, and we investigate the performance impact of the arrival of foreground high priority transactions. The arrival rate of high priority transactions is thus varied while keeping the arrival rate of low priority transactions fixed. Results for transactions of each priority level are presented separately in each experiment.

As discussed earlier, we use an open queuing system to model the DBMS. Since the throughput in a stable open queuing system is equal to the arrival rate, a metric related to transaction response time is more appropriate in our study. Our primary performance metric will be the average *response time ratio* (RTR) for transactions at each priority level. We define the RTR of a transaction as the ratio of its actual response time to $E_{standalone}$, where $E_{standalone}$ is its estimated execution time on an unloaded system with an infinitely large buffer pool. A transaction's response time is computed by subtracting the time at which it was submitted to the DBMS from the time at which the transaction commits. The execution time of the transaction in an unloaded system is estimated by summing the CPU requirements associated with its page accesses and by assuming one I/O per distinct page that it references.[1] That is, only one I/O is assumed for a page, whether it is touched just once by a transaction or accessed repeatedly. Since the transaction will not have any concurrency control conflicts in an unloaded system, lock waiting time does not enter into the picture. The RTR of a transaction, then, reflects the effects of the presence of competing transactions and of the finite size of the buffer pool on the response time of the transaction. As the load on the system increases, contention for buffers causes an increase in the time spent waiting outside the system as well as increased I/O; contention for disks and CPUs causes increased disk and CPU waiting times; finally, contention for data causes increased time spent waiting for locks and may also lead to restarts. These factors all tend to increase the RTR of a transaction. On the other hand, if there is significant data sharing, the RTR of a transaction could be reduced because a portion of the transaction's read and write sets would already be in main memory. Note that, even when

[1]The cost of writing dirty data to disk is not included in this sum, since the asynchronous write engine operates independently of transactions. Also, each disk access is assumed to take 20 milliseconds in the computation of the standalone execution time.

the workload consists of a mix of transactions with different data access patterns and different sizes, the RTR provides a performance measure that is equally valid for all transactions (independent of the transaction mix). Each simulation in our experiments was run long enough so that, as long as the system remained stable for high priority transactions, the 90% confidence intervals of the RTRs of the high priority transactions were within 10% of the mean.

In each experiment, the RTRs for transactions of a given priority level p will be seen to grow dramatically when the load on the system increases beyond a certain threshold value. Of course, this threshold is a function of p as well as of other factors such as the scheduling algorithm being used. A small increase in the load in this region of operation causes the system to enter a state of saturation where it becomes unstable for transactions of priority p.[2] In addition to RTR, then, a second important metric of the performance of a priority-oriented DBMS is the range of loads over which the system remains stable for each priority level. Note that the system is likely to remain stable for high priority transactions long after the offered load has become high enough to make the system unstable for low priority transactions. This is because more and more of the system's resources are devoted to high priority transactions as the overall system load increases.

4.3. Workload Design

Since our main goal is to understand issues related to the use of priority scheduling in database systems, rather than details of query processing, we chose a simple workload design for our experiments. The workload consists of four types

Parameter	Setting
NumRelations	Varied
$RelSize_i$	1000 pages, 500 pages, 5 pages, 4 pages, 3 pages
$NumIndices_{ij}$	2 (1000-page & 500-page relations) 0 (smaller relations)
$IndexType_{ij}$	Both Clustered and Non-clustered (1000-page & 500-page relations)
$Fanout_{ij}$	200 (1000-page & 500-page relations)
$Levels_{ij}$	2 (1000-page & 500-page relations)

Table 4.4: Database Parameter Settings.

[2]The rise in RTR is so rapid here that some of the highest RTR values do not fit in our graphs; they are represented by interpolated points at the right end of each curve.

of transactions, each of which was chosen because it represents a characteristic page access behavior commonly found in relational systems. The first three transaction types, called "Looping," "Scanning," and "Random Reaccess (RR)" transactions, respectively, represent read-only queries; the fourth type, called "Update Transactions," include update operations. For each experiment, a subset of these transaction types were used to make up the workload, depending on the resources being studied. In order to avoid repetitive descriptions of the parameters of the different workload constituents, we describe each transaction type once here. The parameters that determine the details of the relations accessed by these transactions are presented in Table 4.4; the parameters for the transactions themselves are listed in Table 4.5.

4.3.1. Parameters Describing the Database

As discussed in Section 4.1.1, the database is modeled as a collection of relations, with each relation being modeled at the page level. Relations can have one of five sizes: 1000 pages, 500 pages, five pages, four pages, or three pages. The number of relations of each size varies from one experiment to another as we explore the effects of different levels of contention for buffers and for data. The larger relations, with 1000 pages and 500 pages, each had both a clustered index and a non-clustered index available; the smaller relations were not indexed. Each index consisted of a B+ Tree with two levels.

4.3.2. Looping Transactions

As their name indicates, looping transactions reaccess some of their pages sequentially a number of times. Each looping transaction is a select-join query, with the result of a clustered index selection on a 500-page outer relation being joined to a smaller inner relation. The selectivity of the outer relation varies uniformly between 0.5% and 1.5%. In all experiments, for each relation accessed by a transaction, the actual relation is chosen uniformly from among the set of all relations of that size. For the looping transactions, the outer relation is chosen uniformly from among the set of 500-page relations, and the inner relation is chosen uniformly from among the set of small relations (i.e., relations with three, four, or five pages). The average CPU cost for processing each accessed page is set at four milliseconds; the actual cost varies uniformly between two and six milliseconds. *PageAccesses* is the expected number of page reads and writes for the transaction, with repeated references being counted as one access each time. Thus, each looping transaction accesses 42 pages on

Parameter	Looping	Scanning	RR	Update
TransType	Select-join	Scan	Select-join	Scan
JoinMethod	Nested Loops	-	Classic Hash	-
$RelSize_1$ *(outer)*	500-page	1000-page	500-page	1000-page
$RelSize_2$ *(inner)*	3-5 pages	-	3-5 pages	-
$AccessPath_1$	Clustered Index Scan	Clustered Index Scan	Clustered Index Scan	Non-clustered Index Scan
$AccessPath_2$	Sequential Scan	-	Hash Lookup	-
$Selectivity_1$	1%	1%	0.2%	1%
UpdateProb	0	0	0	0-100%
IndexPageCPU	4 ms	4 ms	4 ms	4 ms
DataPageCPU	4 ms	4 ms	4 ms	4 ms
UpdateCPU	-	-	-	1 ms
Page Accesses	42	12	46	21
Locality Set Sizes (index, outer, inner)	1, 1, 3-5	1, 1, -	1, 1, 3-5	1, 1, -
Replacement Policies	MRU, MRU, MRU	MRU, MRU, -	MRU, MRU, MRU	MRU, MRU, -
Fixing Requirements	3	2	3	2

Table 4.5: Workload Parameter Settings.

the average; two to traverse the outer index, and then 20 pairs of (outer, inner) pages.[3] The locality set sizes and the optimum buffer replacement policies for Priority-DBMIN are also listed in Table 4.5, as are the buffer page fixing requirements (assuming that one page needs to be fixed simultaneously for each relation and index accessed). The locality set sizes are 1, 1, and 3-5 for the index, the outer relation, and the inner relation, respectively; the replacement policy is MRU for each locality set.

4.3.3. Scanning Transactions

Scanning transactions touch each page just once. Each such transaction consists of a clustered index scan on a 1000 page relation, and the selectivity is varied uniformly between 0.5% and 1.5% as before. The CPU costs are the same as for looping transactions, and an average of 12 pages are accessed per transaction. The per-transaction fixing requirement is two buffer pages.

---

[3]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 Chapter 3). This is why there are 20 (outer,inner) page pairs, with each of five outer pages looping over the set of four inner pages for the average looping transaction.

4.3.4. Random Reaccess (RR) Transactions

We use the classic hash join algorithm [Shap86] to model random reaccess behavior. The classic hash join 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 is built in the private workspace of the transaction; the hash table maps join attribute values into tuple identifiers. 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 algorithm 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). That is, unlike most hash-based join methods, it is not required that a large block of memory be fixed in the buffer pool for the duration of the join. 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, hash-join methods need to be modified to allow such replacement decisions.

Each RR transaction is a select-join query, with the result of a clustered index selection on a 500-page relation being joined with a smaller inner relation. The mean outer relation selectivity is 0.2%, and for this type of transaction, we assume that each page of the outer relation has 20 tuples; 20 probes of the inner relation are thus required on the average. (We assume that each tuple of the outer relation joins with exactly one tuple of the inner relation). *IndexPageCPU* and *DataPageCPU* are both set to four milliseconds. *PageAccesses* is 46; an average of four pages of the inner relation is read in while building the hash table, followed by two pages of outer relation index traversal, and finally by 20 pairs of (outer,inner) page accesses, with one outer page appearing in all 20 pairs. The fixing requirements and locality set sizes for RR transactions are the same as for looping transactions.

4.3.5. Update Transactions

In order to model random sequences of updates, both within a transaction and across different transactions, a non-clustered index is used to select the data in our update transactions. Each transaction selects an average of 1% of a 1000-page relation. It is assumed that non-indexed attributes of the selected data items are updated; thus, indices do not need to be updated. The mean number of pages accessed by each transaction is 21 (two index pages, one of which is reaccessed 10 times, plus 10 data pages). The fixing requirement of each transaction is two buffers. The probability of updating a page is varied from 0% to 100%. As before, the per-page costs of accessing the data are set at four milliseconds, while an

additional one millisecond is required for those pages that are updated.

CHAPTER 5

BASIC PRIORITY SCHEDULING

As stated in Chapter 1, the primary goal of priority scheduling in a database management system is to make the DBMS behave as much like a preemptive-resume server as possible. In this chapter, the model described in Chapter 4 is used to examine how close we can get to this goal via the use of priority scheduling.

We focus our attention here on the CPUs and the disks, and separately consider workloads where each of these is the bottleneck resource. These two resources were chosen for the initial phase of the study because they are the most basic physical resources in the system; that is, buffer management and concurrency control algorithms ultimately affect the performance of the system at least in part by influencing the utilizations of the disks and the CPUs. The goal of this initial study is to analyze the relative importance of priority scheduling at the CPUs, the disks, and also for admission control. In order to isolate the performance effects of CPU and disk scheduling, we use the same buffer management policy (Priority-LRU) for all of the experiments in this chapter; the impact of alternative approaches to priority scheduling for buffer replacement is examined in Chapter 6. Our objective here is to address the following question: How closely can a DBMS emulate a preemptive-resume server by using priority scheduling only at the bottleneck resource of the system? More specifically, what is the relationship between the use of priority in resource scheduling and the use of priority in admission control?

5.1. Base Experiment: CPU-Bound Workloads

5.1.1. Parameter Settings

The key parameters for our base experiment are listed in Table 5.1. The system configuration and the workload parameters are chosen so that the CPU becomes the bottleneck resource. Since the workload consists of read-only queries, update-related parameters are not listed in Table 5.1.

Parameter	Setting
NumRelations	50 (10 relations of each size)
NumCPUs	1
NumDisks	8
NumTracks	1000
DiskConst	15 ms
SeekFactor	0.6 ms
NumBuffers	50
CPUPrio	ON,OFF
DiskPrio	ON,OFF
AdmPrio	ON,OFF
TransType	100% Looping
Low Priority Load	3 transactions/second

Table 5.1: Parameter Settings for the Base Experiment.

The database is modeled as a collection of 50 relations, including ten relations of each of the five relation sizes listed in Table 4.4. There are eight disks in the system, but just one CPU, as we are interested in making the system strongly CPU-bound. Each disk has 1000 tracks, and the sum of the rotational latency and the transfer time per disk access is set at 15 milliseconds. The factor relating seek time to seek distance is set at 0.6 milliseconds, so the expected disk access time is between 15 and 30 milliseconds. There are 50 page frames in the buffer pool. The switches determining whether priorities are used for scheduling at the CPU, the disk, and for admission control are turned off and on as part of the base experiment. The simplest priority-based buffer replacement algorithm, Priority-LRU, is used throughout.

In the description of the results, we use the following notation: Those schedulers where priority is turned on are referred to by abbreviations; the letters A, C, and D refer to the use of priority for admission control, CPU scheduling, and disk scheduling, respectively. The resources where priority is not used are identified by the absence of the corresponding letter. For example, the label AC refers to an experiment where priority is turned on at admission control and at the CPU, but is turned off at the disk; the label ACD refers to an experiment where priority scheduling is used in all three places. Also, the phrases "High Priority" and "Low Priority" are abbreviated as "HP" and "LP" (respectively) in the graphs.

A read-only workload consisting of the looping transactions (nested-loops joins) described in Table 4.5 is used in this experiment. The low priority arrival rate was chosen to provide a moderate background load. In the absence of any high priority transactions, the CPU utilization was approximately 50% with a low priority arrival rate of three transactions/second, while the disk utilization was approximately 10%.

5.1.2. Discussion

The parameter settings for the base experiment ensure that the disk utilization does not exceed 40%, while the CPU becomes fully utilized as the load on the system increases. Consequently, the presence or absence of priority disk scheduling has no effect on performance, though we use priority scheduling at the disk throughout this experiment. Thus, the priority scheduling combinations studied here are those with and without CPU and admission control priority (i.e., ACD, AD, CD, and D).

Figure 5.1(a) shows the average RTRs for high priority transactions as a function of the high priority arrival rate; Figure 5.1(b) shows the corresponding results for low priority transactions. In addition to the high priority RTR curves, Figure 5.1(a) also includes a curve labeled HPO (for High Priority Only) that indicates what the high priority RTRs would be if no low priority transactions were present in the background. This curve will help us evaluate how successful we are at treating high priority transactions like a preemptive-resume server should; the better we do at approximating this curve, the closer we are to meeting this design goal. Despite the fact that the system is CPU-bound, Figures 5.1(a) and 5.1(b) show that the use of priority for admission control is extremely important. Without priority in admission control, the system saturates at an arrival rate of about three transactions per second for both high and low priority transactions. With priority, however, the system is stable for high priority transactions until the arrival rate reaches about six transactions per second, which is twice the load. These differences, as we will see, are due to the use of priority in the admission control decision.

Let us consider how priority affects a given resource as viewed by transactions of each priority level. Arriving low priority transactions see both high priority and other low priority transactions contending for each resource. This view is the same whether or not priority scheduling is employed at the resource, though the quality of service that low priority transactions receive is affected by the scheduling policy. This is consistent with Figure 5.1(b), as the point where the system becomes unstable for low priority transactions is affected only slightly by the presence or absence of priority scheduling. Of course, the RTRs of low priority transactions are affected adversely by the use of priority; as one would expect, low priority transactions perform best when priority is not used at the CPU or for admission control.

High priority transactions get the same view of a resource as low priority transactions if it is not scheduled using priority. This explains why, when the buffer manager does not use priority to control admission in Figure 5.1(a), the

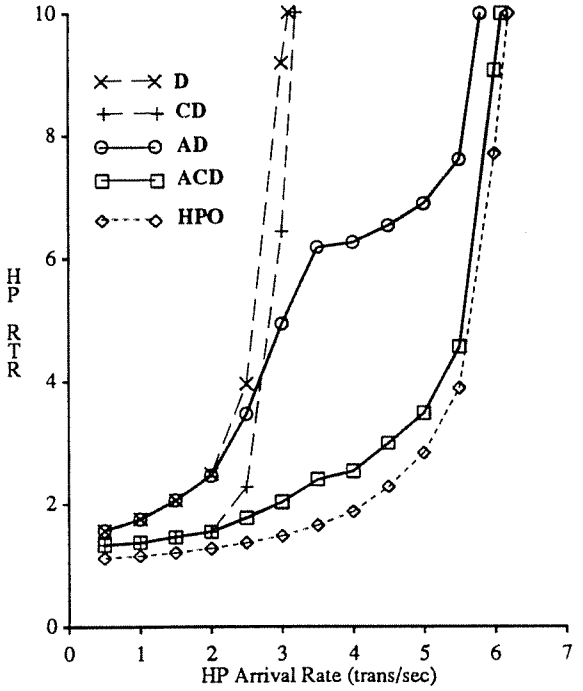

Figure 5.1(a): High Priority.

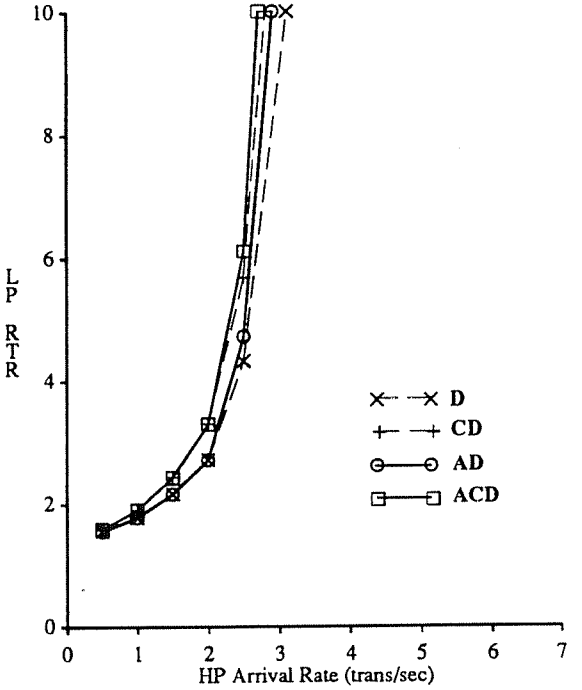

Figure 5.1(b): Low Priority.

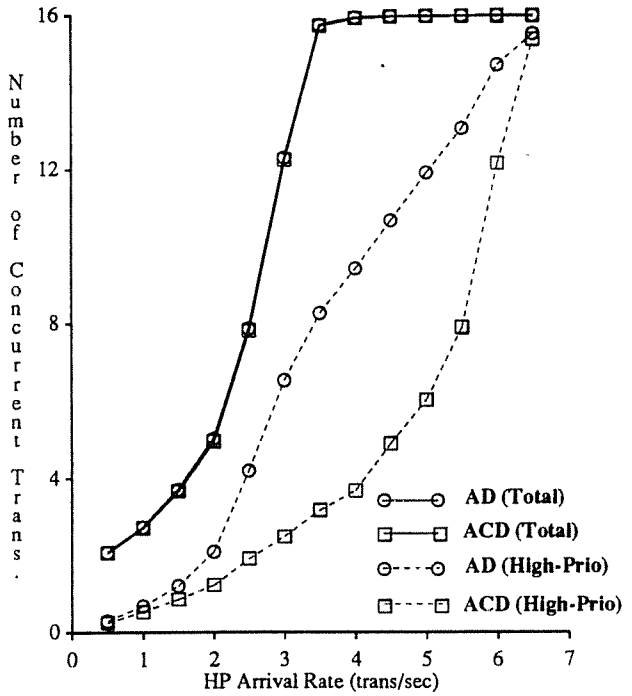

Figure 5.1(c): Number of Concurrent Transactions

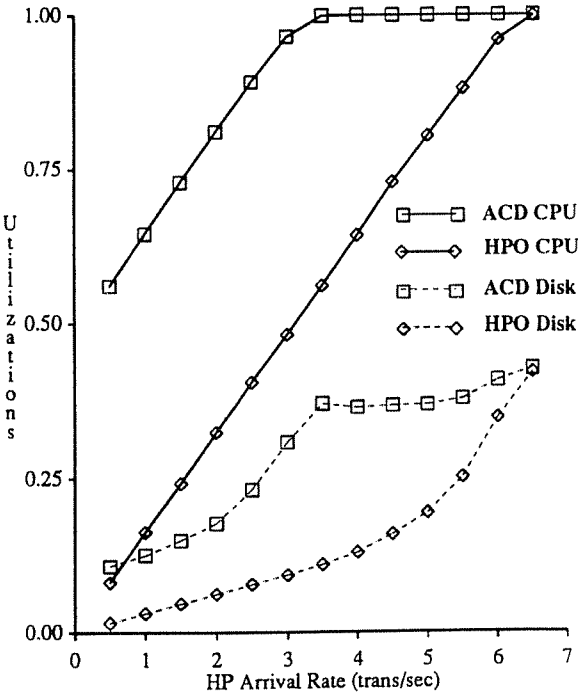

Figure 5.1(d): Resource Utilizations.

system saturates at three transactions per second for high priority transactions. In this case the "resource" is the DBMS itself, and if priority is not used for admission control, the external waiting time for high priority transactions is governed by the response time of low priority transactions. However, when the buffer manager's admission policy does favor high priority transactions, the DBMS admits them preferentially and even suspends low priority transactions in their favor. In this case, they see only other high priority transactions in the system, so they essentially see only a portion of the actual system load.

In order to better understand the relationship between the use of priority for admission control and the use of priority for CPU scheduling, Figure 5.1(c) shows the average total number of transactions and the average number of high priority transactions running in the system for the ACD and AD cases. Not surprisingly, Figure 5.1(c) indicates that the total number of transactions allowed to run concurrently is independent of the use of priority at the critical resource of the system. The distribution of this number between the different priority levels is affected by the use of priority for scheduling the CPU, however. When priority is not used at the CPU, high priority transactions have to wait to use that resource just as long as low priority transactions do; consequently, the response time of high priority transactions is higher than when the CPU is scheduled using priority. This explains the difference between the number of high priority transactions in the ACD and AD cases. Note that even after the system saturates for low priority transactions, a substantial number of these transactions remain in the system in both cases. Gradually, the low priority transactions are driven out of the system by new high priority arrivals, until finally almost all of the executing transactions are high priority transactions. Recall that the low priority arrival rate of three transactions per second accounts for approximately 50% CPU utilization, so the CPU becomes saturated entirely with high priority requests at around six high priority transactions per second. This is why the system remains stable up to an arrival rate of six transactions per second for the high priority transactions when priority is used for admission control.

The flattening of the RTR curve for the AD case in Figure 5.1(a) can also be explained with the help of Figure 5.1(c). Without priority at the CPU, the high priority RTR is similar to the low priority RTR until the system saturates for low priority transactions, i.e., until the total number of concurrent transactions reaches 16 in Figure 5.1(c). After this point, the RTR is fairly stable for a while as more and more low priority transactions are suspended due to high priority arrivals. That is, the buffer manager has already admitted as many transactions as it can, so all it can do in this range is trade low priority

transactions for high priority ones. Since the workload is CPU bound, the load seen by high priority transactions does not change in this range if CPU scheduling is not priority-based. The CPU then eventually becomes saturated with high priority transactions alone. However, when CPU scheduling is also priority-based, we see a different trend. In the ACD case in Figure 5.1(a), the high priority RTR behavior is very close to that of the HPO curve; recall that this curve shows the RTR behavior without low priority transactions.

In order to better appreciate the effects of priority scheduling, the system CPU and disk utilizations corresponding to the ACD and HPO curves are presented in Figure 5.1(d). The utilizations in the ACD curves represent the total load offered to the system, including both low and high priority transactions; the HPO curves reflect the high priority load alone. Figure 5.1(d) confirms that the utilization of the disks is relatively low here; as intended, the bottleneck resource is the CPU. Notice that there is a substantial difference between the utilizations of the CPU with and without low priority transactions; the use of priority scheduling "hides" this utilization difference from the high priority transactions, though, bringing the ACD RTR curve close to that of HPO in Figure 5.1(a). The non-preemptive scheduling used at the CPU (and at the disks) causes the small gap between the RTRs of high priority transactions with and without a background low priority load in Figure 5.1(a); as discussed in Chapter 3, however, non-preemptive scheduling is advisable in database systems, as preemptive scheduling can have serious negative side effects. Note that the effects of non-preemptability are largest in the middle of the range of operation studied here, when there are a substantial number of low priority transactions within the system. This is because the resource utilizations are low enough at low arrival rates that priority scheduling does not make much difference anyway; in contrast, almost all the transactions in the system are high priority transactions at high loads, so the non-preemptive use of the CPU by low priority transactions does not exert much influence.

The following conclusion can be reached from the results of the base experiment: priority is needed both in the buffer manager (for priority-based admissions control) and at the CPU in order to achieve the HPO performance objective for high priority transactions. If priority is not used for admission control, the system itself becomes the bottleneck resource, and the response time of high priority transactions is governed by transactions of lower priority that prevent them from entering the system.

5.2. Experiment 2: Varying Low Priority Load

In this experiment, we examine the sensitivity of the results of the base experiment to the level of the low priority background workload. All parameters except the low priority arrival rate are set exactly as in the base experiment.

In Figure 5.2, we present high priority RTRs for five different levels of low priority load with priority being used everywhere. These include a background load of zero, corresponding to the HPO curve of Figure 5.1(a), and a background load of three transactions per second, corresponding to the ACD curve of Figure 5.1(a). In addition, we present curves for fixed low priority loads of two and four transactions per second. The fifth curve, labeled "EQUAL", shows the RTRs of high priority transactions when the low priority arrival rate is kept equal to the high priority arrival rate at each point on the curve.

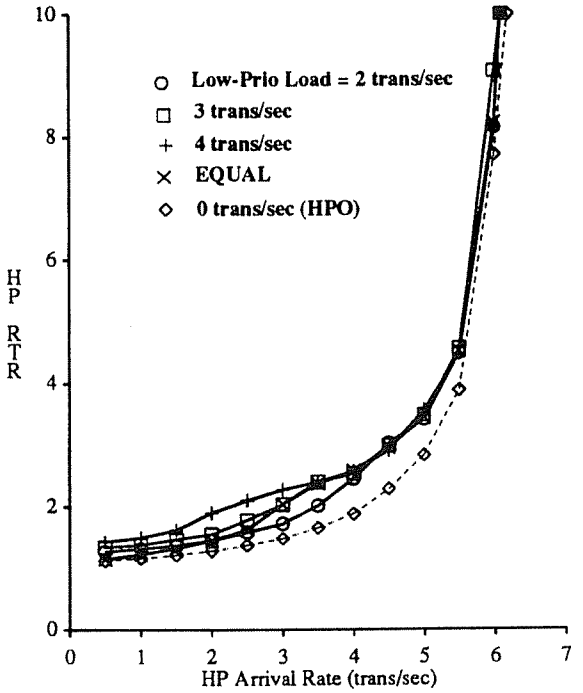

Figure 5.2: Varying Low Priority Load.

Figure 5.2 indicates that, as far as the point of saturation for high priority transactions is concerned, the load offered by low priority transactions is irrelevant. This conforms well to the preemptive-resume philosophy that we are trying to emulate by using priority scheduling. At low and moderate loads, however, when the number of high priority transactions in the system is small relative to the number of low priority transactions, high priority RTRs are affected somewhat by the level of the low priority load. As one would expect, RTRs for high priority transactions increase with the low priority load, as the number of instances where non-preemptive scheduling causes high priority transactions to wait for low priority transactions increases. For each low priority load, including the case where the low priority load was set equal to the high priority load, the general trends for the other priority switch combinations (AD, CD and D) were very similar to the trends observed in the base experiment; these results are not presented here since they do not contribute anything new to the discussion.

To summarize, then, Experiment 2 shows that the overall trends observed in the base experiment are not affected significantly by either a change in the fixed level of the low priority load or by a background load that varies with the foreground load. The system behaves much like a preemptive-resume server when priority is used for both admission control and for scheduling the bottleneck physical resource.

5.3. Experiment 3: Varying Disk Utilization

In the base experiment, the average disk utilization did not exceed 40% for the entire range of operation studied. In this experiment, we double the disk utilization by halving the number of disks in the system to four, while keeping all other parameters exactly the same as in the base experiment. Thus, the maximum disk utilization now reaches about 80%, causing increased contention for the disks; of course, the CPU remains the bottleneck resource.

In Figure 5.3, we present RTR curves for the priority switch combinations ACD, AD, CD, and D, as well as the HPO curve, for the new system configuration of four disks and one CPU. Qualitatively, Figure 5.3 resembles Figure 5.1(a) to a great extent; the major difference is that the gap between the HPO curve and the ACD curve is relatively larger here. This is because, with the system configured so that there is significant contention at both the CPU and the disks, non-preemptive scheduling at each resource begins to affect response time. Even though priority is used for disk scheduling, the increase in disk utilization increases the probability of a high priority disk request having to wait for an in-progress low priority disk request. Thus, not only is CPU waiting time a significant component of the response time of a high priority transaction, but

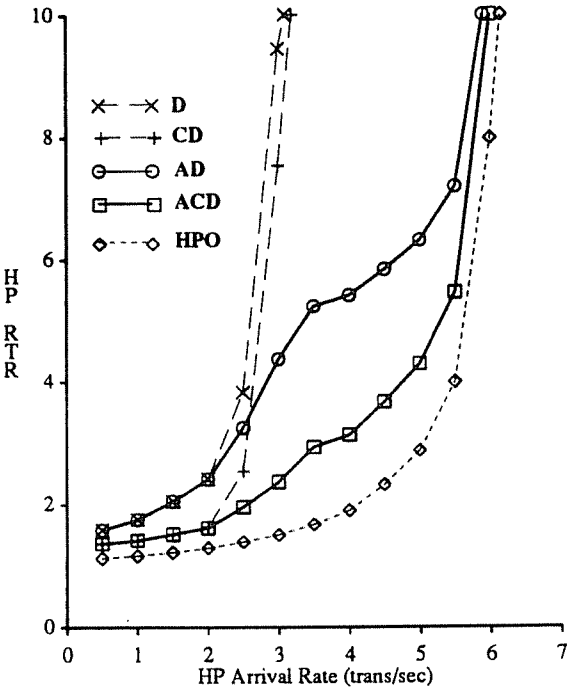

Figure 5.3: Varying Disk Utilization.

waiting time at the disk is significant as well. This experiment confirms the intuition that as the number of system resources with significant contention increases, the gap between the behavior of a priority DBMS and a preemptive-resume server increases; this is because the negative effects of non-preemptive scheduling at the different resources tend to accumulate.

5.4. Experiment 4: Disk-Bound Workloads

In this experiment, we focus on the impact and tradeoffs involved in priority scheduling at the disk. Instead of the select-join workload used in earlier experiments, we now use scanning transactions based on the clustered index selections described in Section 4.3.3. This change was made because we want to isolate the effects of priority-based disk scheduling from the effects of buffer replacement here, and clustered index selections are not as sensitive to buffer replacement decisions as are nested-loops joins. (The effects of priority-based buffer replacement decisions are studied in Chapter 6.) Furthermore, in order to make the workload strongly I/O-bound, the system is configured with four disks and four CPUs. The background low priority arrival rate is set at seven transactions per second, corresponding to a disk utilization of about

40 percent. The remaining simulation parameters are set exactly as in the base experiment. With this workload and configuration, the CPU utilization does not exceed 25% when the disks become 100%-utilized, so the presence or absence of priority-based CPU scheduling has no effect here. Thus, we examine priority scheduling at the disk and at admission control (i.e., the ACD, AC, CD, and C cases); priority scheduling at the CPUs is used throughout this experiment.

Figure 5.4(a) shows the RTRs for high priority transactions, and Figure 5.4(b) shows the associated results for low priority transactions. For the most part, Figures 5.4(a) displays the same trends that we saw in Figures 5.1(a), and for the same reasons, albeit with a different bottleneck resource. Again, it is evident that priority must be incorporated in the admission control decision as well as the bottleneck resource (the disks) in order to provide the desired level of performance for high priority transactions.

In order to clarify the tradeoffs of priority disk scheduling, the mean disk access times for high priority transactions and the overall average disk access times for all transactions for the ACD case are presented in Figure 5.4(c); also shown is the overall average disk access time for the AC case. (In the AC case, where priority is not used for disk scheduling, the average disk access time for high priority transactions was the same as the overall average disk access time.) In Figure 5.4(d), we present the *DiskWaitFrac* for the high priority transactions using the ACD and AC priority switch combinations. The DiskWaitFrac of a transaction is defined as the ratio of the total time spent by the transaction waiting for disk service to $E_{standalone}$, its estimated execution time in an unloaded system (the denominator of the RTR, as described in Section 4.2).

One important point to notice in Figure 5.4(b) is that low priority transactions suffer due to priority-based scheduling of the bottleneck resource to a greater extent here than in the base experiment. This is due to the fact that, in the range where they suffer, priority scheduling has reduced the service capacity of the disk by increasing the mean disk access time (as shown in Figure 5.4(c)). Combined with waiting times that are two to six times larger than those for high priority transactions, this produces earlier response time degradation for low priority transactions. In contrast, the high priority RTR is again close to that of the HPO curve when priority scheduling is used everywhere. This is in spite of the fact that, as shown in Figure 5.4(c), the mean disk access time for high priority transactions increases by 10 to 15 percent as a result of priority disk scheduling.[1] The increase in disk access time is more than offset by the decrease in disk waiting time that priority-

[1]The mean disk access time for high priority transactions exceeds the overall average disk access times in Figure 5.4(c) in the region of operation where the number of low priority transactions in the system exceeds that of high priority transactions. In this range, high priority disk requests occur less frequently than low priority requests, causing longer seeks on the average for high priority requests than for those at low priority.

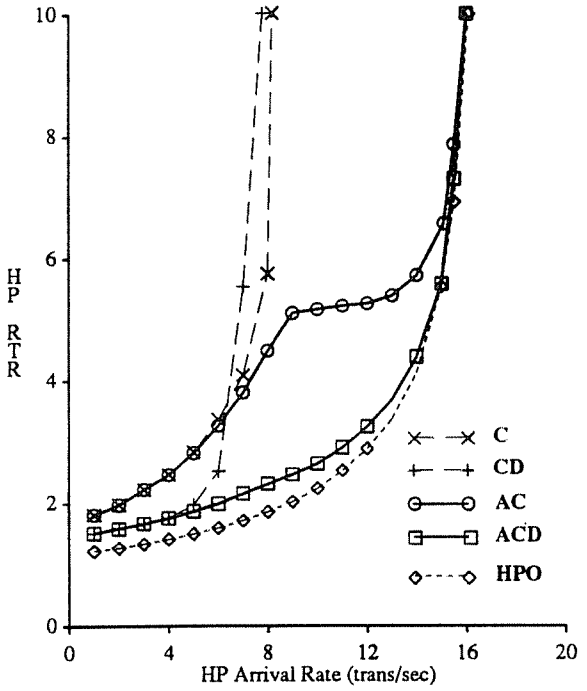

Figure 5.4(a): High Priority.

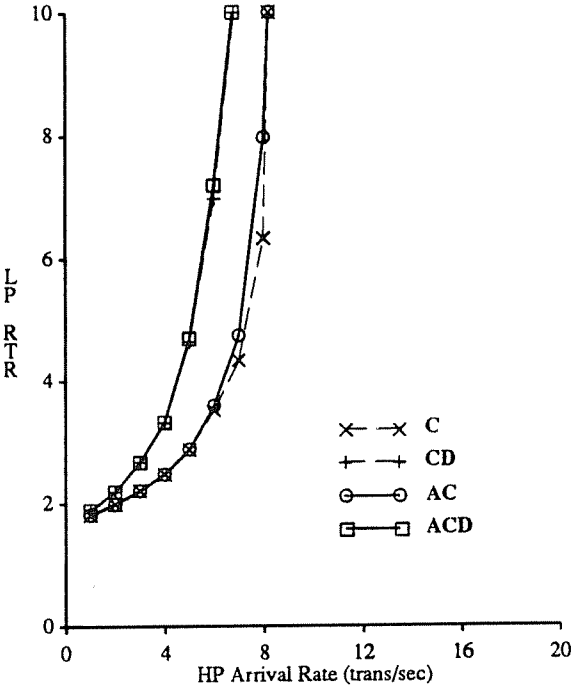

Figure 5.4(b): Low Priority.

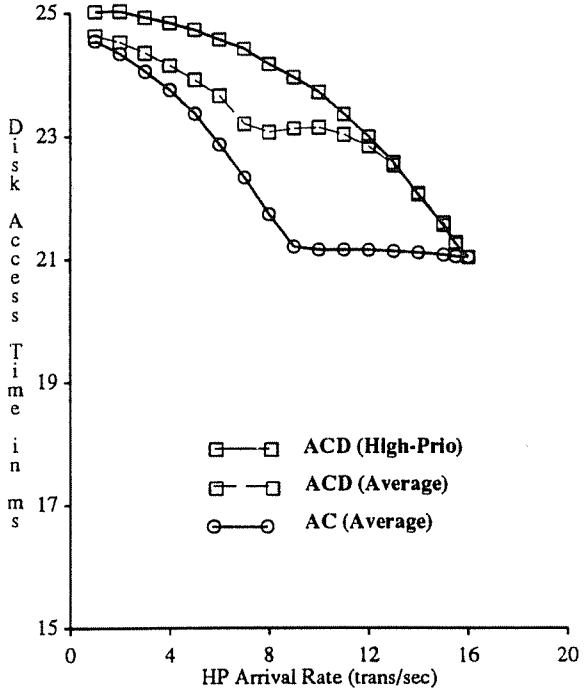

Figure 5.4(c): Disk Access Times.

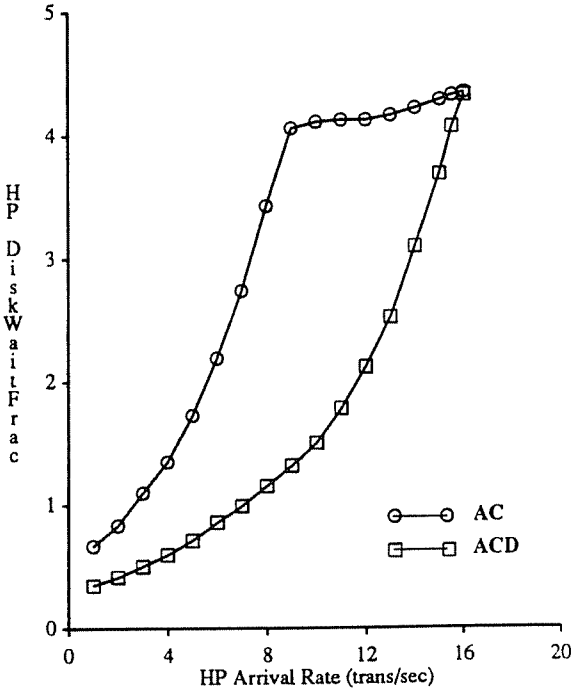

Figure 5.4(d): DiskWaitFrac.

based disk scheduling provides for the high priority transactions, as is clearly evident from Figure 5.4(d). The amount of disk waiting time for high priority transactions is up to 250% worse if priority is not used for disk scheduling, while the maximum penalty in disk access time due to priority does not exceed 20%. Thus, while there is definitely a price to be paid for doing disk scheduling based on priority, it is largely the low priority transactions that pay the price.

Another interesting observation can be made from Figure 5.4(c) as well. At low arrival rates, where the disk load is low, there is a relatively small penalty for priority-based disk scheduling. This is because there is little queuing for the disk in this region, so that disk requests are serviced essentially in FCFS order, independent of the use of priority. However, there is also no penalty at the highest arrival rates. The explanation is different in this case. Here, the reason that priority scheduling is able to do as well as the strict elevator algorithm is that it, too, effectively becomes the elevator algorithm; the disk is so heavily loaded due to high priority transactions that it is kept busy serving their elevator queue. The penalty is greatest in the middle range of arrival rates, where there is a substantial number of waiting low priority requests and relatively few high priority requests. Even in this range, the benefits of priority scheduling far outweigh the penalty for high priority transactions, as described above. Thus, priority-based disk scheduling appears to be very worthwhile for a priority-based DBMS.

5.5. Experiment 5: Four Priority Levels

In the last experiment of this chapter, we examine the performance of the proposed priority-based scheduling algorithms for a workload consisting of four levels of priority. In this experiment, transactions of each priority level have the same arrival rate, so one quarter of the offered load is at each of the four priority levels. The transactions that make up the workload used here are clustered index scans, with the same workload parameters as in Experiment 4, and all simulation parameters except the arrival rates remain unchanged from that experiment. Here we look at how transactions of the different priority levels perform when priority is employed everywhere (i.e, at the CPU scheduler, the disk schedulers, and for admission control).

Figure 5.5 shows the RTR results for the four priority levels. The observed trends are essentially what we would expect based on the results of our earlier experiments. First, the system succeeds at providing preemptive-resume-like performance, as the highest priority RTRs lie quite close to the HPO curve. Second, the saturation points for the different priority levels are not linearly distributed over the range of arrival rates. This is because of the system view that priority

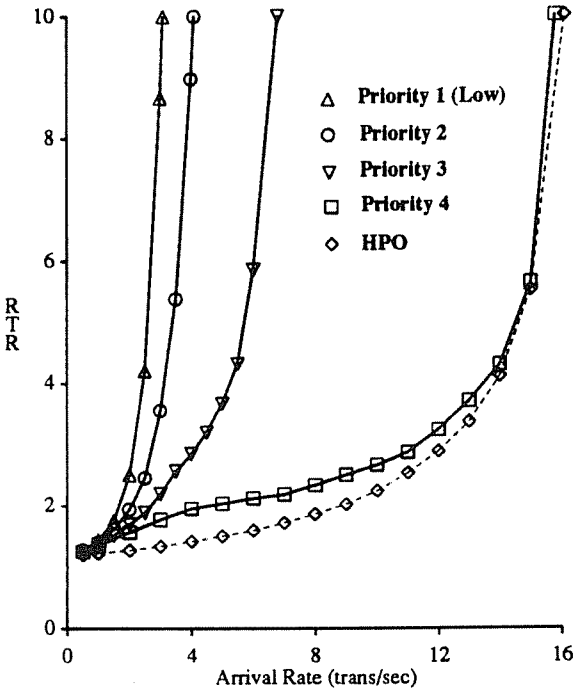

Figure 5.5: Four Priority Levels.

scheduling provides to the various priority levels: The highest priority transactions see only themselves (i.e., one-fourth of the load), transactions at the second-highest priority see themselves and the highest priority transactions (i.e., one-half of the load), the second-lowest priority transactions see the top three levels (i.e., three-fourths of the load), and the lowest priority transactions see the entire load as their competitors for system resources.

5.6. Conclusions

In this chapter, we examined the problem of priority scheduling at the CPUs, the disks, and for admission control in a database management system. Using preemptive resume as our model of desirable priority scheduling behavior, we found that it is indeed possible to do a good job of priority scheduling in a DBMS context. However, our simulation results indicate that the objectives of priority scheduling cannot be met by a single priority-based resource scheduler. Rather, independent of the physical resource that is the bottleneck, it is essential that priority scheduling on the critical resource be used in conjunction with priority-based admission control.

One other important point was brought out by the experiments in this chapter. Experiment 4 showed that priority scheduling is not free of cost: the use of priority at the disk results in an increase in the average seek time per disk access, thus reducing the system's effective disk capacity somewhat. As we have shown, however, this penalty in disk service time is more than compensated by a decrease in high priority disk waiting time, so that the use of priority is still justified.

CHAPTER 6

PRIORITY-BASED BUFFER MANAGEMENT

Having dealt with the CPUs and the disks, we now turn our attention to the next physical DBMS resource of interest: the main memory buffers. Several interesting 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. For example, 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 if the transaction needs to reaccess that page. The total load on the system may therefore increase purely as a consequence of the use of priority in buffer management, just as the effective disk load increases when priority is used for disk scheduling as shown in Section 5.4. 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. The three priority-based buffer management algorithms described in Chapter 3 (Priority-Hints, Priority-LRU, and Priority-DBMIN) assume different levels of information about transactions' data access patterns. The tradeoff between the level of information required by an algorithm and the resulting performance gains has not yet been explored in a priority context.

A third issue in priority-based buffer management is inter-transaction buffer 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 could thus be affected adversely by low priority updates. Another example of inter-transaction interference in the presence of priority could occur if high priority sequential scans quickly replace many 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 replacement in a DBMS that already uses priority at the CPUs, the disks, and for admission control, may itself be open to question. In this chapter, we present the results of experiments designed to investigate all of these issues.

In our buffer management experiments, we use each of the four transaction types from Section 4.3 to make up the workload. 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, though, we also present an experiment where the workload includes RR transactions. Finally, the effect of low priority updates on high priority transaction performance is studied using the non-clustered index scans discussed in Section 4.3.5. To allow us to focus on the tradeoffs of using different admission control and buffer replacement policies for priority-based buffer management, priority scheduling was used at all the resources in each of the experiments described here. That is, each of the three priority "switches" (*CPUPrio*, *DiskPrio*, and *AdmPrio*) remained *on* throughout all of the experiments in this chapter.

As stated in Chapter 3, 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 (50 buffers) in most experiments. 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[1], 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 by changing the database size.

6.1. Buffer Management Without Priority

We precede the description of the base experiment by running one experiment where 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-independent parameters for this

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

Parameter	Setting
NumRelations	50
$RelSize_i$	1000 pages, 500 pages, 5 pages, 4 pages, 3 pages (10 relations of each size)
$NumIndices_i$	2 (1000-page & 500-page relations) 0 (3- , 4- , and 5-page relations)
$IndexType_{ij}$	Clustered and Non-clustered (1000-page & 500-page relations)
$Fanout_{ij}$	200 (1000-page & 500-page relations)
$Levels_{ij}$	2 (1000-page & 500-page relations)
NumCPUs	4
NumDisks	4
NumTracks	1000
DiskConst	15 ms
SeekFactor	0.6 ms
NumBuffers	50
Foreground Workload Mix	50% looping, 50% scanning
Foreground Arrival Rate	0-10 transactions/second
Background Workload Mix	50% looping, 50% scanning
Background Arrival Rate	5 transactions/second

Table 6.1: Parameters for Buffer Management Without Priority.

experiment remain unchanged from Experiment 4 in Chapter 5; for easy reference, these parameters are listed again in Table 6.1. The workload in this experiment consists of a mix of 50% looping transactions and 50% scanning transactions; for this experiment, the background and foreground transactions are all run at the same priority level. In Figure 6.1, we show the RTRs of the foreground transactions when the Priority-LRU, Priority-DBMIN, and Priority-Hints algorithms are used.[2] 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 Chapter 3. 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.

[2]The RTRs of the background transactions are almost exactly the same as the RTRs of the foreground transactions, as one would expect, so they are not presented here.

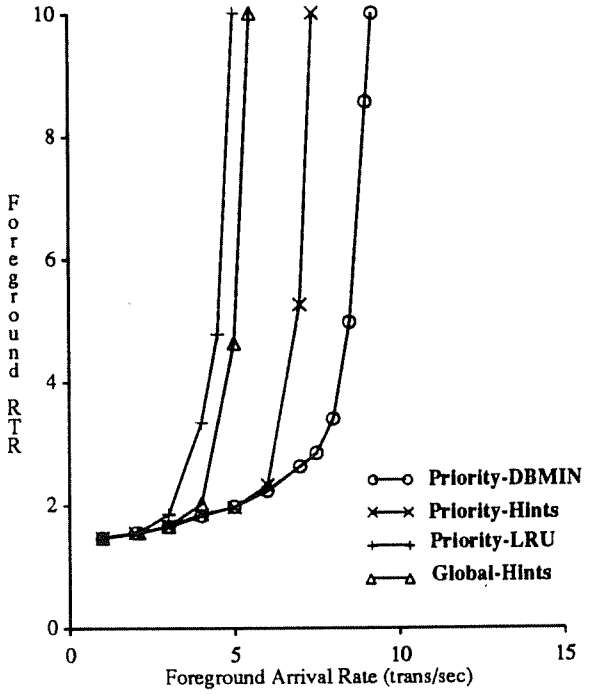

Figure 6.1: No Priority.

In Figure 6.1, 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 transactions. 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, once a looping transaction is able to obtain enough buffers to keep its loop (the inner relation for 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 five 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 [Chou85]. Secondly, Global-Hints frees normal 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 the workload considered. In addition, Priority-Hints matches the performance of Priority-DBMIN over a wide range of arrival rates. This experiment also isolated 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. Given this background, we can now investigate the performance of the system when the workload consists of transactions of different priority levels.

6.2. Base Experiment: Looping and Scanning Transactions

In this experiment, we study the impact of using priority in buffer management for the workload described in Table 6.1 with the foreground transactions running at high priority and the background transactions running at low priority. Figure 6.2(a) shows the RTRs for high priority transactions for the three priority-based buffer management algorithms (Priority-DBMIN, Priority-Hints, and Priority-LRU) and for two non-priority-based algorithms (Global-Hints and Global-LRU). RTRs for low priority transactions are shown in Figure 6.2(b).

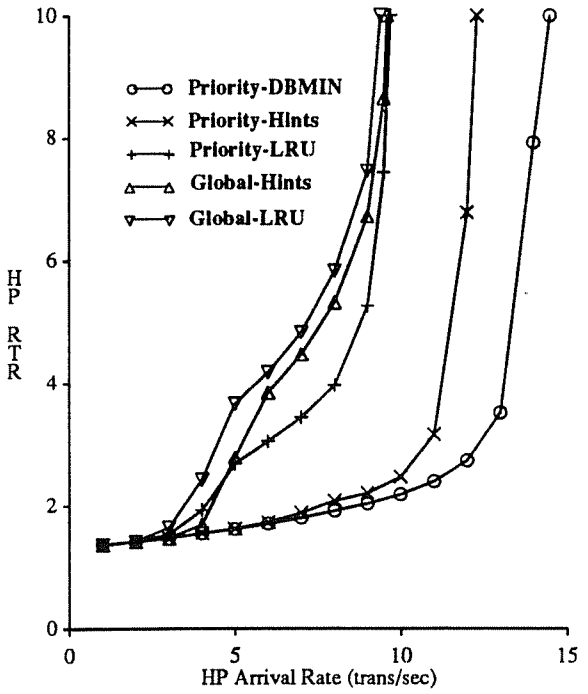

Figure 6.2(a): High Priority.

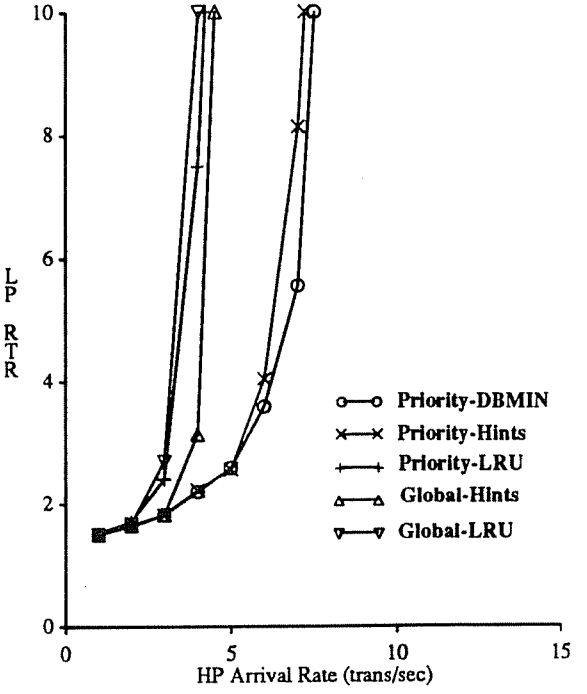

Figure 6.2(b): Low Priority.

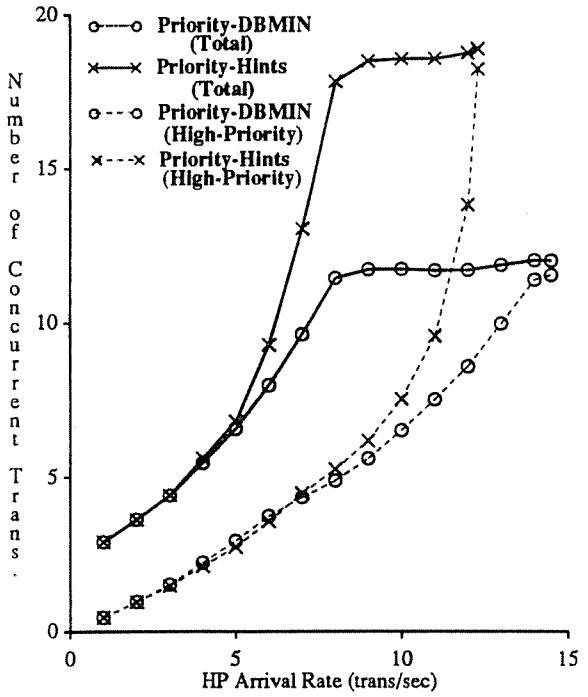

Figure 6.2(c): Number of Concurrent Transactions

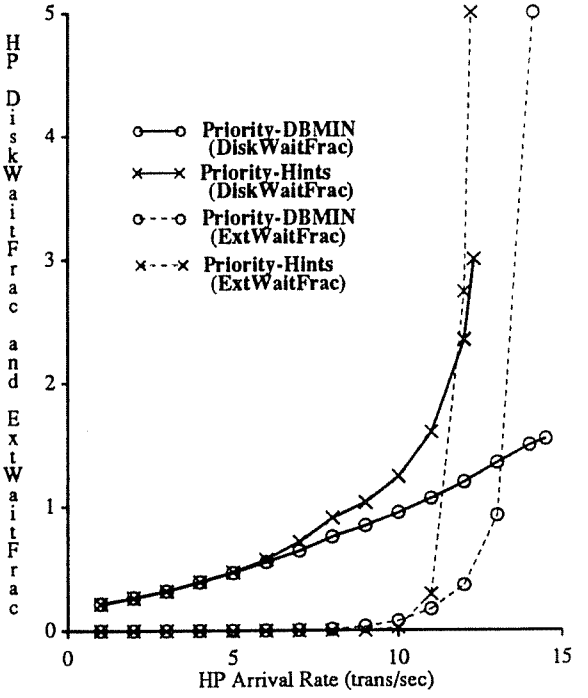

Figure 6.2(d): DiskWaitFrac and ExtWaitFrac.

Comparing the curves for each algorithm in Figure 6.2(a) with the corresponding curves in Figure 6.1, we see that, as expected, the use of priority results in improved performance for high priority transactions. For example, the system remains stable for a foreground arrival rate of up to 12 transactions/second in Figure 6.2(a) for Priority-Hints, while the system saturated at a foreground arrival rate of about 7.5 transactions/second in Figure 6.1 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 6.1 and 6.2(b). 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 6.2(a), we see that the behavior of Priority-Hints for high priority transactions is 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.[3] This tradeoff will be referred to again in the following experiments, where we will refer to it as the "conservative-liberal (C-L)" tradeoff. Figures 6.2(c) and 6.2(d) quantify this tradeoff. In Figure 6.2(c), we present the mean number of transactions (both total and high priority) that are allowed to run concurrently by the two algorithms. Figure 6.2(d) shows the mean DiskWaitFrac and the mean normalized *ExtWaitFrac* for high priority transactions for the two algorithms. Recall that DiskWaitFrac was defined in Section 5.4 as the ratio of the time spent by a transaction waiting for disk service to $E_{standalone}$, the expected execution time of the transaction in an unloaded system; similarly, ExtWaitFrac is the ratio of the time spent by a transaction waiting outside the system to $E_{standalone}$.

Figure 6.2(c) 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 6.2(d) shows the consequences of this. The DiskWaitFrac curves reflect the relative disk contention in the two algorithms, and Priority-DBMIN is the clear winner

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

in limiting disk contention; both the buffer miss ratio and the average disk waiting time per transaction are higher in Priority-Hints than in Priority-DBMIN. In contrast, Priority-Hints is somewhat more effective in limiting ExtWaitFrac up to an arrival rate of up to ten 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' ExtWaitFrac 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 6.2(c) 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 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 6.2(c) indicates 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 6.2(c) 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 6.2(a) shows that all of the algorithms provide similar levels of performance to high priority transactions at low loads. As the high priority load increases, however, 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 the worst of all; it does not distinguish between transactions of high and low priorities, and it uses the LRU criterion for page replacement. As for Global-Hints, it actually performs better than Priority-LRU for a small range of arrival rates (from approximately three to five transactions/second). In this range, Global-Hints' use of MRU and the fact that it frees normal 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 in this

region for the low priority transactions, so it must sometimes steal 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 effect of Global-Hints' use of MRU since more of the buffers are now owned by high priority transactions. The RTR of high priority transactions remains lower for Priority-LRU than for Global-Hints until the system becomes unstable for both algorithms.

In Figure 6.2(b), 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. 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. Rather, 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. Thus, the ExtWaitFrac for low priority transactions is significantly higher when Priority-DBMIN is used than when Priority-Hints is used, making the C-L tradeoff more "even" for low priority transactions than it is for high priority transactions. (Recall that high priority transactions cannot be suspended in either algorithm.) As long as there is sufficient disk capacity in the system to handle the increased low priority load in Priority-Hints, the two algorithms provide similar performance for low priority transactions.

Figure 6.2(b) also shows that there is little difference in the performance provided by the two 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. As a result, the globally least-recently-used buffer is 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.

Low priority transactions perform better under the Global-Hints algorithm than under the two LRU algorithms in Figure 6.2(b) 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 does not consider 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.

To summarize, the base experiment shows that the use of priority in buffer management is clearly beneficial (independent of the algorithm) if the response time of high priority transactions is the main criterion of system performance. However, the conclusions are less clear for low priority transactions: their performance may be worse under some priority-based buffer management algorithms (e.g., Priority-LRU) than for algorithms that do not consider priority when making replacement decisions (Global-Hints). The base experiment also shows that, for both high and low priority transactions, the performance provided by Priority-Hints is significantly better than the performance provided by Priority-LRU; Priority-Hints was found to perform almost as well as Priority-DBMIN for transactions of both priority levels. In subsequent experiments in this chapter, 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.

6.3. Experiment 2: Varying 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 6.1). This represents a low level of data sharing. We consider two higher 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 that in the base experiment; when there are just 5 relations, the level of data sharing is ten times that of the base experiment.

In Figure 6.3(a), we present the RTRs of high priority transactions for the three algorithms with 25 relations in the database. Figure 6.3(b) shows the RTRs of low priority transactions for the same database size. As one would expect, the performance of all three algorithms improves somewhat relative to that in the base experiment. The key difference between the trends shown in Figures 6.3(a) and 6.2(a) is the fact that Priority-Hints now performs almost exactly as well as Priority-DBMIN throughout the stable region of operation. This is because the C-L tradeoff is almost even for high priority transactions in this experiment; the increased time spent by high priority transactions blocked outside the system is almost exactly counterbalanced by the increased disk waiting times for high priority transactions in Priority-Hints, where a larger number of transactions is allowed to run concurrently. For low priority transactions, the tradeoff begins to favor Priority-Hints for this level of data sharing; Figure 6.3(b) shows that Priority-Hints consistently performs as well as or better than Priority-DBMIN for low priority transactions in a range where the performance of the two algorithms for high priority transactions is nearly identical.

Figure 6.3(c) presents the RTRs for high priority transactions in the case where there are just five relations in the database. In this case, 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 actually 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 the inner relation; a large fraction of the page requests made to the buffer manager are for one of these other 1500 pages. 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. In Priority-Hints and Priority-DBMIN, in contrast, the inner relations pages will remain in memory as long as there are some high priority transactions that need them. Also, these two algorithms both free normal 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. This is why the curve for Priority-LRU diverges from the other two in Figure 6.3(c).

In brief, this experiment shows that the performance of Priority-Hints improves to a greater extent than the performance of Priority-DBMIN as the relative size of the buffer pool increases. This is a consequence of Priority-DBMIN's

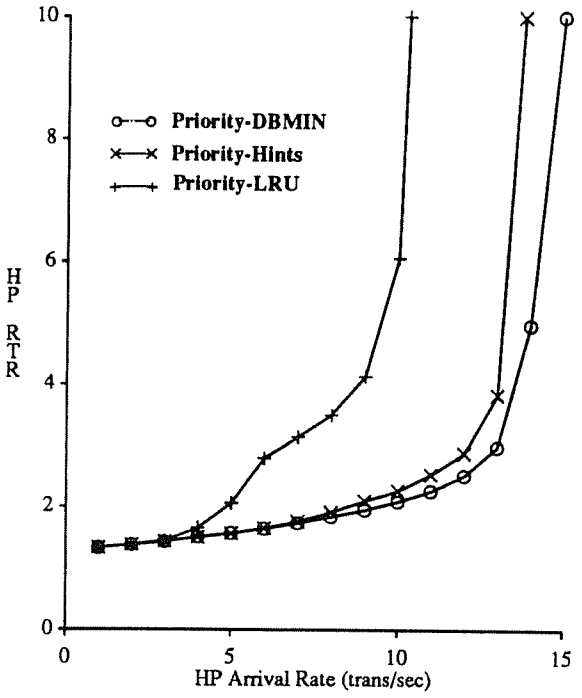

Figure 6.3(a): High Priority (25 Relations).

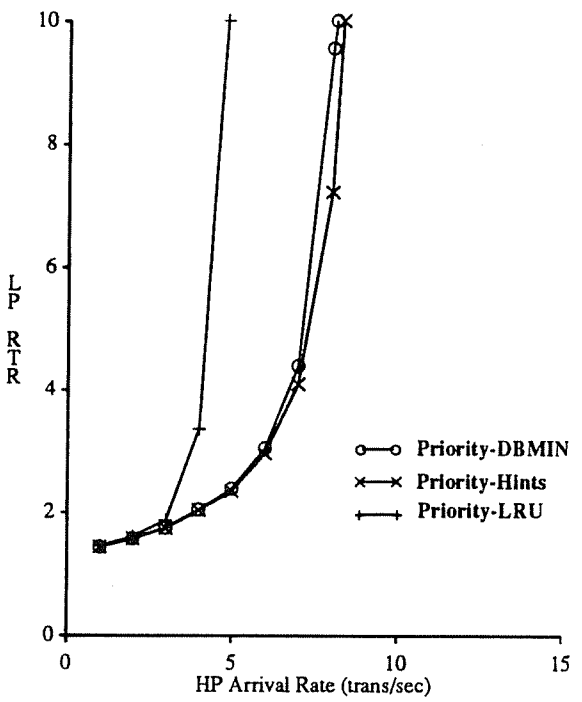

Figure 6.3(b): Low Priority (25 Relations).

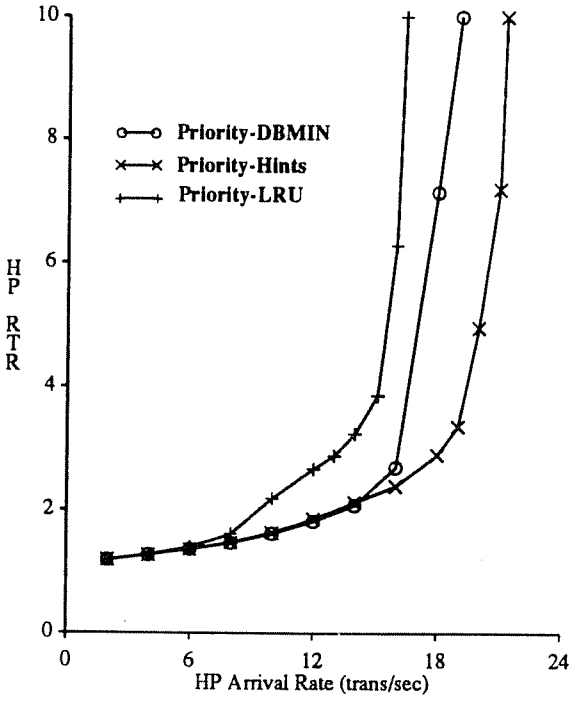

Figure 6.3(c): High Priority (5 Relations).

conservative admission control policy. Also, both Priority-Hints and Priority-DBMIN provided better performance here than Priority-LRU, even under very high data sharing.

6.4. Experiment 3: Varying 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 mix of transactions in the system while keeping the workload-independent parameters as in Table 6.1.

Figure 6.4(a) shows the RTRs for high priority transactions when the high priority workload consists entirely of scanning transactions; the low priority workload still consists of a mix of 50% looping and 50% scanning transactions. RTRs for the low priority transactions are shown in Figure 6.4(b). In Figure 6.4(a), the curves for the three algorithms coincide almost exactly. This is not unexpected, as the high priority workload is insensitive to buffer replacement and the admission control criteria for the three algorithms coincide for scanning transactions.[4] The interesting feature of this experiment is the relative performance of low priority transactions. As shown in Figure 6.4(b), Priority-Hints performs far better than Priority-LRU here. The reason is that, unlike Priority-Hints, Priority-LRU does not free scanning pages as soon as they are unfixed. High priority transactions thus steal 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 6.4(c) and 6.4(d) show the RTRs for high priority transactions and low priority transactions, respectively, when the high priority workload consists entirely of looping transactions; the low priority workload is still the same as before. There is now a relatively larger difference between the performance of Priority-Hints and Priority-DBMIN for high priority transactions than in the base experiment. The C-L tradeoff for high priority transactions favors Priority-DBMIN in this experiment, as 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.

---

[4]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 4.5.

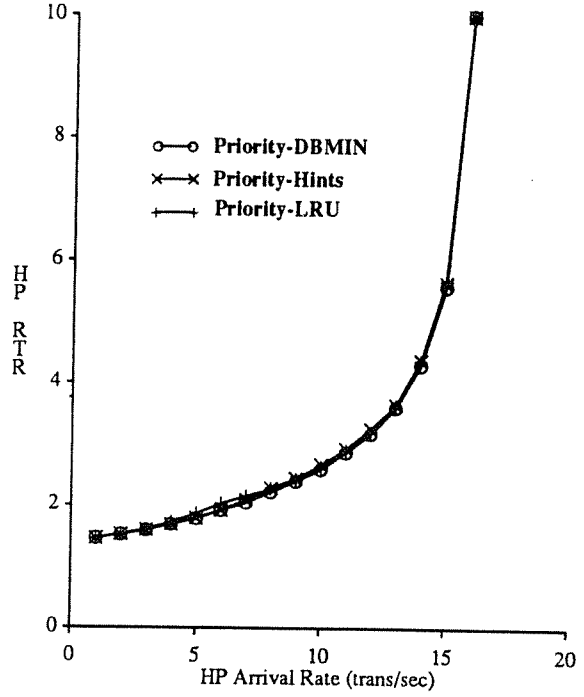

Figure 6.4(a): High Priority (HP Load = 100% Scanning).

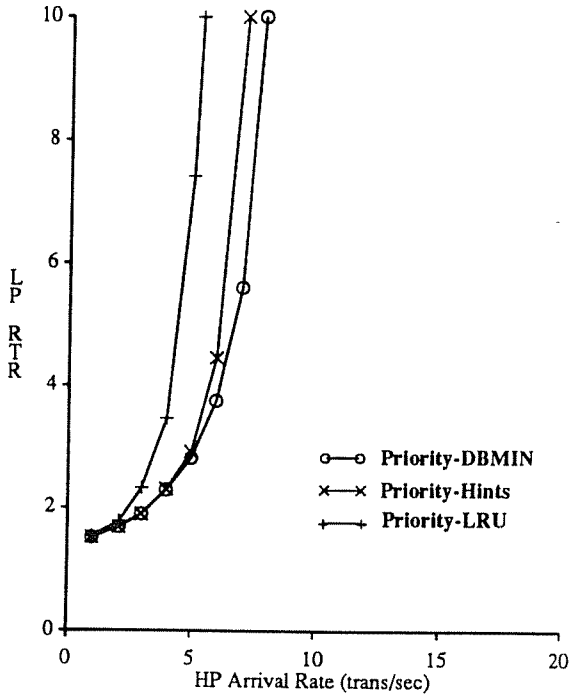

Figure 6.4(b): Low Priority (HP Load = 100% Scanning).

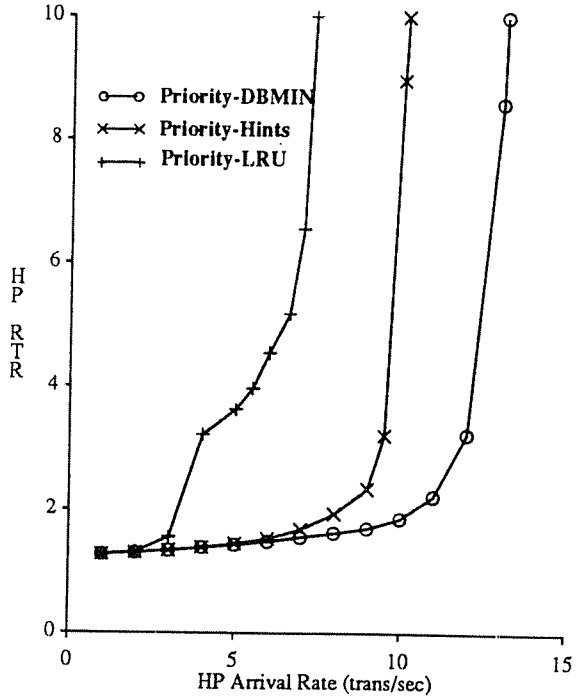

Figure 6.4(c): High Priority (HP Load = 100% Looping).

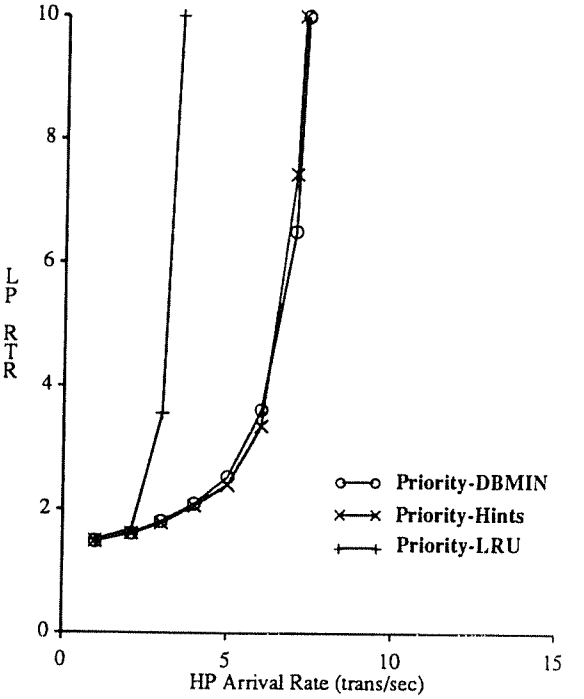

Figure 6.4(d): Low Priority (HP Load = 100% Looping).

In the experiments described thus far, the workloads consisted entirely of looping and scanning transactions. We now examine the performance of the three algorithms when the workload includes random reaccess (RR) transactions as well (see Section 4.3.4 for a description of our model of RR transactions). Note that the inner relation pages for the classic hash joins of the RR workload will be favored by the Priority-Hints algorithm. Figure 6.4(e) shows the RTRs for high priority transactions when the workload at each priority level consists of an equal mix of scanning and RR transactions; Figure 6.4(f) shows the corresponding low priority RTRs. The use of MRU provides no advantage here, as transactions either randomly reaccess their pages or scan through each page just once; despite this fact, 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 "normal" 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 loads. For low priority transactions, however, the performance of Priority-Hints and Priority-DBMIN is very similar for the same reasons as in the base experiment.

Finally, we also consider a case where the workload at each priority level consists of an equal mixture of each transaction type; i.e, 33% of the workload consists of looping transactions, 33% of scanning transactions, and the remaining fraction consists of RR transactions. Figure 6.4(g) shows the high priority RTRs for this workload, while the corresponding low priority RTRs are shown in Figure 6.4(h). Once again, the same overall trends in the relative performance of the three algorithms are evident; Priority-Hints does significantly better than Priority-LRU, but not as well as Priority-DBMIN.

From Experiment 3, we learn that Priority-Hints tends to perform better than Priority-LRU, independent of the proportion of looping, scanning, or random reaccess transactions in the workload. For high priority transactions, Priority-Hints provides significantly better performance than Priority-LRU unless the workload is insensitive to buffer replacement policies. Even when the high priority workload is insensitive to buffer replacement, though, Priority-Hints provides much better performance than Priority-LRU for low priority transactions if the low priority workload includes looping or RR transactions. As the proportion of looping or RR transactions in the high priority workload increases, Priority-DBMIN's admission control policy allows it perform better than Priority-Hints for high priority transactions, but the two algorithms provide very similar support for low priority transactions.

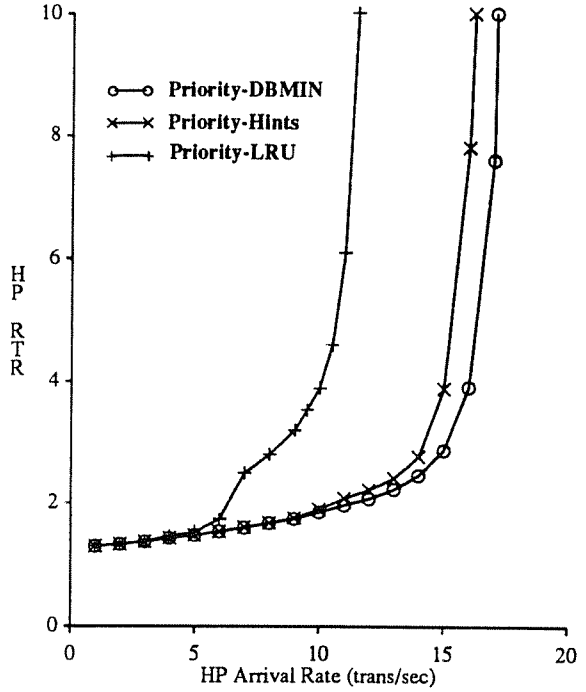

Figure 6.4(e): High Priority (50% Scanning, 50% RR).

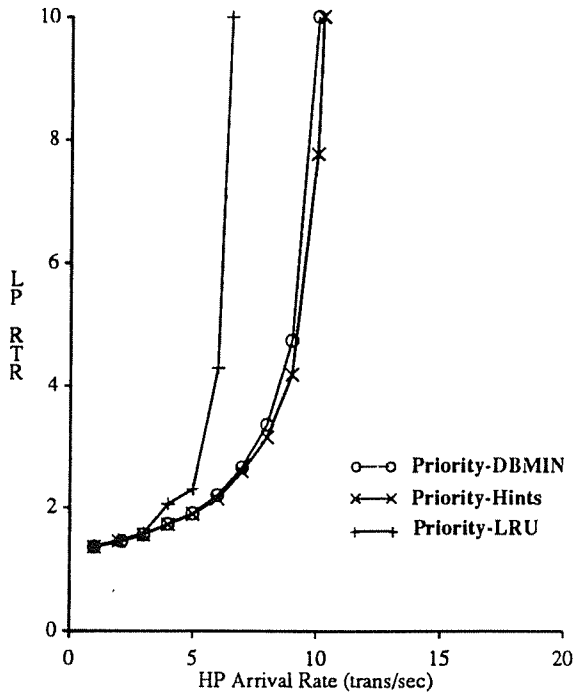

Figure 6.4(f): Low Priority (50% Scanning, 50% RR).

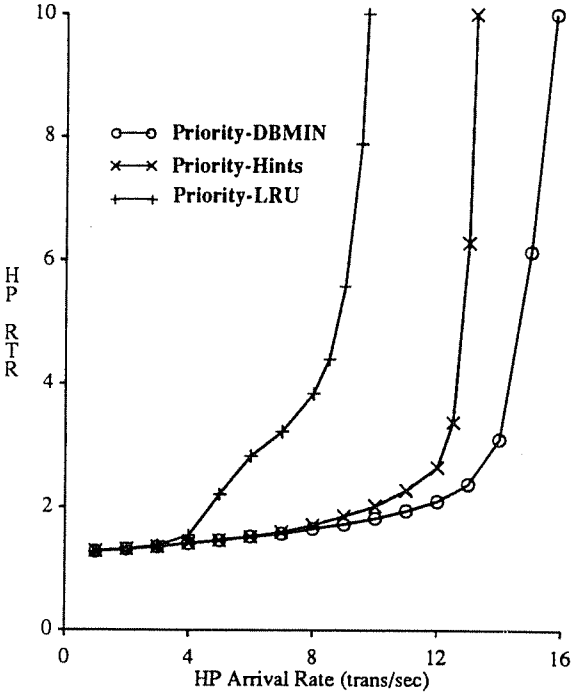

Figure 6.4(g): High Priority
(33% Scanning, 33% Looping, 34% RR.)

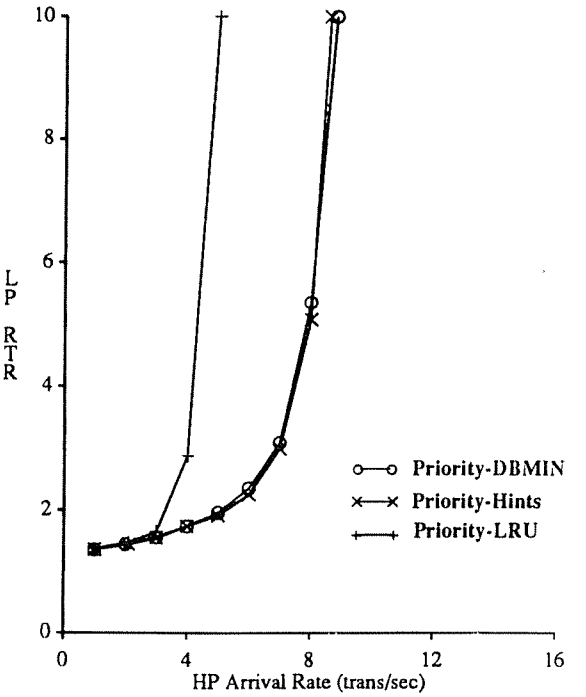

Figure 6.4(h): Low Priority.
(33% Scanning, 33% Looping, 34% RR.)

6.5. Experiment 4: Low-Priority Updates

In all of the experiments described thus far, the workload has 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. Recall also that, in our model, the disk scheduler assigns asynchronous disk writes a higher priority than that of any read request. As discussed earlier, the key update-related 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.

The workload-independent parameters for this experiment remain unchanged from those listed in Table 6.1. The high priority workload for this experiment consists of 100% looping, read-only transactions. The low priority workload consists of 100% scanning, update transactions; each update transaction consists of a non-clustered index scan, as described in Section 4.3.5. Low priority transactions arrive at a rate of five transactions/second, as before. The probability that a page is updated is varied from 0% to 100% in steps of 20%. *FlushThreshold* is set to zero in order to study the worst-case impact of low priority updates on high priority performance. The results presented here are not affected by the *EngineSleepTime* parameter; in fact, the asynchronous write engine is activated at every buffer miss. Other parameters are the same as in the base experiment of Section 6.2.[5]

Figure 6.5 shows the high priority RTRs for the three buffer management algorithms with a high priority arrival rate set at two transactions/second as the low priority update probability is increased from 0% to 100%. In order to isolate the impact of priority, we also present the corresponding curves for the case when all transactions have the same priority. From the curves labeled "1 Priority" in Figure 6.5, we see that when all transactions have the same priority, looping transactions suffer severely under Priority-LRU when the update probability is increased. Note that in this experiment, the pages that are updated are normal. Recall also that all three algorithms ignore dirty pages as far as possible in their search for replacement victims. Priority-Hints and Priority-DBMIN place dirty normal pages in the dirty list as soon as they are unfixed, just as they place clean normal pages in the free list at unfix time. In contrast, Priority-LRU does not free pages

[5]In this experiment, we ignore concurrency control conflicts; i.e., locks are granted immediately on request. Priority-based concurrency control will be the subject of Chapter 7.

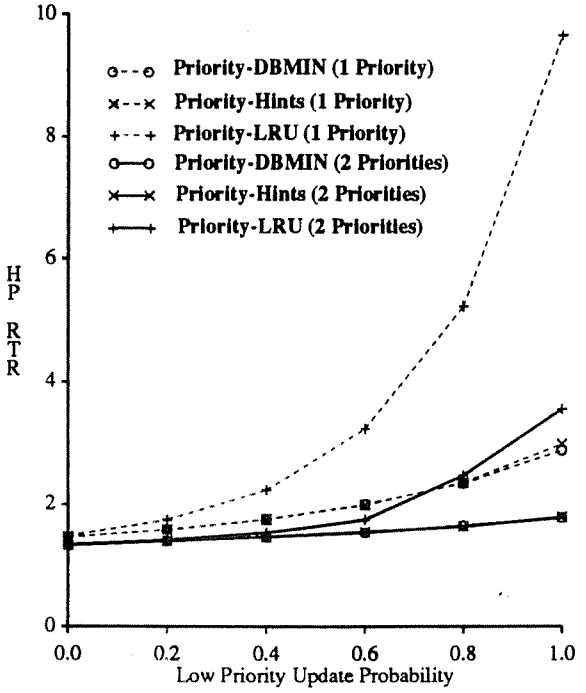

Figure 6.5: High Priority.
(Low Priority Updates.)

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 buffer deadlock as explained in Chapter 3. This causes dirty normal pages to accumulate in the buffer pool in Priority-LRU, which is why Priority-LRU performs much worse than Priority-Hints and Priority-DBMIN in the absence of priority. Priority-Hints performs just as well as Priority-DBMIN here.

When priority is introduced, all three algorithms provide improved performance for high priority transactions. However, Priority-LRU's tendency to delay the flushing of low priority updates causes the RTRs of high priority transactions to increase significantly as the update probability increases. In contrast, 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 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.

6.6. Experiment 5: Varying System Resource Capacities

In the buffer-related experiments described thus far, there have been 50 buffers, four CPUs and four disks in the system. In this experiment, we vary the number of main memory buffers, the number of CPUs, and the number of disks in the system in order to understand the impact of different resource capacities on the relative performance provided by the three buffer management algorithms. In Figure 6.6(a), we present the RTRs for high priority transactions when there are 100 frames in the DBMS buffer pool; all other parameters are set as in the base experiment. In Figure 6.6(b), the only change from the base experiment is that the number of CPUs has been decreased from four to one. Finally, in Figure 6.6(c), we present the high priority RTRs when the DBMS has just two disks and one CPU.

A comparison of Figure 6.6(a) with Figure 6.2(a) indicates that, although the RTRs for all three algorithms improve as a result of increasing the buffer pool size, the relative performance provided by the algorithms does not change significantly with the increase in memory size. In Figures 6.6(b), and 6.6(c), we see that Priority-Hints and Priority-DBMIN again provide similar levels of performance and that both these algorithms perform better than Priority-LRU. Even in Figure 6.6(b), where the CPU becomes the bottleneck resource, Priority-LRU performs significantly worse than the other two algorithms. In Figure 6.6(c), there is a slightly wider gap between the curves for Priority-Hints and Priority-DBMIN because the workload is disk-bound again in this case. Figures 6.6(b) and 6.6(c) confirm that the performance differences between Priority-DBMIN and Priority-Hints occur primarily as a consequence of higher disk activity when the latter policy is used; if the workload is CPU-bound, these two algorithms will provide comparable performance.

6.7. Conclusions

In this chapter, it has been demonstrated that buffer management can have a 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 buffering requirements of the offered load exceed the system's buffer capacity. The Priority-Hints algorithm, which uses page-level information provided by the database access methods, was shown to perform better than Priority-LRU for all the workloads considered. For most workloads, Priority-Hints performed almost as well as Priority-DBMIN; for workloads with high data sharing, Priority-Hints actually provided better performance than Priority-DBMIN. These results are significant because most existing database management systems use LRU-based buffer management algo-

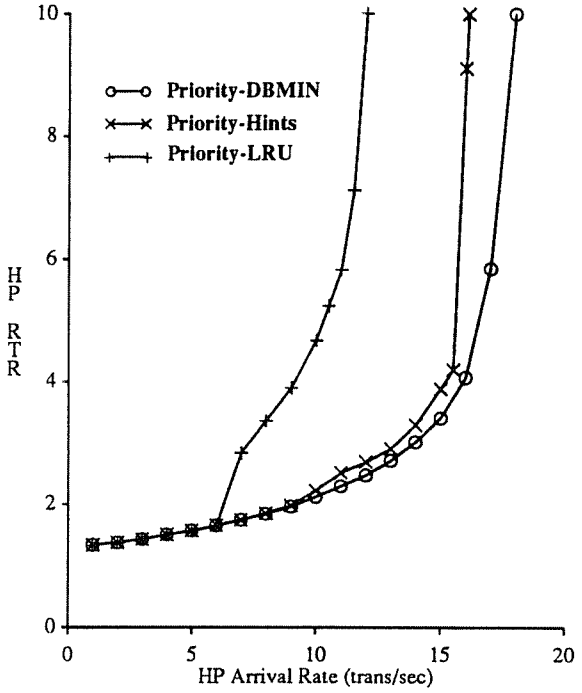

Figure 6.6(a): High Priority.
(100 buffers, 4 CPUs, 4 disks.)

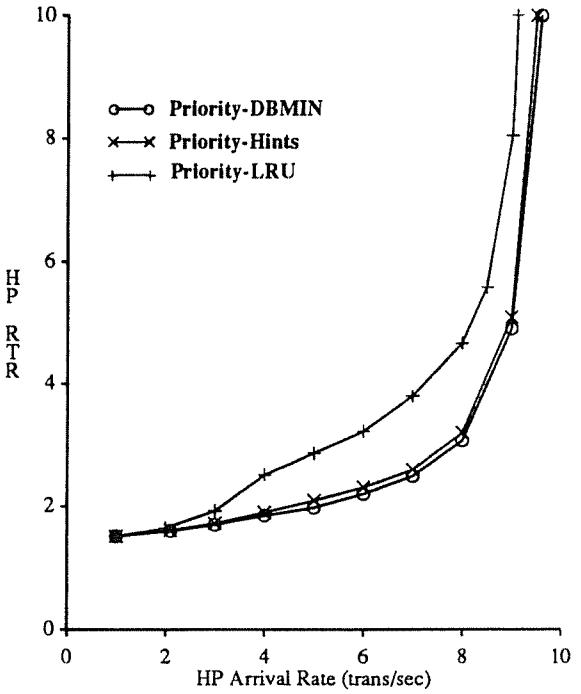

Figure 6.6(b): High Priority.
(50 buffers, 1 CPU, 4 disks.)

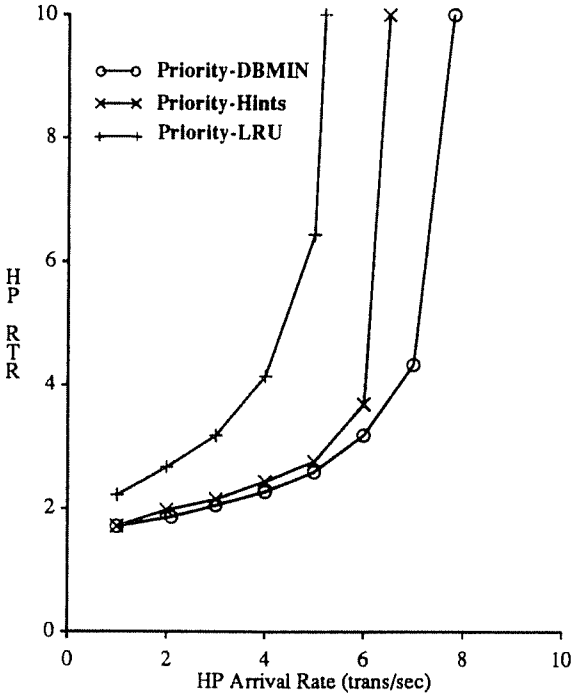

Figure 6.6(c): High Priority.
(50 buffers, 1 CPU, 2 disks.)

rithms as they do not require as much information as DBMIN-like approaches; here, we have shown that simple page-level hints can result in performance gains comparable to those provided by DBMIN-like algorithms. Similar hints are already provided to buffer managers in some systems, including DB2 [Teng84] and Starburst [Haas90].

CHAPTER 7

PRIORITY-BASED CONCURRENCY CONTROL

In our final set of experiments, we investigate the use of priority in concurrency control. As demonstrated in [Agra87], even in the absence of priority scheduling, the relative performance of different concurrency control algorithms is strongly affected by the level of utilization of the physical resources of the DBMS. In a priority-oriented DBMS, the use of priority scheduling at the physical resources tends to accelerate the progress of high priority transactions compared to that of low priority transactions. Intuitively, then, as the resource utilization increases, it should become increasingly important that the DBMS prevent low priority transactions from blocking high priority transactions at the lock manager. In this chapter, we will examine the extent to which this intuition is justified by comparing the performance of the Wound-Wait/Two-Phase-Locking (WW/2PL) algorithm with standard Two-Phase-Locking (2PL). In conventional database systems, [Agra87] also found that blocking-based mechanisms for concurrency control tend to outperform restart-based methods in the presence of resource contention. In order to examine the extent to which this result holds true in priority-based systems, we also present results for the Prioritized Wound-Wait (PWW) algorithm.

7.1. Base Experiment: Moderate Data Contention, High Resource Utilization

The workload in this experiment consists of the update transactions described in Section 4.3.5. All of the transactions access 10% of the same 1000-page relation here; i.e., the database effectively consists of a single 1000-page relation with a non-clustered index. There is a 40% probability of updating a given data page; thus, the average transaction updates four pages. The asynchronous write engine is awakened at each buffer pool miss, and the *FlushThreshold* parameter is set to zero so that all the pages in the dirty list are flushed to disk whenever the write engine is activated.[1] Priority scheduling is used at all physical resources in all the experiments in this chapter, and the Priority-Hints algorithm is used

[1]Consequently, the results of the experiments described here are not affected by the *EngineSleepTime* parameter.

for buffer management. The workload-independent parameters used in this experiment are exactly the same as in the base experiment for the buffer management study; there are four CPUs and four disks in the system, and the buffer pool contains fifty buffer frames. As one would expect, given the results of our earlier experiments, the disks are the bottleneck physical resource with this configuration. *CCReqCPU*, the CPU cost of servicing a lock request, is set to zero; the cost of granting and releasing locks is assumed to be negligible compared to transaction response times. The low priority arrival rate is set at five transactions/second; this corresponds to a disk utilization of about 40% in the absence of high priority updates. Note that each transaction needs to fix two pages, and there are 50 buffers in the buffer pool, so that the maximum number of concurrent transactions is limited to 25.

In order to understand the relative impact of resource contention and data contention for this system configuration and workload, we first present Figure 7.1 (a), which shows four high priority RTR curves, labeled WW/2PL, No-CC, HPO, and HPO-No-CC. The WW/2PL curve shows the RTR for high priority transactions when the WW/2PL algorithm is used.

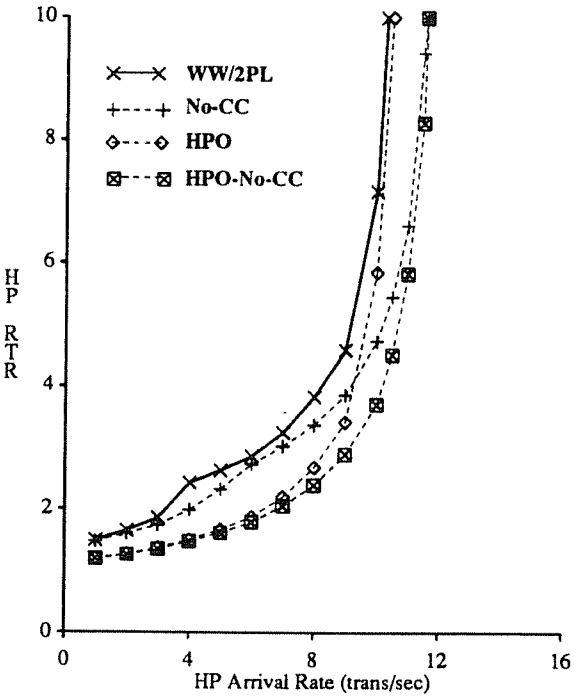

Figure 7.1(a): High Priority.
Resource Contention and Data Contention.

If concurrency control is ignored, high priority transactions have the RTRs shown by the No-CC curve; thus, the gap between the WW/2PL curve and the No-CC curve represents the effects of concurrency control conflicts, both across priority levels and among high priority transactions. The HPO (High Priority Only) curve indicates the RTRs when WW/2PL is used for concurrency control but there is no competing low priority workload. Finally, HPO-No-CC indicates the RTRs of the high priority transactions when there is no low priority workload and concurrency control conflicts are also ignored.

As one would expect, the HPO-No-CC curve represents the lowest RTRs for high priority transactions among the four curves; neither concurrency control conflicts nor resource contention with transactions of lower priority hinder the progress of high priority transactions. The gap between the HPO curve and the HPO-No-CC curve represents the effects of concurrency control conflicts among high priority transactions themselves. At low loads, this gap is negligible, but at the highest sustainable loads, the HPO curve converges to the WW/2PL curve, indicating that in these regions of operation, low priority transactions have little impact on the performance of high priority transactions. The gap between the HPO-No-CC and No-CC curves isolates the effects of resource contention between transactions of the two priority levels. Notice that there is a substantial gap between these two curves, especially at low loads, showing that resource contention across priority levels is indeed a major factor affecting high priority transactions in this region of operation. This may seem surprising in the light of the experiments described in Chapter 5. Recall, however, that updates lead to asynchronous disk writes, and that as explained in Chapter 3, disk writes are assigned a priority higher than that of any transaction's read requests. Consequently, unlike the case of read-only workloads studied earlier, low priority update transactions do affect the view of the disks seen by high priority transactions; this explains the gap between the HPO-No-CC and No-CC curves at low loads.

Finally, a comparison of the gap between the WW/2PL curve and the No-CC curve on the one hand, and the gap between the HPO and the HPO-No-CC curves on the other, indicates the impact of concurrency control conflicts across priority levels on the performance of high priority transactions. Since the WW/2PL algorithm restarts low priority transactions in favor of high priority transactions, the effect of such conflicts is felt by the high priority transactions only in the form of an increased load on the physical resources. This explains why the gap between the WW/2PL and No-CC curves is slightly larger than the gap between the other pair of curves at moderate loads. On the whole, however, WW/2PL does a pretty good job of shielding high priority transactions from any effects of concurrency control conflicts with transactions of

lower priority.

Having discussed the relative impact of resource contention and data contention, we now turn our attention to a comparison of the three alternative concurrency control algorithms. In Figure 7.1(b), we present the RTR results for high priority transactions for the PWW and 2PL algorithms together with the WW/2PL and No-CC curves from Figure 7.1(a). The gap between the No-CC curve and each of the other curves represents the effects of data contention for the corresponding algorithm. RTRs for low priority transactions are presented in Figure 7.1(c); again, a No-CC curve is included to indicate the effect of concurrency control conflicts on low priority transactions.

Figure 7.1(b) shows that the concurrency control algorithm employed has little impact on the high priority transactions response times at low loads; in this region of operation, there is little data contention. As the load increases, the WW/2PL algorithm provides the best performance, keeping the high priority RTRs close to the No-CC RTRs over the largest range of arrival rates. The PWW curve remains close to the WW/2PL curve over most of its stable range of operation; at an arrival rate of approximately eight transactions per second, however, the PWW curve branches away and becomes

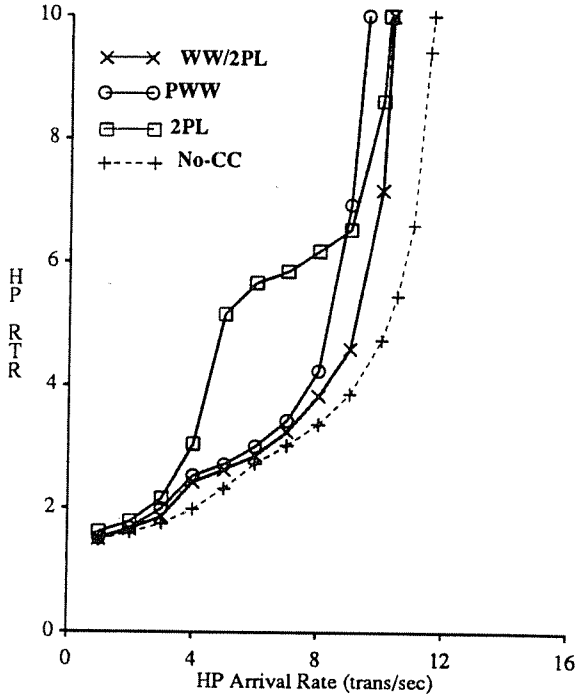

Figure 7.1(b): High Priority.

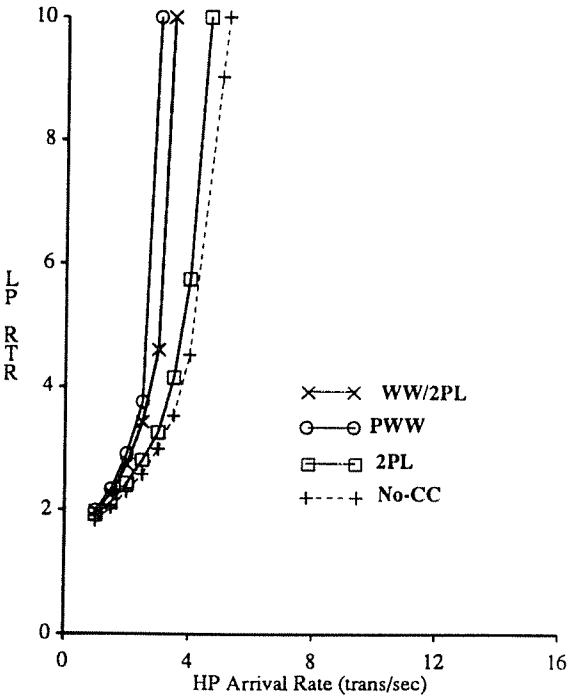

Figure 7.1(c): Low Priority.

unstable for high priority transactions. The 2PL curve, where priority is not used for concurrency control, displays a rather interesting behavior. At an arrival rate of approximately four transactions per second, the high priority RTR for 2PL shows a dramatic increase; the curve then flattens out for a while before ultimately converging towards the curve for WW/2PL. Figure 7.1(c) shows that for low priority transactions, the performance of 2PL is best, while the two priority algorithms are similar (with WW/2PL performing slightly better for low priority transactions than PWW).

The relative behavior of the WW/2PL and PWW algorithms in Figure 7.1(b) is not very surprising. Recall that within the group of high priority transactions, the PWW algorithm restarts younger transactions when a concurrency control conflict occurs, while WW/2PL allows transactions to block and uses restarts only to remove deadlocks. The average number of high priority restarts as a fraction of high priority commits for all three algorithms is presented in Figure 7.1(d). The high priority restarts are caused by the Wound-Wait mechanism in the case of PWW, by deadlocks among high priority transactions in the case of WW/2PL, and by deadlocks among both high and low priority transactions in 2PL. As the load on the system increases, and data contention among high priority transactions increases, PWW restarts far more high

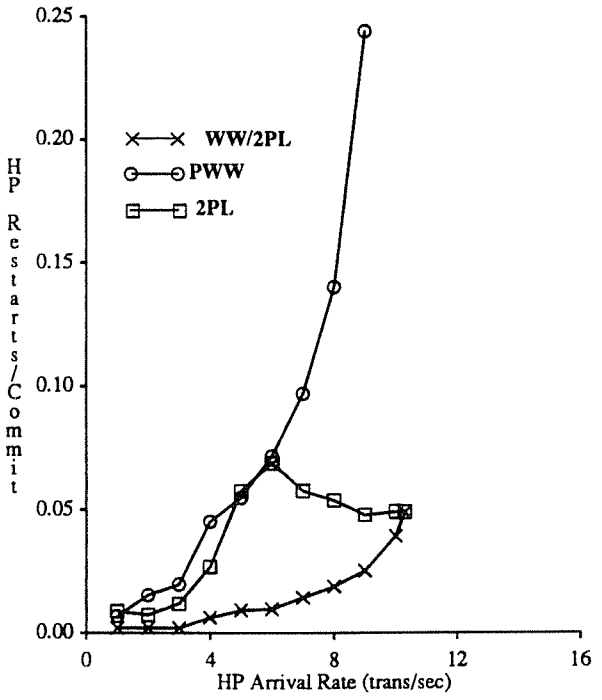

Figure 7.1(d): High Priority Restarts.

priority transactions than WW/2PL, thus leading to worse performance. This result confirms a conclusion of [Agra87] for priority-oriented DBMSs; i.e., restart-based algorithms tend to perform worse than blocking algorithms in the presence of significant resource contention.

The relative behavior of high priority transactions using 2PL versus WW/2PL is more interesting. In order to explain this behavior, we present Figures 7.1(e) and 7.1(f). Figure 7.1(e) shows the total number of transactions within the system, as well as the number of low priority transactions, under 2PL and WW/2PL. Figure 7.1(f) presents *CCBlockFrac* curves for the two algorithms, where the CCBlockFrac of a transaction is defined as the total time spent by a transaction waiting for locks divided by $E_{standalone}$. Note the similarity in the general shapes of the following three curves for the 2PL algorithm: the curve showing the number of low priority transactions in Figure 7.1(e), the CCBlockFrac curve in Figure 7.1(f), and the restart curve in Figure 7.1(d). In each case, the curve for 2PL peaks at an arrival rate of between five and six transactions per second, and then begins to fall, gradually approaching the curve for the WW/2PL algorithm. The reason for this similarity is that the concurrency control behavior of high priority transactions is governed largely by the number of

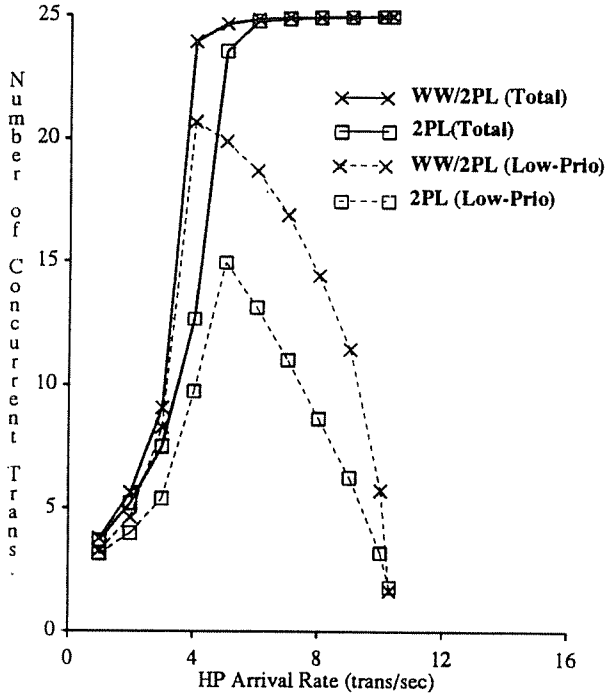

Figure 7.1(e): Number of Concurrent Transactions

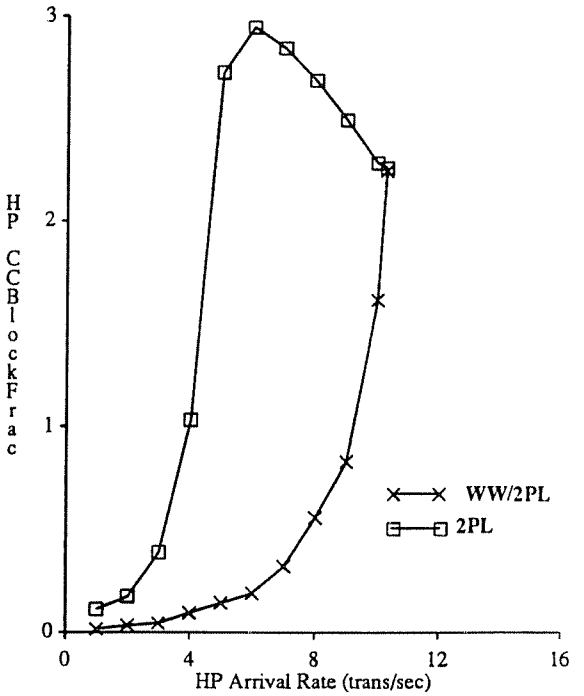

Figure 7.1(f): CCBlockFrac.

competing low priority transactions because of priority inversion under 2PL.

The 2PL lock manager provides the same level of service to all transactions regardless of priority; however, all other schedulers in the system are trying to favor high priority transactions over those at low priority. The result is that some high priority transactions are made to wait at the lock manager by low priority transactions, while these low priority transactions are prevented from making progress at the CPUs and the disks by other high priority transactions. Figure 7.1(e) shows that the number of active low priority transactions in the system rises until the system saturates at around five high priority transactions/second; the use of priority for admission control ensures that the number of concurrent low priority transactions then begins to fall. This is why the negative low priority effects peak for 2PL in Figures 7.1(d) and 7.1(f) at a load of five to six transactions/second; i.e., as the high priority load increases, the role played by low priority transactions in determining high priority performance becomes smaller and smaller. Ultimately, after almost all of the low priority transactions have been driven out of the system, both WW/2PL and 2PL show the same behavior, since the two algorithms deal identically with transactions of the same priority level. Notice also that the relative performance of 2PL and WW/2PL is similar to the relative performance of the AD and ACD curves of Figure 5.1(a); the 2PL and AD curves reflect cases where priority is ignored in making scheduling decisions at a bottleneck resource, while the ACD and WW/2PL curves reflect the use of priority everywhere.

The base experiment has shed light on a number of issues related to concurrency control in a priority context. Firstly, the use of priority for scheduling physical resources alone does not suffice to attain the objectives of priority scheduling when the workload includes updates. If priority is ignored in making concurrency control decisions, the response time of high priority transactions deteriorates rapidly as a result of priority inversions. Secondly, as discussed earlier, low priority update transactions can increase the level of disk contention seen by high priority transactions; this is because their updates are written to disk at high priority. Finally, in spite of this last effect, the WW/2PL algorithm succeeds in providing a preemptive-resume-like view of a priority DBMS in the presence of concurrency control conflicts.

7.2. Experiment 2: Varying Data Contention

In the base experiment, the probability of updating a data page was set at 40%, so each transaction updated an average of four pages. In this experiment, we vary the page update probability, keeping the other simulation parameters the same as in the base experiment. In Figure 7.2(a), we present high priority RTRs for the three algorithms and the No-CC

case when the probability of updating a page is set at 20%. Figure 7.2(b) shows the corresponding curves when the update probability is set at 60%. The trends for low priority transactions were very similar to those of Figure 7.1(c), so they are not shown here.

Comparing Figures 7.2(a), 7.1(b), and 7.2(b), we see that the relative performance trends remain similar as the update probability, and thus the level of data contention, is varied. In each case, WW/2PL provides the best performance, PWW is slightly worse, and 2PL shows the same trend relative to WW/2PL as in the base experiment. As one would expect, the importance of using priority at the concurrency control manager increases with data contention. In Figure 7.2(a), the RTR of high priority transactions is increased by at most 70% or so due to data contention, while in Figure 7.2(b), the worst-case penalty increases to about 150%.

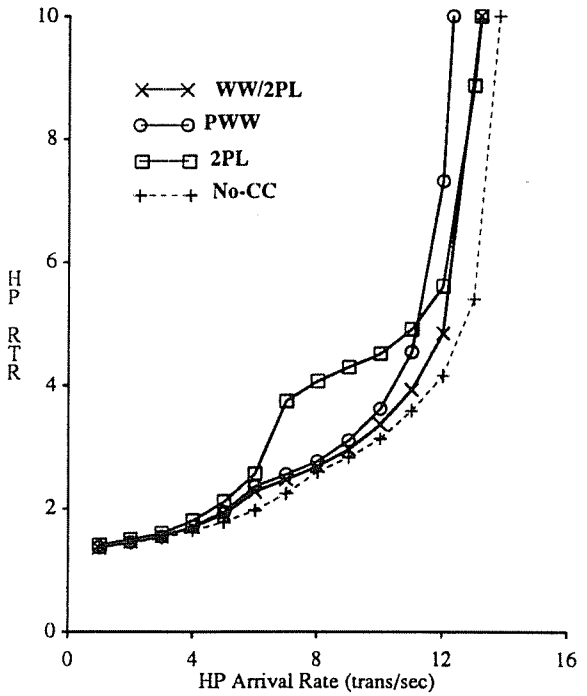

Figure 7.2(a): High Priority.
(Update Probability = 20%.)

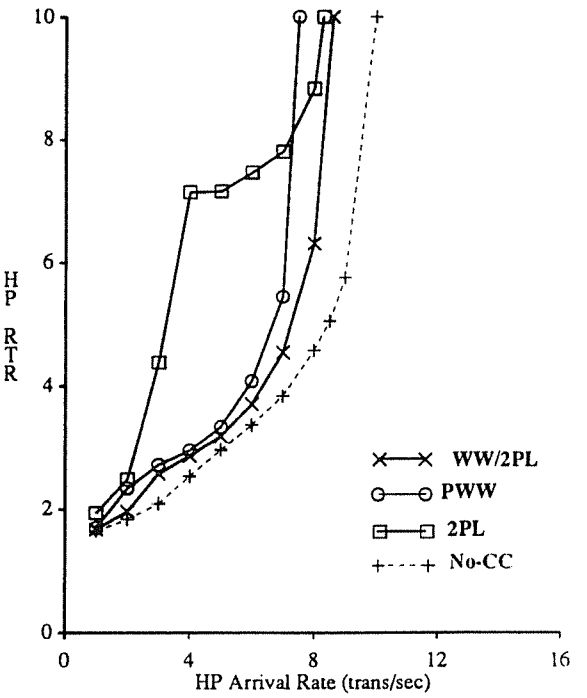

Figure 7.2(b): High Priority.
(Update Probability = 60%.)

7.3. Experiment 3: Low Resource Contention

In the experiments thus far in this chapter, there were four disks and four CPUs in the system. In this experiment, we double the number of disks and CPUs to eight each while keeping the remaining simulation parameters set as they were in the base experiment. The update probability is once again set at 40% here.

In Figure 7.3(a), the high priority RTRs for the WW/2PL, No-CC, HPO, and HPO-No-CC cases are shown for this larger configuration. Contrasting these results with those of the base experiment shown in Figure 7.1(a), two major differences can be seen. First, the four algorithms are closer together at low loads in Figure 7.3(a); not surprisingly, this indicates that disk contention due to low priority updates is not as much of an issue with the larger system configuration. Secondly, in a related effect, the gap between the No-CC curve and the WW/2PL curve (as well as the HPO-No-CC and HPO curves) is relatively larger in Figure 7.3(a); this reflects the relative increase in the influence of concurrency control conflicts on system performance.

Figure 7.3(b) shows the RTRs for high priority transactions for the three concurrency control algorithms studied, while Figure 7.3(c) shows the corresponding low priority results. Performance is better for both priority levels since resource contention has decreased. The overall trends are similar to those seen in Figure 7.1(b) and 7.1(c), however, except that there is now a relatively larger gap between the No-CC curves and the others. The main point to notice here is that 2PL's priority inversion problem is less significant with decreased resource contention, both in the range of loads over which it affects performance and in the extent of the performance penalty. This is to be expected since the increased availability of physical resources for low priority transactions means that they do not block high priority transactions for as long as they did when there was a high level of resource contention. Finally, a comparison of Figure 7.3(b) with Figure 7.1(b) shows that, although decreased resource contention reduces the performance differences between the two priority-based algorithms somewhat, the WW/2PL algorithm still outperforms the PWW algorithm under high loads.

7.4. Conclusions

The experiments in this chapter have shown the importance of using priority for resolving concurrency control conflicts in a priority-oriented DBMS. If priority is not used in concurrency control, high priority transactions' response times increase dramatically as they are blocked by low priority transactions at the lock manager. If priority is used via the

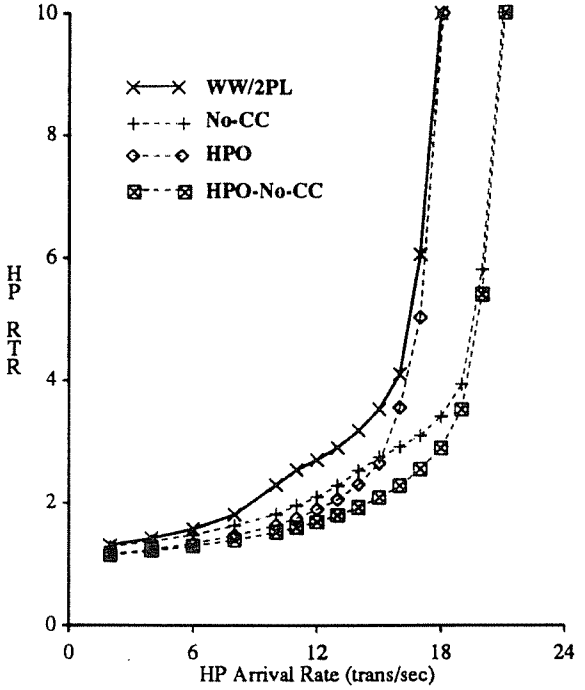

Figure 7.3(a): High Priority Resource Contention and Data Contention.
(8 CPUs, 8 disks.)

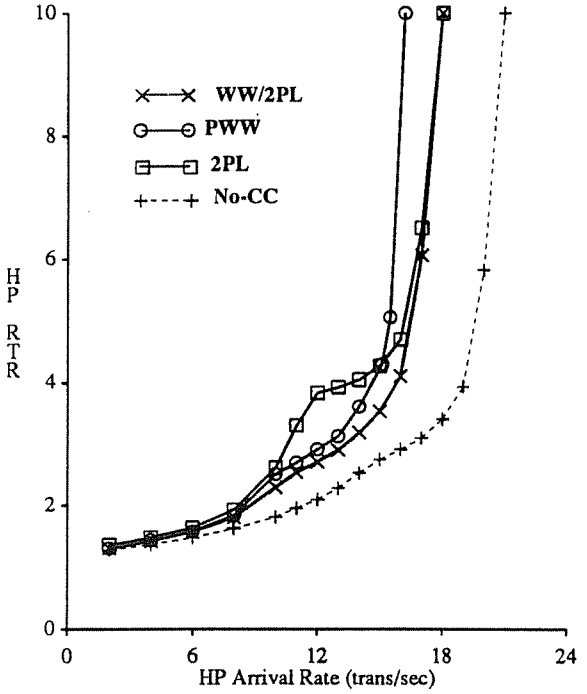

Figure 7.3(b): High Priority.
(8 CPUs, 8 disks.)

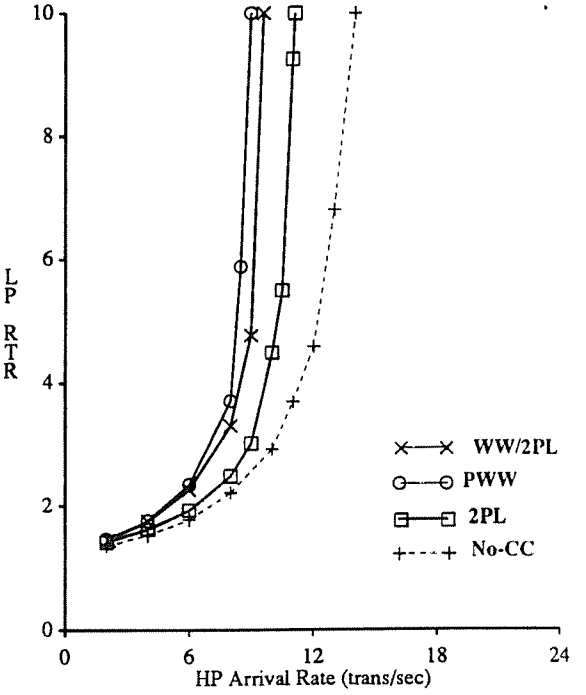

Figure 7.3(c): Low Priority.
(8 CPUs, 8 disks.)

WW/2PL algorithm, however, the effect of data contention across priority levels becomes negligible for high priority transactions and the DBMS behaves much like a preemptive-resume server. As in conventional database systems, the relative impact of concurrency control conflicts is related to the physical resource utilization levels of the DBMS; in general, the importance of priority scheduling increases both with resource utilization and with data contention. Finally, the experiments described here show that blocking-based algorithms lead to better performance than restart-based algorithms in priority-oriented database systems with limited resources, just as they do in conventional databases.

CHAPTER 8

SUMMARY AND FUTURE RESEARCH

8.1. Summary and Conclusions

As database management technology is applied to increasingly complex domains, where transactions may differ in relative importance, the need for priority scheduling of DBMS resources is likely to grow as well. For example, some applications such as stock trading may associate deadlines with individual transactions, and the performance objective may be to maximize the number of transactions that meet their deadlines. In other applications, the customers of a DBMS may be divided into classes, with a different level of service (quantified by a measure such as average per-class response time) being promised to each class. In each case, some flavor of priority scheduling can be used by the DBMS to achieve the desired performance goals.

In this dissertation, we have examined the implications of using priority to schedule the resources of a DBMS in an environment where each transaction is assigned one of a set of priority levels. Priorities are assumed to be assigned by an agent external to the DBMS, and the priority of a transaction remains unchanged during its lifetime. A fundamental assumption underlying this research is that the goal of priority scheduling in such an environment is to emulate a *preemptive-resume* server as closely as possible. That is, transactions of a given priority level should be provided the same service they would experience in a system that does not serve transactions of lower priority; in effect, lower priority transactions should be made "invisible."

With this objective in view, we began by considering the architectural consequences of using priority scheduling in a DBMS. There are several places in a DBMS where priority can be incorporated in making scheduling decisions: for admission control, for CPU and disk scheduling, for buffer allocation and replacement, and for concurrency control. In each case, priority scheduling can reduce the waiting time for high priority transactions by changing the order in which outstanding requests are served. Of course, there is a corresponding increase in the waiting time for low priority transactions;

this waiting time tradeoff is well-known in priority scheduling theory. Priority scheduling becomes an interesting problem in a database context, however, because of the variety and multiplicity of the resources. In addition, the use of priority can actually lead to an increase in the overall amount of service required at some resources; such resources include disks and main memory buffers.

The next step of our study was to design a set of priority scheduling algorithms for use at each of the key resources identified in our architectural analysis. For CPU and disk scheduling, we proposed simple extensions to well-known scheduling algorithms. A simple, nonpreemptive, priority-based approach was suggested for CPU scheduling, where transactions voluntarily gave up the CPU after short bursts of use. For disk scheduling, we presented the Prioritized Elevator algorithm; in this extension of the well-known Elevator algorithm, requests are grouped on a priority basis and the Elevator algorithm is used within each group.

For priority-based buffer management, we proposed three possible approaches. Priority-Hints is a new algorithm that uses hints supplied by the database access methods in making buffer replacement decisions. Priority-LRU is an extension of the commonly used Global LRU buffer management policy. Finally, Priority-DBMIN is a relatively sophisticated policy, based on the DBMIN algorithm, where the buffer manager uses information supplied by the query optimizer. Each of these algorithms uses priority and some level of information related to transactions' expected page access behavior to make admission control, buffer allocation, and buffer replacement decisions. For concurrency control, a hybrid algorithm, Wound-Wait/Two-Phase Locking, was presented. The key idea here is to resolve lock conflicts in favor of higher priority transactions, with low priority lock holders being restarted when they would otherwise block a high priority transaction. To handle conflicts among transactions of the same priority, standard two-phase locking with deadlock detection is used.

Having developed a set of priority scheduling algorithms, we then constructed a performance model of a priority-oriented DBMS. This model served as the vehicle for simulation experiments conducted to explore the impact of priority scheduling in a DBMS and to investigate the relative performance of the proposed alternative priority scheduling algorithms. Three sets of experiments were conducted.

The first set of experiments focused on admission control, CPU scheduling, and disk scheduling in order to determine the extent to which the preemptive-resume performance objective can be attained. We found that it was indeed possible for a DBMS to emulate a preemptive-resume server quite closely, in spite of the multiplicity and heterogeneity of its resources.

However, this goal could not be achieved by simply using a priority-based scheduler at the system's bottleneck physical resource, as the system itself became the bottleneck when priority scheduling was not used for admission control. Priority must be used both for resource scheduling and for admission control for preemptive-resume-like behavior to be attained. A second result of the first set of experiments was an important insight into the seek time penalty associated with priority disk scheduling. We found that, although the average disk access time increases as a consequence of priority, the reduction in disk waiting time for high priority transactions is sufficient to make priority-based disk scheduling worthwhile. The seek time penalty increases with the number of priority levels, however, suggesting that, if there are many levels of priority in the system, it may be advisable to map transaction priorities onto a smaller set of priority levels for disk scheduling.

In the second set of experiments, we investigated the role of priority in making buffer allocation and replacement decisions. The main objective of these experiments was to understand the tradeoff between the level of workload-related information needed by each algorithm and the resulting performance advantages in a priority context. Our results showed that buffer management can affect the performance of a priority-based DBMS significantly. The Priority-Hints algorithm was found to be superior to the Priority-LRU algorithm over almost the entire spectrum of workloads examined; the two algorithms provided similar performance only in cases where the workload made buffer replacement a non-issue. For most workloads, the new Priority-Hints algorithm performed almost as well as Priority-DBMIN. Moreover, for some workloads with a high degree of data sharing, Priority-Hints actually outperformed Priority-DBMIN, despite the fairly detailed workload information needed by Priority-DBMIN. This result is significant for several reasons. First, though earlier studies demonstrated that DBMIN performs better than simple strategies such as LRU [Chou85], most existing database systems continue to use LRU-based approaches because they require much less information than DBMIN does. Second, the type of hints being provided to the buffer manager in Priority-Hints are already available in several existing database systems [Teng84, Haas90]. Thus, Priority-Hints combines the advantages of both DBMIN-like approaches and LRU-based approaches, providing good performance while requiring relatively little information.

Our final set of experiments explored the importance of priority in concurrency control. Without priority-based concurrency control, we found that when data contention was significant, the preemptive-resume performance objective could not be attained even if all other resources were scheduled using priority. This is because priority inversions occurred; high priority transactions were blocked by low priority transactions at the lock manager, while low priority transactions could

make little progress because of priority-based resource scheduling. When priority was used for concurrency control via the Wound-Wait/Two Phase Locking algorithm, however, the DBMS again achieved behavior similar to that of a preemptive-resume server.

To summarize, this study has shown that it is indeed possible to make a database management system behave much like a single resource preemptive-resume server with the help of appropriate scheduling algorithms. This is in spite of the facts that the DBMS has multiple heterogeneous resources, that preemptive scheduling cannot be used at every resource, and that priority scheduling involves overheads at some of the resources.

8.2. Directions for Future Research

Our research has demonstrated that the concept of priority can be incorporated successfully into a DBMS. It has also pointed out some of the potential dangers of DBMS priority scheduling, as the use of priority can increase the overall service requirements associated with a DBMS workload. The insights provided by this research can be used to further explore a number of interesting issues.

One attractive avenue for future work is to actually build a prototype priority-oriented DBMS. One could then select actual applications, such as automated stock trading [Pein88], and test the proposed algorithms for each application domain in a more realistic environment. Such tests could also lead to the development of a set of benchmarks for priority-based database management.

In this thesis, we have assumed that an external agent is responsible for assigning transaction priorities. In applying the key ideas of this thesis to applications where priority is a means to another end, the problem of priority assignment must also be addressed. In some rule processing applications, for example, the relative importance of incoming transactions may not be known *a priori*, since each incoming transaction may potentially lead to the triggering of many rules, and rules may differ widely in priority. That is, the consequences of any given incoming transaction are not known in advance. The problem of priority assignment is significant here because, as we have shown, the relationship between priority and performance is nonlinear; the performance penalty suffered by low priority transactions can be severe. If an incoming transaction is assigned low priority in a heavily-loaded rule-processing system, for example, the triggering of high priority rules may be delayed; on the other hand, if incoming transactions are all run at high priorities, the response times of the triggered rules will increase. Thus, priority assignment is a non-trivial, application-dependent problem that deserves an

analysis of its own.

DBMS priority scheduling involves overheads, as shown by our experiments, which raises interesting performance questions for systems where transaction priorities are dynamic. A priority-oriented DBMS is non-conservative, i.e., the total work done increases purely as a consequence of the use of priority. When transaction priorities remain fixed, the overheads are absorbed primarily by low priority transactions, as we have shown. However, if transaction priorities can change over time, each transaction may suffer at different stages of its lifetime (depending on its current priority relative to that of the other transactions). Consequently, the performance benefits gained while a transaction runs at high priority may be lost in periods when its priority is lower, particularly given the nonlinear relationship discussed above. An analysis of the tradeoff between the fraction of time spent by a transaction at low priority and the benefits (or losses) due to priority scheduling is thus an important area for future research. Such an analysis could help to extend our ideas to rule-based and/or real-time database environments.

Another possible avenue for future research would be to consider alternatives to the preemptive-resume paradigm as the guiding philosophy for a priority-oriented DBMS. For example, instead of allowing unrestricted use of the DBMS resources by the transactions of the highest priority, one could "ration" the resource usage for each priority class. While this does not seem difficult for CPU scheduling, the extension of this idea to the scheduling of multiple DBMS resources may be an interesting and non-trivial possibility for future research.

Finally, this thesis has assumed a centralized, priority-oriented DBMS where each transaction has a single thread of control. The relaxation of these two assumptions, i.e., a study of priority DBMS scheduling in a distributed and/or parallel processing environment, also represents a potentially challenging area for future research.

REFERENCES

[Abbo88] Abbott, R., and Garcia-Molina, H., "Scheduling Real-time Transactions: A Performance Evaluation," *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, CA, August 1988.

[Abbo89] Abbott, R., and Garcia-Molina, H., "Scheduling Real-time Transactions With Disk Resident Data," *Proceedings of the 15th International Conference on Very Large Data Bases*, Amsterdam, August 1989.

[Agra87] Agrawal, R., Carey, M. J., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transactions on Database Systems*, Vol. 12, No. 4, December 1987.

[Bern80] Bernstein, P., and Goodman, N., "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," *Proceedings of the 6th International Conference on Very Large Data Bases*, Montreal, October 1980.

[Bitt88] Bitton, D., and Gray, J., "Disk Shadowing," *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, CA, August 1988.

[Blas79] Blasgen, M., et al, "The Convoy Phenomenon," *Operating Systems Review*, Vol. 13, No. 2, April 1979.

[Buch89] Buchmann, A. P., McCarthy, D. R., Hsu, M., and Dayal,U., "Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control," *Proceedings of the 5th International Conference on Data Engineering*, Los Angeles, CA, February 1989.

[Buze83] Buzen, J. P., and Bondi, A. B., "Preemptive Resume in M/M/m," *Operations Research*, Vol. 31, No. 3, May-June 1983.

[CADF90] The Committee for Advanced DBMS Function, *Third-Generation Data Base System Manifesto*, Memorandum No. UCB/ERL M90/28, Electronics Research Laboratory, University of California, Berkeley, CA, April 1990.

[Care88] Carey, M. J., and Livny, M., "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, CA, August 1988.

[Casa79] Casanova, M., *The Concurrency Control Problem for Database Systems*, Ph. D. Thesis, Computer Science Department, Harvard University, Cambridge, MA, 1979.

[Chan87] Chang, H.-Y., *Dynamic Scheduling Algorithms for Distributed Soft Real-Time Systems*, Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI, November 1987.

[Chou85] Chou, H-T., and DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, August 1985.

[Coff68] Coffman, E. G., and Kleinrock, L., "Computer Scheduling Methods and Their Countermeasures," *Proceedings of the AFIPS Spring Joint Computer Conference*, April 1968.

[Coff73] Coffman, E. G., and Denning, P. J., *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Coff76] Coffman, E. G., *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York, NY, 1976.

[Conc90] Concurrent Computer Corporation, *RTU Operating System Version 6.0 Product Overview*, Tinton Falls, NJ, 1990.

[Effe84] Effelsberg, W., and Haerder, T., "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, Vol. 9., No. 4, December 1984.

[Gray79] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Haas90] Haas, L., et al, "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990.

[Hari90] Haritsa, J., Carey, M. J., and Livny, M., "On Being Optimistic About Real-Time Constraints," *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, Nashville, TN, April 1990.

[Hsu83] Hsu, M., and Madnick, S., "Hierarchical Database Decomposition — A Technique for Database Concurrency Control," *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Atlanta, GA, March 1983.

[Hsu86] Hsu, M., and Chan, A., "Partitioned Two-Phase Locking," *ACM Transactions on Database Systems*, Vol. 11, No. 4, Dec. 1986.

[Huan89] Huang, J., Stankovic, J. A., Towsley, D., and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE 1989.

[Jack60] Jackson, J. R., "Queues With Dynamic Priority Discipline," *Management Science*, Vol. 8, 1960.

[Jais68] Jaiswal, N. K., *Priority Queues*, Academic Press, New York, NY, 1968.

[Jens86] Jensen, E. D., Locke, C. D., and Tokuda, H., "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of the IEEE Real-Time Systems Symposium*, IEEE 1986.

[Klei76] Kleinrock, L., *Queueing Systems*, John Wiley and Sons, New York, NY, 1976.

[Knap87] Knapp, E., "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, Vol. 19, No. 4, December 1987.

[Kort90] Korth, H. F., Soparkar, N., and Silberschatz, A., "Triggered Real-Time Databases with Consistency Constraints," *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems* Vol. 6, No. 2, June 1981.

[Lamp68] Lampson, B. W., "A Scheduling Philosophy for Multiprocessing Systems," *Communications of the ACM*, Vol. 11, No. 5, May 1968.

[Lin88] Lin, K-J., and Lin, M-J., "Enhancing Availability in Distributed Real-Time Databases," *Special Issue on Real-Time Data Base Systems, SIGMOD Record 17*, No. 1, March 1988.

[Lind79] Lindsay, B., et al, *Notes on Distributed Databases*, Rep. No. RJ2571, IBM San Jose Research Laboratory, San Jose, CA, 1979.

[Livn88] Livny, M., *DeNet User's Guide*, Version 1.0, Computer Sciences Department, University of Wisconsin, Madison, WI, 1988.

[Mena78] Menasce, D., and Muntz, R., "Locking and Deadlock Detection in Distributed Databases," *Proceedings of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, San Fransisco, CA, August 1978.

[Pein88] Peinl, P., Reuter, A., and Sammer, H., "High Contention in a Stock Trading Database: A Case Study," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, June 1988.

[Pete86] Peterson, J., and Silberschatz, A., *Operating Systems Concepts*, Addison-Wesley, 1986.

[Reed78] Reed, D., *Naming and Synchronization in a Decentralized Computer System*, Ph. D. Thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1978.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems," *ACM Transactions on Database Systems*, Vol. 3, No. 2, June 1978.

[Sacc86] Sacco, G.M., and Schkolnick, M., "Buffer Management in Relational Database Systems," *ACM Transactions on Database Systems*, Vol 11., No. 4, December 1986.

[Schw87] Schwan, K., Bihari, T., Weide, B. W., and Taulbee, G., "High-Performance Operating System Primitives for Robotics and Real-Time Control Systems," *ACM Transactions on Computer Systems*, Vol. 5, No. 3, August 1987.

[Sha87] Sha, L., Rajkumar, R., Lehoczky, J. P., *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, Technical Report CMU-CS-87-181, Departments of CS, ECE, and Statistics, Carnegie Mellon University, 1987.

[Sha88] Sha, L., Rajkumar, R., Lehoczky, J. P., "Concurrency Control for Distributed Real-Time Databases," *Special Issue on Real-Time Data Base Systems, SIGMOD Record 17*, No. 1, March 1988.

[Shap86] Shapiro, L.D., "Join Processing in Database Systems With Large Main Memories," *ACM Transactions on Database Systems*, Vol. 11, No. 3, September 1986.

[SIGM88] *SIGMOD Record*, Vol. 17, No. 1, Special Issue on Real-Time Data Base Systems, S. Son, editor, March 1988.

[Stan88a] Stankovic, J., Zhao, W., "On Real-Time Transactions," *Special Issue on Real-Time Data Base Systems, SIGMOD Record 17*, No. 1, March 1988.

[Stan88b] Stankovic, J., "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer*, Vol. 21, No. 10, October 1988.

[Teng84] Teng, J., and Gumaer, R. A., "Managing IBM Database 2 Buffers to Maximize Performance," *IBM Systems Journal*, Vol. 23, No. 2, 1984.

[Teor72] Teorey, T., and Pinkerton, T., "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, Vol. 15, No. 3, March 1972.

[Zhao87a] Zhao, W., Ramamritham, K., and Stankovic, J., "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 5, May 1987.

[Zhao87b] Zhao, W., Ramamritham, K., and Stankovic, J., "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers*, Vol C-36, No. 8, August 1987.