On Using Priority Inheritance In Real-Time Databases [*]

Jiandong Huang
Department of Electrical and Computer Engineering

John A. Stankovic
Krithi Ramamritham
Don Towsley
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

March, 1991

Abstract

Due to resource sharing among tasks, priority inversion can occur during priority-driven preemptive scheduling. In this work, we investigate solutions to the priority inversion problem in a real-time database environment where two-phase locking is employed for concurrency control. We examine two basic schemes for addressing the priority inversion problem, one based on priority inheritance and the other based on priority abort. We also study a new scheme, called conditional priority inheritance, which attempts to capitalize on the advantages of each of the two basic schemes. In contrast with previous results obtained in real-time operating systems, our performance studies, conducted on an actual real-time database testbed, indicate that the basic priority inheritance protocol is inappropriate for solving the priority inversion problem in real-time database systems. We also show that the conditional priority inheritance scheme and the priority abort scheme perform well for a wide range of system workloads.

Keywords - real-time database systems, concurrency control, scheduling, priority inheritance, priority inversion, implementation, performance evaluation.

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

1 Introduction

Priority-driven preemptive scheduling is an approach commonly used in real-time systems. However, in this case, *priority inversion* may occur due to the sharing of resources among tasks [11]. Priority inversion is said to occur when a higher priority task must wait for a lower priority task to release a shared resource. Priority inversion can cause unbounded delay to high priority tasks. This prolonged wait may result in the higher priority tasks missing their deadlines, thus degrading the performance of real-time systems.

Priority inversion has been studied in real-time operating systems [10]. The basic approach proposed to rectify the problem is the *priority inheritance protocol* [11], where the low priority task is allowed to execute at the highest priority of all of the high priority tasks that it blocks, and eventually return to its original priority level after using the shared resources that caused the blocking. The idea is to allow the low priority task to run and release its resources quickly so that the higher priority tasks can continue its execution. The performance studies based on the rate-monotonic scheduling framework [10] have demonstrated that the priority inheritance protocol, applied to the shared resources accessed via semaphores, provides a significant performance advantage.

The goal of this work is to investigate the priority inversion problem in a particular real-time environment - a real-time database system where a two-phase locking concurrency control protocol is employed to enforce data consistency. Unlike real-time systems where task conflict over shared system resources may last for a short period of time, in real-time database systems, transaction conflict over shared data may last as long as the execution time of a transaction. It can last even longer in the case of cascaded blocking. Here we consider tow basic schemes for rectifying the problems due to priority inversion, one based on priority inheritance and the other based on *priority abort* (abort the lower priority transaction when priority inversion occurs). We seek answers to the following questions.

- Is the priority inheritance scheme an appropriate method for solving the priority inversion problem in real-time database systems?
- Which mechanism, priority inheritance or priority abort, is better?
- Is there a better approach other than these two basic schemes?

In this study, we first evaluate the implications of using the priority inheritance and priority abort schemes. Based on this evaluation, we develop a combined priority abort and priority inheritance scheme, called *conditional priority inheritance*, which attempts to capitalize on the advantages of each of the above two schemes. Specifically, this scheme uses priority inheritance once a transaction is near completion; otherwise it aborts the transaction if its priority is low. The performance studies conducted on our testbed indicate that the basic priority inheritance protocol is inappropriate for solving the priority inversion problem in real-time database systems. We show that a better solution is to make use of the priority-abort based strategies, namely the simple priority abort and the conditional priority inheritance schemes. Moreover, there appears to be no clear winner between these two schemes. The choice depends on the degree of resource contention and the transaction deadline distribution.

This paper is organized as follows. In Section 2, we first describe the problem of priority inversion in real-time database systems. Then we discuss the implications of using the priority inheritance scheme and the priority abort scheme under two-phase locking, and propose a conditional priority inheritance scheme for resolving priority inversion. In Section 3, we describe the real-time database testbed that was used for the performance studies. The experimental results are presented and discussed in detail in Section 4. Finally, we give concluding remarks and outline future work in Section 5.

2 Transaction Scheduling Under Two-Phase Locking

In this study, we consider real-time database systems where a deadline is associated with each transaction and data consistency is maintained according to the serializability correctness criterion. In such a real-time environment, the execution of concurrent transactions should be scheduled so as to meet their timing constraints, and at the same time, transaction execution should also be governed by a concurrency control protocol in order to achieve serializability. In this work, we study real-time database systems that employ *priority-driven preemptive scheduling*[1] for CPU scheduling and *two-phase locking* for concurrency control.

For real-time priority-driven preemptive scheduling, each transaction is assigned a priority according to its deadline. The execution of concurrent transactions is scheduled based on their assigned priority. Ideally, a high priority transaction should never be blocked by any lower priority transaction. In particular, a transaction may get CPU service by preempting a lower priority transaction from the CPU. The preempted transaction is placed in the ready queue and resumes its execution when the CPU becomes available.

Under two-phase locking, on the other hand, a transaction must obtain a lock before accessing a data object and release the lock when it terminates (commits or aborts). A lock-requesting transaction will be placed in a wait queue if its lock mode is found to be incompatible with that of the lock-holding transaction(s). The queued transaction can proceed only when it is granted the lock.

When the priority-driven preemptive scheduling approach and the two-phase locking protocol are simply integrated together in a real-time database system, a problem, known as *priority inversion*, arises. Priority inversion occurs when a higher priority transaction must wait for the execution of lower priority transaction(s). This scheduling problem results from the use of locking for concurrency control. Under the two-phase locking protocol, as we just described, a high priority transaction which requests a lock may conflict with a lower priority transaction which holds the lock. The high priority transaction has to wait for the lock while the lower priority transaction continues. Here transaction wait is necessary in order to ensure data consistency. However, blocking with priority inversion implies that higher priority transactions are waiting while lower priority transactions are in execution.

[1]Non-preemptive scheduling is not appropriate in a database setting since long transactions and I/O cause unnecessary blocking to other transactions thus preventing them from meeting their deadlines.

This defeats the purpose behind priority assignment. Even worse, the blocking delay may be unbounded. This can be illustrated by the following two, or combination of the two, situations.

- **Data resource blocking**: A high priority transaction can be blocked by more than one low priority transaction when a data access conflict occurs. For instance, a transaction requesting a *write* lock on a data object may conflict with a group of transactions holding *read* locks on that object. Furthermore, it is possible that the lock holder(s) is waiting for other transactions due to data access conflict. Thus, the blocked high-priority transaction can be delayed for an unbounded period of time.
- **CPU resource blocking**: The blocking of a high priority transaction can be prolonged by *intermediate priority* transactions that have no access conflict with it but need CPU time. Assume that transaction T_H has higher priority than T_L and is blocked by T_L due to an access conflict. While T_H is waiting for the lock and T_L is executing, some transaction(s) T_M may arrive, whose assigned priority lies between the priorities of T_H and T_L. In this case, T_M will preempt T_L and take over the CPU. Clearly, even if T_M has no access conflict with the higher priority T_H, its execution will delay T_H. Thus, the high priority transaction can be blocked indefinitely due to the execution of the intermediate priority transactions.

Clearly, the blocking resulting from priority inversion is a serious problem for real-time transaction scheduling. In the following, we first look at a scheme, called *priority inheritance* [12, 10], which eliminates the problem of CPU resource blocking and attempts to reduce the period of priority inversion. Then, we examine another approach based on *priority abort* [1, 6], which completely eliminates the problem of priority inversion as well as CPU resource blocking. Based on the analysis of the two schemes, we then propose a combined priority abort and priority inheritance scheme, called *conditional priority inheritance.*

Note that the *priority ceiling protocol* [12, 3, 13] is another scheme developed to solve the priority inversion problem. Under this scheme, the priority inversion is bounded to no more than one transaction execution time. The scheme also has the property of deadlock freedom. However, this scheme requires prior knowledge about the data objects to be accessed by each transaction. This condition appears to be too restrictive in many applications. Moreover, the scheme becomes extremely conservative, with respect to the number of transactions that can execute concurrently, if transactions can access any data objects in the database. Due to these reasons, we do not consider this particular scheme in this study.

Priority Inheritance (PI)

Under priority inheritance, when priority inversion occurs, the low priority transaction holding the lock will execute at the priority of the highest priority transaction waiting for the lock, until it terminates. Because of the increase in priority, the lock-holding transaction may run faster than it will without priority change, thus releasing its lock quickly. As a result, the blocking time for the high priority transaction may be reduced.

The priority inheritance scheme also eliminates the problem of CPU resource blocking. Consider the example that we discussed above. Suppose T_H is blocked by T_L due to data access conflict. Then, by using priority inheritance, T_L will execute at the priority of T_H. Now if T_M, an intermediate priority transaction, arrives, it cannot preempt T_L since its priority is less than the inherited priority of T_L. Thus, T_H will not be delayed by T_M.

The priority inheritance scheme provides a significant performance improvement in real-time operating systems, as shown in [10]. However, it has shortcomings when used in real-time database systems. First, under strict two-phase locking, a transaction may hold a lock throughout its execution. This *"life-time blocking"* may be too long for a high priority transaction to wait, even though the lock-holding transaction may execute faster after inheriting the higher priority. The problem will be worse if a high priority transaction encounters priority inversion with *data resource blocking*, or if the high priority transaction is blocked by low priority transactions many times along the course of its execution, a situation called *chained blocking* [11]. Second, an increase in priority for one transaction may affect other concurrent transactions. In other words, a low priority transaction with an inherited high priority will compete for system resources (CPU, I/O, data objects, critical sections, etc.) with other non-blocking high priority transactions, which might lower the performance of those high priority transactions. Also, due to the increase in the competition for data objects, the conflict rate may increase.

As we can see, there are many ramifications when priority inheritance is combined with two-phase locking. On the one hand, the scheme may reduce the duration of priority inversion and eliminate the problem of CPU resource blocking. But on the other hand, the "life-time blocking" nature of two-phase locking may prevent the scheme from being effective.

Priority Abort (PA)

The *priority abort* scheme overcomes the priority inversion problem by aborting the low priority transaction. When a lock-requesting transaction T_1 conflicts with a lock-holding transaction T_2, T_2 is aborted if T_1's priority is higher than that of T_2; otherwise, T_1 will wait for T_2.[2] In this way, a high priority transaction will never be blocked by any lower priority transactions. Therefore, priority inversion is completely eliminated. Note that under the priority abort scheme, the problems of "life-time blocking" and "chained blocking" encountered by priority inheritance scheme disappears.

The "non-blocking" nature of the priority abort scheme is highly desirable to real-time transaction scheduling, but it may have a negative side effect on the system, i.e., it may lead to a high transaction abort rate due to data access conflict. The higher the abort rate, the more the wasted system resources. This may become a serious problem when a system already contains highly utilized resources.

Each of the two schemes, PI and PA, for addressing the priority inversion problem has its advantages and disadvantages. The main trade-off between the two schemes is the potentially long blocking time versus the high conflict abort rate. In order to minimize the effects of these two problems, we propose a combined priority inheritance and priority abort scheme, called *conditional priority inheritance.*

[2]If T_1 conflicts with more than one transaction, T_1 will abort the conflicting transactions only if its priority is higher than that of all the conflicting transactions.

Conditional Priority Inheritance (CP)

The basic idea behind this scheme is the following. When priority inversion is detected, if the low priority transaction is near completion, it inherits the priority of the high priority transaction, thus avoiding an abort with its waste of resource; otherwise, the low priority transaction is aborted, thereby avoiding the long blocking time for the high priority transaction, and also reducing the amount of wasted resources used thus far by the low priority transaction.

Here we assume that transaction length (defined as the number of *steps*) is known in advance. This assumption is reasonable for many application environments like banking and inventory management. Let L_T be the length of transaction T, x_T be the number of steps that T has executed thus far, and p_T be T's priority. Then, our conditional priority inheritance scheme can be described by the following algorithm.

```
$T_1$ requests a data object being locked by $T_2$ with an incompatible lock mode.
if $p_{T_1} \leq p_{T_2}$
  then $T_1$ waits for $T_2$
  else
        if $(L_{T_2} - x_{T_2}) > h$
          then abort $T_2$
          else $T_2$ inherits $T_1$'s priority
        end if
end if
```

In the algorithm, h is a threshold. For the lock-holding transaction with a lower priority, if its remaining work $(L_T - x_T)$ is less than h, then we apply PI; otherwise we use PA. This threshold policy, CP, is expected to reduce the blocking time with respect to PI, and to reduce the abort rate with respect to PA. We define the *priority inheritance period* to be the time interval in which the priority inheritance scheme is used rather than priority abort, i.e., the last h steps of the transaction.

The principal question regarding CP is the choice of threshold value, h. If h is too small, CP may behave like PA. On the other hand, if h is too large, CP may behave like PI. In addition, the setting of h should take into account the distribution of data conflicts with respect to transaction length. The sensitivity of h on performance has been studied through experiments and the results are discussed in Section 4.

In summary, the presented three schemes solve the priority inversion problem based on either one or both of the two basic mechanisms, namely, priority inheritance and priority abort. The trade-off among the schemes is blocking time versus resource consumption.

3 Test Environment

The three schemes, PI, PA and CP, for addressing the priority inversion problem have been implemented and evaluated on our real-time database testbed RT-CARAT [6]. In this section, we briefly introduce the test environment, including the testbed organization, the real-time transaction model, the system parameter settings, and the performance baselines and metrics.

3.1 Testbed organization

Currently, RT-CARAT is a centralized, secondary storage, real-time database testbed built on top of the VAX/VMS operating system. It contains all of the major functional components of a transaction processing system, such as transaction management, data management, log management, and communication management. The testbed is implemented as a set of cooperating server processes which communicate via efficient message passing mechanisms. A pool of transaction processes (TR's) simulate the users of the real-time database. Accordingly, there is a pool of data managers (DM's) which service transaction requests from the user processes (the TR's). There is one transaction manager, called the TM server, acting as the inter-process communication agent between TR and DM processes. Communication between TR, TM and DM processes is carried out through mailboxes, a facility provided by VAX/VMS. To be more efficient, TM and DM processes also share some information, such as transaction deadline and priority, through a common memory space, called the global section in VAX/VMS.

Using the underlying VAX/VMS operating system real-time priorities, the priority-driven preemptive scheduling is done by a CPU scheduler embedded in the TM. Upon the arrival of a new transaction, the scheduler assigns a priority to the transaction according to the CPU scheduling policy (*earliest-deadline-first* in this study). The scheduling operation is done by mapping the assigned transaction priority to the real-time priority of the DM process which carries out the transaction execution. At this point, an executing DM will be preempted if it is not the highest priority DM process at the moment, otherwise it will continue to run until it completes or until it needs to wait for an I/O. The operation on priority inheritance is incorporated in the concurrency control protocol in the DM. Similarly, the priority inheritance operation is performed by changing the priority of corresponding DM process(es).

RT-CARAT is a system that contains a fixed number of users that submit transaction requests one after another, with a certain think time (τ) in-between. This model captures many applications in the real world. For example, in an airline reservation system, there is a fixed number of computer terminals. The airline clerk at each terminal may check a flight, reserve a seat, or cancel a reservation for customers. After submitting a request to the system, the clerk waits for a result. He may submit another request after getting a response from the previous one. (On the other hand, this model certainly does not capture all applications. For instance, an open system model is more appropriate for a process control system.)

3.2 Real-time transactions

A transaction is characterized by its *length* and *deadline*. The length is specified by $T(x, y)$, where x is the number of steps that a transaction needs to execute, and y is the number of records accessed in each step. Transaction deadline is randomly generated from a uniform distribution within a deadline window, $[d\_base, \alpha \times d\_base]$, where *d_base* is the window baseline and α is a variable determining the upper bound of the deadline window. For each workload in the experiments, *d_base* is specified first by the formula:

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

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

A transaction terminates upon a normal completion or a termination abort. The latter refers to the situation where a transaction has missed its deadline and it is thus aborted by the system. The termination abort can happen, for example, in program trading in stock market when a transaction carrying out a bidding operation fails to meet the broker-specified deadline. A transaction aborted due to deadlock or data access conflict will be restarted as long as it has not passed its deadline. Hence a transaction may make multiple *runs* before it eventually terminates. Note that a restarted transaction will access the same set of records as it did in its first run.

3.3 Parameter settings

Table 1 summarizes the parameter settings in the experiments. The table is divided into two parts. The first part presents the parameters that are kept constant across all workloads, and the second part are those that change. In our experiments, two separate disks are used, one for the database and the other for the log. The database consists of 1000 physical blocks (512 bytes each) with each block containing 6 records for a total of 6,000 records. In all of the experiments, the multi-programming level in the system is 8. While this is low compared to what we would find in practice, the database size in the experiments (6000 records) is also smaller than what we would find in practice. With a proper system scaling, many factors, such as the level of data access conflict, can model practical situations. Thus, the performance results obtained from the smaller system can reflect the performance of a larger system. A database may be accessed uniformly or with a certain probability for some *hotspots*. Hotspot accesses may create high data contention among transactions. This effect, in the context of this study, can be modeled by varying the ratio of read and write transactions (see the parameter P_w in the following). For simplicity, only uniform access is considered here. Similarly, the change of external think time, τ, may affect the actual system multi-programming level and, in turn, the degree of data contention and resource contention. Since data contention and resource contention are modeled by P_w and *system type* (see below), we fix τ at 0. The number of records to be accessed per transaction step, y, is also simply fixed at 4. Thereby, the transaction length, $T(x, y)$, can be varied by one parameter x.

Experiments were conducted in two different types of systems. One is I/O bound (a VAXstation 3100/M38 with two RZ55 disks) and the other is CPU bound (a VAXstation II/GPX with two RD50 disks). The reason for this is to see how different kinds of resource contention will affect protocol performance. A critical factor in this performance study

Table 1: Experimental Settings

Parameter	Settings
Disks	disk1: database; disk2: log.
Database size	1000 blocks (6000 records)
Multiprogramming level	8
Access distribution	uniform
τ (external think time)	0
y (records accessed per trans. step)	4 records
system type	I/O bound, CPU bound
x (steps per transaction)	4 - 16 steps
P_w (prob. of write transactions)	0.2 - 1.0
α (deadline window factor)	2.0 - 6.0
h (threshold parameter)	0 - 16 steps

is transaction length $T(x, y)$, since it is directly related to transaction blocking time. As mentioned above, we vary x, while fixing y. We are also interested in the effect of P_w, the probability of write transactions among the concurrent ones on performance. This is because it directly affects data contention and, hence, the chance of priority inversion. In addition, the deadline window factor, α, is a timing-related parameter which specifies the deadline distribution of real-time transactions. The smaller the value of α, the tighter the transaction deadlines are and vice versa. Moreover, to study the conflict resolution scheme CP, we vary h, the threshold parameter.

3.4 Performance baselines and metrics

Table 2 lists the policies we examined in this performance study. Here each policy is a combination of a CPU scheduling scheme and a conflict resolution scheme. The CPU scheduling scheme is either priority-driven or non priority-driven (*multi-level feedback queue*). The former represents the nature of CPU scheduling in real-time databases, while the latter in traditional databases. Transaction priority can be assigned based on various transaction parameters, such as deadline, criticalness (i.e., the degree of importance), and length. Since the study of priority inheritance scheme is considered to be orthogonal to priority assignment policies and also meeting transaction deadline is the main concern on protocol performance, we employ the *earliest-deadline-first* (EDF) policy for transaction priority assignment. For studies on other priority assignment policies based on transaction slack time, criticalness, or length, the reader is referred to [1, 6, 4, 8].

Besides PI, PA and CP discussed in Section 2, we also look at two other policies, NRT (non-real-time) and WAIT, for the sake of performance comparisons. Representing a non real-time transaction processing system, NRT is the performance baseline in these experiments. Under NRT, transactions are scheduled by a *multi-level feedback queue* policy, and in case of data access conflict the lock-requesting transaction is always placed in a

Table 2: Policies Examined

Policy	CPU scheduling	Conflict resolution
NRT	multi-level feedback queue	always wait
WAIT	earliest deadline first	always wait
PI	earliest deadline first	priority inheritance
PA	earliest deadline first	priority abort
CP	earliest deadline first	conditional PI/PA

FIFO wait queue. In other words, transaction timing information is used neither in CPU scheduling nor in conflict resolution under NRT.

Unlike NRT, WAIT uses priority-driven CPU scheduling, based on EDF. However, it does not take transaction priority into account for conflict resolution: It is always the lock-requesting transaction that will be placed into a wait queue. This policy enables us to isolate the performance differences due to the use of PI, PA, or CP.

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

- Deadline guarantee ratio - the percentage of submitted transactions that complete by their deadlines.
- Priority inversions per run - the average number of instances, for each run, that a lock-requesting transaction is blocked by lower priority transaction(s).
- Data resource blocking - the average number of (lower priority) transactions blocking a lock-requesting transaction. (See Section 2.)
- PD abort ratio - the total number of priority aborts and deadlock aborts, divided by the number of submitted transactions.
- Number of waits per run - the average number of lock-waiting instances that a transaction encountered at each run, including blocking instances caused by priority inversion.
- Waiting time - the average waiting time (in seconds) in each wait instance.
- Total waiting time per run - *waiting time* times *number of waits per run* (in seconds).
- Wasted operations per transaction - the average total number of transaction steps wasted due to priority abort or deadlock abort for each submitted transaction.

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

The data collection in the experiments is based in the method of replication. The statistical data has 95% confidence intervals with less than $\pm$ 2% of the point estimate for deadline guarantee ratio. In the following graphs, we only plot the mean values of the performance measures.

4 Experimental Results

In this section, we present performance results for the experiments conducted on the RT-CARAT testbed. The schemes for addressing priority inversion are evaluated in a four dimensional test space defined by *resource contention* (I/O or CPU bound), *transaction length*, *data contention*, and *deadline distribution*. In addition, we examine the effect of varying h on the performance of the CP scheme. In the following, we first present 4 sets of experiments carried out in an I/O bound system (CPU utilization = 65% and I/O utilization = 95%, on average), and then 1 set of experiments in a CPU bound system (CPU utilization = 92% and I/O utilization = 55%, on average).

4.1 Data contention

In this set of experiments, we vary P_w, the probability of write transactions, so as to vary the level of data contention in the system. Transactions are all equal in length with $x = 6$ steps, the deadline window factor α is fixed at 4, and the threshold parameter h is set at 2 (steps).

Figure 1 plots the *transaction deadline guarantee ratio* versus P_w for policies NRT, WAIT, PI, PA and CP, respectively. As one would expect, the deadline guarantee ratio drops as data contention increases. Among the five policies, CP performs the best, especially when data contention becomes high. PA works very well for low data contention. However, compared to CP, its performance degrades as P_w increases. PI and WAIT perform basically the same with PI being slightly better than WAIT when P_w is small. The deadline guarantee ratio under NRT is the lowest, because it does not make use of any transaction information such as deadline and transaction length.

As we discussed in Section 3, the performance of these schemes is affected by several factors. In the following, we explain the results shown in Figure 1 by analyzing the performance with respect to priority inversion, blocking time, abort rate, and resource utilization.

Figure 2 shows the average number of *priority inversions per transaction run* for WAIT, PI and CP, respectively. PI reduces the number of priority inversions compared to the WAIT scheme. CP achieves much larger reduction in the number of priority inversions by conditionally aborting the lower priority transaction(s). Note that for all the schemes, the average number of priority inversions per transaction run is less than one (even for long transactions). Therefore, *chained blocking* [11] does not happen frequently in RT-CARAT.

Figure 3 demonstrates the situation of *data resource blocking*. As we can see, when priority inversion occurs, the average number of blocking transactions with lower priority is just slightly more than one and basically does not change with P_w.

The average number of *transaction wait instances per run* and the average *waiting time per wait instance* are illustrated in Figures 4 and 5, respectively. As data contention increases, both the average number of wait instances and average waiting time increase. Of the five policies, PA has the lowest number of waits and the shortest waiting time. Note that under PA, priority inversion does not occur (except under the situation where a low priority transaction is in its commit stage). Thus, PA results in the minimal waiting for real-time transactions. From these Figures we also note that priority inheritance, PI, does reduce transaction waiting time with respect to WAIT policy by about 8% for ($P_w = 0.2$). CP further reduces the waiting time over PI, but is still no better than the pure abort scheme PA.

Figure 6 plots *PD abort ratio.* At one extreme, PA has the highest abort ratio. At the other extreme, NRT, WAIT and PI result in the lowest abort ratio. CP falls in-between. This is understandable since PA relies on transaction abort, while NRT, WAIT and PI are based on a wait mechanism. The proposed conditional priority inheritance scheme combines both abort and wait strategies. Hence, the abort ratio under CP is higher than NRT, WAIT and PI, but lower than PA. It is important to mention that there is (almost) no deadlock abort under PA. This is because PA enforces a total ordering by priority among the concurrent transactions (except the situation where a high priority transaction conflicts with a group of transactions, some of which are at the lower priority). For CP, the ratio of priority abort and deadlock abort is approximately 7:1.

To see the negative effect of transaction abort, we plot the *wasted operations per transaction* in Figure 7. Clearly, due to the high abort rate, PA wastes much more work than other schemes. The conditional abort policy CP wastes much fewer operations than PA and nearly the same as NRT, WAIT and PI. Figure 8 shows the *CPU and I/O utilizations*, respectively. As one expects, PA consumes more CPU and I/O resources than any other scheme.

Our observations and discussions in this set of experiments lead to the following points:

- Applying priority inheritance does reduce transaction blocking time. However, PI, the basic priority inheritance scheme, does not provide significant performance improvement over the WAIT scheme. PI performs even worse than WAIT when data contention is high.
- The performances of PA and CP are similar, in terms of transaction deadline guarantee ratio, when data contention is low. CP works better than PA when data contention becomes high.
- Blocking resulting from priority inversion (including the period of priority inheritance) is a more serious problem than wasting system resources. PA and CP, which attempt to eliminate or reduce transaction blocking, perform better than WAIT and PI, which attempt to reduce resource waste.
- Chained blocking and data resource blocking are not frequent and are negligible under PA and CP.

4.2 Sensitivity of threshold (h) settings

The proposed CP scheme employs a *threshold* policy where the decision to use priority inheritance or priority abort depends on h. In the above experiments, h is fixed at 2 (steps). In this set of experiments, we further study the performance of CP by varying the threshold parameter h.

Figures 9, 10 and 11 depict the total waiting time per transaction run, PD abort ratio, and transaction deadline guarantee ratio, respectively, for the workload with $x = 6, P_w = 0.6$, and $\alpha = 4$. To see the relation between CP and PI and PA, we also show the performance of PI and PA, even though they are independent of h. In the figures, CP performs the same as PA when h is equal to 0, and the same as PI when h is equal to 6. This is exactly how CP should behave according to the algorithm described in Section 2. As h changes from 0 to 6, the total waiting time increases while wasted operations decrease. Clearly, due to its threshold policy - switching between priority abort and priority inheritance based on h setting, CP is bounded by PA and PI in terms of total waiting time and wasted operations. With respect to deadline guarantee ratio (see Figure 11), however, CP performs the best for $1 \leq h \leq 3$.

We have also exercised workloads with longer transactions and have observed a similar performance trend to that observed for $x = 6$, with the effective range of h being extended from $1 \leq h \leq 3$ for $x = 6$ to $1 \leq h \leq 6$ for both $x = 12$ and $x = 16$. The performance of CP indicates that, in general, the priority inheritance strategy only works well over a certain range. In other words, the priority inheritance period of a lock-holding transaction cannot be too long (i.e., h should be small with respect to L_T); otherwise, the resulting long blocking time will degrade the performance of other higher priority transactions. We will further explain this result in the following sub-sections.

4.3 Deadline distribution

In this experiment, we examine the schemes along another dimension of our test space, i.e., deadline distribution. We vary the deadline window factor α so as to change the tightness of transaction deadlines.

Figure 12 shows the transaction deadline guarantee ratio for the workload with $x = 6$, $P_w = 0.6$, and $h = 2$. Unlike the results we have shown above where CP performs the best with $\alpha = 4$, here PA becomes the best when transaction deadlines are loose ($\alpha > 4.7$). This is because under PA, a high priority transaction (almost) never waits for a lower priority transaction. With dynamic EDF scheduling policy, a transaction restarted due to conflict abort may get a higher priority, and eventually complete its execution. This is true as long as the deadline of a transaction is long enough to allow it to be (repeatedly) aborted and restarted. Unlike PA, CP may have a high priority transaction wait for lower priority transactions when its conditional priority inheritance is applied. The experimental result implies that when transaction deadlines are loose, it is better for high priority transactions to proceed by aborting lower priority transactions than to wait by applying priority inheritance to lower priority transactions.

Comparing PI and WAIT, one can see the intersection of their performance curves. When transaction deadlines are tight, WAIT performs slightly better than PI. But when deadlines are loose, PI works better than WAIT. This is due to the fact that PI raises the process priority of the lower priority transaction which holds the lock. This operation will increase the actual number of processes executing concurrently in the system. With a higher priority to compete for system resources, the priority inheriting transaction may degrade the performance of other concurrent transactions. This becomes true when transaction deadlines are tight. However, the concurrent transactions can withstand this negative effect when deadlines are loose.

We can see from these results that all of the schemes are sensitive to deadline distribution. Overall, the non-blocking schemes based on priority abort, like PA and CP, perform far better than wait oriented schemes, like WAIT and PI, as long as transaction deadlines are relatively loose.

4.4 Transaction length

All of the results shown above are for transactions with $x = 6$. We now vary transaction length x from 4 steps to 16 steps, with P_w fixed at 0.2, α at 4, and h at 2. Figure 13 plots the transaction deadline guarantee ratio for the five different schemes. For short transactions ($x = 4$), the access conflict rate is low (less than 10%). In this case, the particular scheme used to avoid priority inversion has no significant impact. Since WAIT, PI, PA and CP use the same scheduling policy, namely EDF, their performance is quite close. Although there is little difference between the schemes, their relative order is the same as in the earlier experiments. Here PI is slightly better than WAIT, and PA is slightly better than CP. These are the results that we obtained in the earlier experiments (see Figure 1 for $P_w = 0.2$).

As transaction length increases, on the other hand, the performance difference among the four schemes increases. In addition, the differences between CP and PA and between PI and WAIT are reversed. Now CP performs better than PA, and WAIT performs better than PI. Note that for a fixed P_w value, varying the transaction length also changes the access conflict rate among the concurrent transactions. Hence, we need to isolate the two factors for any further analysis.

To examine the performance for long transactions, we fix transaction length at $x = 16$ while varying P_w from 0.05 to 0.20. Figure 14 shows the deadline guarantee ratio for such workloads. Comparing Figure 14 with Figure 1, we can see that for long transactions, the schemes making use of priority abort, namely PA and CP, perform much better than wait-based schemes PI and WAIT. This is because access conflict over long transactions leads to longer waiting time under PI and WAIT. In other words, *life-time blocking* becomes a severe problem as transactions become long.

Because of the problem of life-time blocking, the priority inheritance scheme does not work well. As one observes from Figure 14, PI performs even worse than WAIT, regardless of the degree of data contention. This is due to the fact that priority inheritance increases the degree of process parallelism. As a result, PI causes a higher deadlock abort rate than WAIT for long transactions (the results are not plotted here). At this point, it is important to compare PI with CP, which also employs a priority inheritance scheme, but on the basis of transaction execution length. Here h is set to 2 (steps) for CP. That means the priority inheritance period under CP is only one eighth of that under PI. Without the life-time blocking problem, CP works well for long transactions, and it outperforms PA as data contention becomes high.

We have also exercised workloads with a mix of different transaction lengths. Here we show one set of the experimental results for such mixed workloads. Figure 15 depicts the average deadline guarantee ratio of two lengths of transactions, one being 4 steps ($x = 4$) and the other 8 steps ($x = 8$) with mean value 6 (i.e., $P[x = 4] = P[x = 8] = 1/2$). Comparing Figure 15 with Figure 12 which shows the performance of transactions with equal length $x = 6$, we can see that the behavior of the five policies and their relative performance under the two different workloads are basically the same. We have also observed similar performance results for transactions with $x = 4$ and with $x = 8$, respectively. In addition, we note that the deadline guarantee ratios of different length transactions are different, with shorter transactions having higher deadline guarantee ratio than longer transactions on average. This phenomenon, called *transaction starvation* [8], relates to the issue of scheduling fairness and is out of the scope of this paper. The reader is referred to [8] for details.

4.5 CPU bound system

As we described at the beginning of this section, the schemes for addressing priority inversion are evaluated in a four-dimensional test space. In the experiments demonstrated above, the performance evaluation was carried out by varying *data contention*, *deadline distribution* and *transaction length*, while fixing the *resource contention* in an I/O bound system. We also examined a CPU bound system, where the CPU and I/O utilizations were 92% and 55%, respectively. Due to the similarity of the experiments, we only illustrate one set of the performance results from such a CPU bound system.

Figure 16 shows the deadline guarantee ratio versus transaction length for the workload with $P_w = 0.2, \alpha = 4$ and $h = 2$. Our first observation is that PA and CP, which are based on priority abort, perform better than WAIT and PI, which are based on a wait mechanism, especially for long transactions. This result is consistent with what we have obtained in an I/O bound system. Secondly, the reader can see that the priority inheritance scheme works well only for short transactions and it performs worse than WAIT for long transactions. This result is also the same as the one we obtained from the I/O bound system.

It is interesting to compare Figure 16, the results obtained from the CPU bound system, with Figure 13, the results from the I/O bound system. We observe that the deadline guarantee ratio of WAIT, PI, PA and CP in the CPU bound system is higher than in the I/O bound system. This is because CPU scheduling plays a more important role in CPU bound systems than in I/O bound systems[6]. (Note that real-time I/O scheduling is not considered in our work.) We also observe that the performance difference between PA/CP and WAIT/PI is larger in the CPU bound system than in the I/O bound system. This is understandable since CPU scheduling will make non-blocking (or less-blocking) conflict resolution schemes work better in a CPU bound system than in a I/O bound system.

These results suggest that in a CPU bound system we may simply incorporate CPU scheduling with a non-blocking resolution scheme, like PA. In other words, it is most important to eliminate priority inversion when CPU contention exists.

5 Conclusions

We have studied several schemes for addressing the priority inversion problem in a real-time database environment where two-phase locking is employed for concurrency control. We first examined two basic schemes, one based on priority inheritance (PI) and the other on priority abort (PA). Based on the analysis, we proposed a combined priority inheritance and priority abort scheme, called conditional priority inheritance (CP). The three schemes, plus two performance baselines NRT and WAIT, have been implemented and evaluated on a real-time database testbed. Our performance studies indicate that with respect to deadline guarantee ratio, the basic priority inheritance scheme does not work well. Rather, the conditional priority inheritance scheme and the priority abort scheme perform well for a wide range of system workloads.

We have clarified through experiments that the priority inheritance scheme is sensitive to the priority inheritance period. A long priority inheritance period (*life-time blocking*) will affect not only the blocked higher priority transactions but also other concurrent non-blocked higher priority transactions. It is the life-time blocking that makes the basic priority inheritance scheme infeasible in resolving priority inversion in real-time database systems. On the other hand, the proposed conditional priority inheritance scheme works well because of its reduced priority inheritance period. From this result, we hypothesize that the basic priority inheritance approach will also not work well in real-time systems where the priority inheritance period is long.

Besides the problem of life-time blocking, it has also been found that the basic priority inheritance scheme has another shortcoming when used in real-time database systems. Applying priority inheritance may increase the degree of actual process concurrency. We have observed that this side effect causes a higher deadlock rate, especially with long transactions.

As a result of our comparison of the abort-oriented scheme, PA and CP, and the wait-oriented scheme, WAIT and PI, we have identified that blocking resulting from priority inversion is a more serious problem than wasting of system resources. This is especially true when transaction deadlines are loose or when a system is CPU bound[3]. In addition, PA is sensitive to data contention, particularly in I/O bound systems. In such an environment, CP, which wastes less resources and yet incurs less blocking (with shorter blocking time), should be used to achieve better performance.

It is assumed in CP that the transaction length is known a priori. In fact, one may only have an estimate of the transaction length. This means that if there is an error in the estimate, performance may be effected. Further experiments may have to be conducted to study how errors in the length estimate would affect the performance.

The use of CP, however, is not limited to applications where the transaction length is known. It may be used, for example, at transaction validation phase in real-time database systems which employ optimistic concurrency control [8]. Since the validation phase, the

[3]These results are similar to what we obtained in our studies on real-time optimistic concurrency control scheme (OCC) [7, 8], where we show that the abort-oriented OCC performs better than wait-oriented two-phase locking approach.

last stage of transaction execution, is usually relatively short, priority inheritance may be applied to the validating transaction which blocks a higher priority transaction.

We may futher extend this work by incorporating real-time I/O scheduling [5, 2, 4, 9]. It would be interesting to re-examine the various schemes studied in this paper in a system that integrates both priority-based CPU scheduling and real-time I/O scheduling.

References

[1] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, Aug. 1988.

[2] Abbott, R. and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.

[3] Chen, M. and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *Real-Time Systems*, Vol.2, No.4, Dec. 1990.

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

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

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

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

[8] Huang, J., J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *A Technical Report, COINS 91-16*, University of Massachusetts, Feb. 1991.

[9] Jauhari, R., "Priority Scheduling in Database Management Systems," *Computer Sciences Technical Report #959*, University of Wisconsin - Madison, Aug. 1990.

[10] Rajkumar, R., "Task Synchronization in Real-Time Systems," *Ph.D. Dissertation*, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, Aug. 1989.

[11] Sha, L., R. Rajkumar and J.P. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report*, Computer Science Dept., Carnegie-Mellon University, 1987.

[12] Sha, L., R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[13] Son, S.H. and C. Chang, "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 1990.