EXTENDING THE PRIORITY CEILING PROTOCOL USING READ/WRITE AFFECTED SETS

BY

MICHAEL A. SQUADRITO

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

1996

ABSTRACT

Concurrency control algorithms that are used in a real-time database must satisfy the timing requirements of the transactions as well as maintain the consistency of the data. Concurrency control techniques vary in the amount of concurrency they allow in a system. As more concurrency is allowed, it is more likely that transactions will meet their timing constraints. In lock-based concurrency control techniques, the problem of deadlock must be addressed. Additionally, if these techniques are used in a real-time system, the problem of *priority inversion* must be addressed.

The priority ceiling protocols prevent deadlock and bound priority inversion in a real-time system. The original protocol was designed to be used with exclusive locking. The read/write priority ceiling protocol was developed later to allow more concurrency in real-time databases that use read/write locking. Since neither protocol uses the semantics of objects, they are not appropriate for use with semantic concurrency control techniques used in real-time object oriented databases. Any gain in concurrency achieved by using semantics would be reduced to the lower concurrency for which that particular protocol was designed.

This thesis presents the *affected set* priority ceiling (ASPC) protocol. This protocol is compatible with semantic concurrency control techniques that support data logical consistency. Proofs will be presented to show that the affected set priority ceiling protocol prevents deadlock and bounds priority inversion in the same manner as the existing priority ceiling protocols.

This thesis also describes how the affected set priority ceiling protocol is implemented in a testbed real-time database system and compared to the existing protocols. The tests indicate that the ASPC protocol performs as well as the existing protocols, and under certain conditions performs better than the existing protocols.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis describes a priority ceiling protocol that uses the semantic information of objects in a real-time database. It also presents proofs that the protocol prevents deadlock and bounds priority inversion. Finally, the thesis describes a prototype implementation on which the tests were conducted, and analyzes the results.

1.1 Motivation

Real-time databases are required for applications that have time constrained data and time constrained transactions, such as automated vehicle control, manufacturing, and air-traffic control [Ram93]. Concurrency control algorithms that are used to control access to the data must satisfy the timing requirements of the transactions as well as maintain the consistency of the data.

Concurrency control techniques vary in the amount of concurrency they allow in a system. As more concurrency is allowed, it is more likely that transactions will meet their timing constraints. In lock-based concurrency control techniques, the problem of deadlock must be addressed. Additionally, if these techniques are used in a real-time system, the problem of *priority inversion* must be addressed. Priority inversion occurs in a real-time system when a low priority transaction prevents a higher priority transaction from executing. Two real-time *priority ceiling* protocols that prevent deadlock and limit priority inversion are presented in [SRL90, SRSC91].

The original priority ceiling protocol [SRL90] was designed to be used with exclusive locking. The read/write priority ceiling protocol [SRSC91] was developed later to allow more concurrency in real-time databases that use read/write locking. Since neither protocol uses the semantics of objects, they are not appropriate for use with semantic concurrency control techniques [BR92, DiP95] used in real-time object oriented

databases. Any gain in concurrency achieved by using semantics would be reduced to the lower concurrency for which that particular protocol was designed.

This thesis presents the *affected set priority ceiling* (ASPC) protocol. This protocol is compatible with semantic concurrency control techniques that support data logical consistency in real-time object-oriented databases. Proofs will be presented to show that the ASPC protocol prevents deadlock and bounds priority inversion in the same manner as the existing priority ceiling protocols.

1.2 Goal of Research

The goal of this research is to develop a priority ceiling protocol for use in an object-oriented database. This protocol will use the semantic information of the objects in the database, thereby providing the potential for more concurrency than either the original or read/write priority ceiling protocols. The protocol developed by this research will also prevent deadlock and bound priority inversion.

1.3 Approach Used

In order to achieve the goal, the existing priority ceiling protocols were examined and evaluated. The protocols consist of a static and dynamic algorithm. The original and read/write priority ceiling protocols use the same dynamic algorithm. However, the static algorithm is dependent on the concurrency control being used, i.e., exclusive locking for the original protocol, and read/write locking for the read/write protocol.

The goal of this research was to alter the static algorithm of the priority ceiling protocol to use semantic locking information. The semantic locking techniques of [BR92, DiP95] were used to modify the static algorithm of the priority ceiling protocol. The dynamic algorithm remains the same as the original and read/write protocols.

1.4 Outline

Chapter 2 is a review of lock-based concurrency control, priority inheritance, deadlock prevention, and the original and read/write priority ceiling protocols. Chapter 3 describes the ASPC protocol and presents the proofs for deadlock prevention and bounded priority inversion. Chapter 4 describes the prototype implementation that was used to evaluate the protocols. Chapter 5 presents the results of the performance tests using simulated

workloads. Chapter 6 compares the protocols, explains the contributions and limitations of this thesis, and discusses future work.

Chapter 2

Related Work

This chapter gives some background on the work that has been done in the area of lock-based concurrency control and deadlock prevention. The chapter then describes priority inheritance, which is a technique used to bound priority inversion in real-time systems. Finally, the original and read/write priority ceiling protocols are explained and illustrated with an example.

2.1 Concurrency Control

In a conventional database, mutual exclusive locking is often used to maintain the consistency of the data. In addition, *two phase* locking is normally used to maintain the *serializability* of the transactions. Two phase locking is a well known technique which requires that a transaction cannot acquire any locks once it has released a lock. Serializability means that transactions will interact concurrently and leave the data in a state equivalent with one of the possible serial executions of the transactions.

There are a variety of concurrency control techniques [BHG86, YWLS82], each using a different level of granularity to resolve conflicts between locks. Figure 2.1 shows a hierarchy of some of the general locking techniques available.

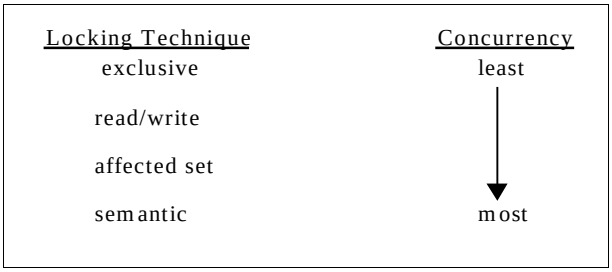

Figure 2.1: Locking and Concurrency

Exclusive locking requires that an entire object be locked no matter what part of an object a transaction may be accessing. Read/write locking is less restrictive because it allows many readers to access an object as long as a transaction is not writing to the object. Since read/write locking potentially allows many readers to access an object, it offers more concurrency than exclusive locking.

Affected Sets. An object is defined as having *attributes* and *methods*. An attribute is a data variable of an object. A method is a function of the object which is used to access attributes contained within the object. The *read affected set* (RAS) of a method contains all attributes of the object that the method reads. The *write affected set* (WAS) of a method contains all attributes of the object that the method writes. Under read/write affected set semantics, two methods m_1 and m_2 are compatible if and only if:

$$(\mathrm{WAS}(m_1) \cap \mathrm{WAS}(m_2) = \emptyset) \wedge$$
$$(\mathrm{WAS}(m_1) \cap \mathrm{RAS}(m_2) = \emptyset) \wedge$$
$$(\mathrm{RAS}(m_1) \cap \mathrm{WAS}(m_2) = \emptyset)$$

Consider the following example to show how the r/w affected sets potentially allow more concurrency than simple read/write locking. An object has two methods that write to two different attributes, i.e., write_speed() and write_depth(), which write to the speed and depth attributes respectively. Using read/write locking, any transaction that uses either method would be granted an exclusive write lock on the object, preventing any other transaction access. Using r/w affected set locking, a transaction executing write_speed() would not prevent any other transaction from executing write_depth() because their affected sets do not intersect, thereby allowing more concurrency than read/write locking.

Semantics. For the past three years our research group at the University of Rhode Island has been performing research in real-time object-oriented databases. This work has included specification of the RTSORAC model [PDPW94] for real-time object-oriented databases, and the specification, implementation, and analysis of an associated semantic locking technique for concurrency control [DiP95]. The semantic locking technique allows the designer of individual objects to determine the allowable level of concurrency within an object, based on the semantics of the object. These semantics may require the relaxation of serializability. A critical issue in the field of real-time databases involves the conflicting requirements of logical and temporal consistency. In order to maintain the logical consistency of the data and/or transactions, transactions may be blocked and miss their deadlines, or they may not be able to write data within the data's timing constraints. On the other hand, by allowing a transaction to preempt a conflicting transaction in order

to write time-constrained data, the logical consistency of the data or of the transactions may be compromised. The semantic locking technique allows the object designer to explicitly express this trade-off between logical and temporal consistency.

The RTSORAC model extends the traditional object-oriented notion of an object to include attributes that have a value, a timestamp, and an amount of imprecision. The imprecision that is recorded accumulates due to the relaxation of serializability by the semantic locking concurrency control technique. RTSORAC objects also include constraints and a compatibility function. The constraints can be placed on the attributes to express logical and temporal correctness of the object.

The user-defined compatibility function determines how the methods of the object may interleave. It is through this function that the object designer expresses the semantics of allowable concurrency. The flexibility of the compatibility function allows the object designer to specify different levels of concurrency for different objects. For instance, one object may require serializability, while another object may tolerate a less restrictive form of correctness. To enforce serializability the object designer may use affected set semantics to determine compatibility.

A less restrictive form of correctness may be needed to express the trade-off between temporal and logical consistency. In such a case, the semantics of compatibility between methods are based on dynamic information, including current temporal consistency and imprecision of data. For example, if a method m_1 that reads an attribute a is currently executing, it would violate the logical consistency of m_1's return value if another method m_2 that writes *a* were to execute. However, if the timing constraint of a has been violated, i.e., it has become old, then allowing m_2 to execute would restore the temporal consistency of *a*. When determining each potential allowable interleaving of method executions, the compatibility function can also examine the amount of imprecision that could be introduced by the possible interleaving.

2.2 Deadlock

In both conventional and real-time databases, transactions compete for access to resources. If a request for a resource is denied, the requesting transaction enters a wait state. Deadlock occurs when transactions are waiting for resources that are held by other waiting transactions [SPG92]. An example of deadlock follows : Consider two transactions, T_1, and T_3, sharing two objects O_A and O_B.

Transaction T_1 locks O_A.

T_3 preempts T_1 and locks O_B.

T_3 is blocked when it attempts to lock O_A.

T_1 resumes, but is blocked when it attempts to lock O_B.

Neither transaction can execute. Deadlock has occurred. Deadlock can happen if and only if four particular conditions occur simultaneously in a system; (1) mutual exclusion, (2) hold and wait, (3) no preemption, and (4) circular wait [SPG92].

(1) Mutual exclusion is required by the database to maintain the consistency of the data and therefore cannot be eliminated.

(2) Hold and wait is a situation when a transaction that currently holds at least one lock is also waiting to acquire additional locks held by other transactions. This situation can be avoided by requiring a transaction to acquire all locks before beginning execution. This solution can significantly lower the concurrency of the transactions since locks are held for a longer period of time than they would otherwise be. More importantly, starvation can occur because a transaction may have to wait indefinitely to acquire a lock that it needs to begin execution [SPG92].

(3) No preemption means that a lock can only be released by the transaction that holds it. This requirement is needed to maintain the consistency of the data.

(4) Circular wait exists when some transaction T_1 is waiting for a lock that is held by T_2, T_2 is waiting for a lock that is held by T_3, T_3 . . ., T_{n-1} is waiting for a lock held by T_n, and T_n is waiting for a lock held by T_1. This problem can be avoided by ordering all locks and requiring all transactions to request locks in that predetermined order. Consider the previous deadlock example with the constraint that locks must be acquired in alphabetical order.

Transaction T_1 locks O_A.

T_3 preempts T_1 and is blocked when it attempts to lock O_A.

T_1 resumes and locks O_B. -- Deadlock is avoided.

There are two problems with this solution. The first is that the lock requests must be ordered. The second is exactly how to pick an order that makes sense while still offering the most concurrency.

As can be seen, of the four conditions necessary for deadlock, eliminating a circular wait seems to be the best way of preventing deadlock in a database that uses two phase locking.

2.3 Priority Inheritance [SRL90]

In a real-time database, transactions must meet timing constraints. To help the transaction scheduler meet these timing constraints, the transactions are typically assigned priorities. One major problem in a real-time system is the priority inversion [SRL90] that occurs when a low priority transaction obtains a lock that blocks a high priority transaction. If nothing is done to address this problem, the inversion can be unbounded, with a high priority transaction waiting for an indefinite amount of time. Unbounded priority inversion makes it impossible to determine the worst-case blocking times, thereby making it impossible to reason about a schedule for the transactions.

As an example, consider three transactions, T_1, T_2, and T_3, in ascending order of priority, and an object O shared by T_1 and T_3.

T_1 locks object O.
T_3 preempts T_1 and begins execution.
T_3 attempts to lock object O and when it is blocked, T_1 resumes execution.

The duration of the blocking is unbounded since T_2 can preempt T_1 any number of times before T_1 can release object O.

Priority inversion can be bounded if the priority of the blocking transaction is raised temporarily to the priority of the blocked transaction during the time that the lock is held. If a low priority transaction T_1 blocks a higher priority transaction T_3, T_1 *inherits* the priority of T_3. When T_1 releases its lock, it acquires its previous priority. Using the example from above:

T_1 has a lock on object O.
T_3 preempts T_1 and begins execution.
T_3 attempts to lock object O and when it is blocked, priority inheritance is used.
T_1 is raised to the priority of T_3, or inherits T_3's priority.
T_2 cannot preempt T_1 since it has a lower priority than T_1's inherited priority.
T_1 releases object O, and its priority is lowered to 1.
T_3 is allowed to preempt T_1 and lock object O.
T_3 executes to completion, at which time T_2 is allowed to run.

Although T_2 is blocked by T_1 with the inherited priority, it is acceptable because it is better than the previous situation when T_3 could be indirectly blocked by T_2 for an indefinite amount of time. Priority inheritance is also transitive [SRL90] . For example,

consider the three transactions T_1, T_2, and T_3. If T_1 blocks T_2, and T_2 blocks T_3, T_1 would inherit the priority of T_3 via T_2.

It has been proven that the *priority inheritance protocol* places an upper bound on inversion [SRL90]. The bound will be the smaller of the following numbers for a given transaction:

1) The number of lower priority transactions.

2) The number of locks used by the transaction.

The priority inheritance protocol still has two problems. The first is that although the inversion is bounded, it can be excessive depending on the number of transactions and locks in a given system. The second problem is that deadlock is not prevented.

2.4 The Priority Ceiling Protocols

The priority ceiling protocols prevent deadlock and bound priority inversion to at most one critical section locked by a lower priority transaction. Proofs to support this claim are presented in [SRL90, SRSC91], and section 3.2 of this thesis. The priority ceiling protocols are based on a major assumption about the system. Every object and every transaction in the system must be known *a priori* in order to gain all of the information needed to execute the protocols. Thus, no dynamic information may be used to determine concurrency control.

There are three basic priority ceiling steps (PC Steps) to any of the priority ceiling protocols, including the one developed in this thesis:

1. Before running, the protocol determines a *priority ceiling* (PC) for each critical section that may be locked. The granularity of these critical sections is the core difference among the various priority ceiling protocols.

2. At run-time, when a transaction T requests a lock, the lock can be granted only if T's priority is strictly higher than the ceiling of locks held by all other transactions.

3. If transaction T_2's lock request is denied because T_1 (of lower priority transaction) holds a lock with a priority ceiling equal to or greater than T_2's priority, T_1 inherits the priority of T_2 until T_1's lock is released.

Note that no checking of conflict is necessary when granting a lock. This is because, conflict in a priority ceiling protocol is captured in the definition of the priority ceiling.

Each of the protocols described follow these PC Steps. The difference among them arises in how conflict is defined among locks and thus, how the priority ceiling is defined. Each priority ceiling protocol will be explained in terms of these three steps using an example to illustrate the potential benefit of using a finer granularity critical section.

2.4.1 The Original Priority Ceiling Protocol

In the original priority ceiling protocol, exclusive locks are placed on entire objects. Thus, the critical section in this version of the protocol is an object lock. PC Step 1 is to determine a priority ceiling for each critical section. The priority ceiling of a lock is defined as the priority of the highest priority transaction that will ever use this lock. This value is assigned using *a priori* information about the transactions and the locks they access.

Consider four transactions, T_1, T_2, T_3, and T_4, where the transaction's subscript indicates its priority (1 = lowest, 4 = highest), sharing two objects O_A and O_B. For this example, the transactions will execute as follows:

T_1 : . . . lock(O_B) . . . lock(O_A) . . . release locks . . .
T_2 : . . . lock(O_A) . . . lock(O_B) . . . release locks . . .
T_3 : . . . lock(O_A) . . . release lock . . .
T_4 : . . . lock(O_A) . . . lock(O_B) . . . release locks . . .

Figure 2.2: Example Original Priority Ceiling Protocol Transaction Definition

Using this information, the priority ceiling of O_A is equal to 4, and the priority ceiling of O_B is equal to 4. A transaction T can lock a critical section only if it passes the following test (PC Step 2):

> *The priority of transaction T must be strictly higher than the priority ceiling of locks held by all other transactions.*

Figure 2.3 is a timeline graph representing one possible concurrent interaction of these transactions:

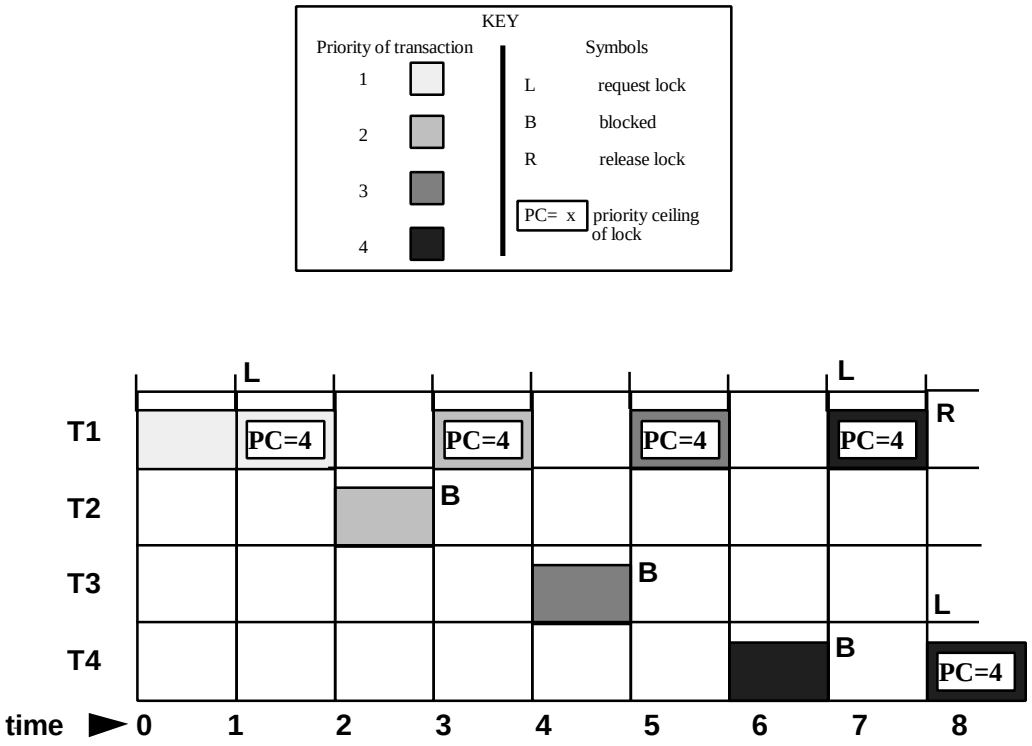

Figure 2.3: Example Original Priority Ceiling Protocol Transaction Timeline

Time	Event	PC Step applied		Explanation
0	Transaction T_1 begins.			
1	Transaction T_1 locks O_B.	(PC Step 2)	--	The lock is granted.
2	T_2 preempts T_1.			
3	T_2 attempts to lock O_A.	(PC Step 2)	--	The priority of T_2 is not greater than the PC of O_B.
	T_2 is blocked.		--	Deadlock is avoided.
	T_1 resumes at priority 2.	(PC Step 3)	--	Priority inversion is limited.
4	T_3 preempts T_1.			

5	T_3 attempts to lock O_A.	(PC Step 2)	--	The priority of T_3 is not greater than the priority ceiling of O_B.
	T_3 is blocked.			
	T_1 resumes at priority 3.	(PC Step 3)	--	Priority inversion is limited.
6	T_4 preempts T_1.			
7	T_4 attempts to lock O_A.	(PC Step 2)	--	The priority of T_4 is not greater than the priority ceiling of O_B.
	T_4 is blocked.			
	T_1 resumes at priority 4.	(PC Step 3)		
	T_1 attempts to lock O_A.	(PC Step 2)	--	The lock is granted since only T_1 holds a lock.

At this point T_1 will continue execution until it releases all of its locks at time 8, and lowers its priority (PC Step 3). At that time T_4 will be allowed to lock object O_A, and complete. Order is maintained as follows: if a transaction T has a priority that is greater than the priority ceilings of all objects currently locked by other transactions, then transaction T is not going to use any of those objects, and a deadlock cannot occur.

The additional benefit of this protocol is a bounded priority inversion of at most one critical section held by a lower priority transaction. Using the example from above, it can be seen that any transaction with a priority less than T_4 attempting to acquire a lock will be blocked by the priority ceiling of the lock held by the lower priority transaction T_1. This means that T_4 will only be blocked as long as T_1 holds the lock on O_B.

One drawback of this protocol for real-time systems is that locking an entire object is very restrictive and can unnecessarily inhibit concurrency that is important to the fast execution that is often needed in real-time databases. As can be seen in the example from above, only one transaction can hold locks at any time.

2.4.2 The Read/Write Priority Ceiling Protocol

In a database that allows select, insert, and update functionality, a division can be made between read and write operations. Instead of acquiring an exclusive lock on an entire

object, a transaction can request read and write locks. Bounding priority inversion and preventing deadlock with read/write locking has been addressed by the *read/write priority ceiling protocol* [SRSC91].

In the r/w priority ceiling protocol, since each object can allow both readers and writers, each object will require two static priority ceilings, and one dynamic priority ceiling that are defined as follows:

The *write priority ceiling* is set equal to the highest priority transaction that will ever write the object.

The *absolute priority ceiling* is set equal to the highest priority transaction that will ever read or write the object.

The *r/w priority ceiling* is set at run-time. If a transaction is allowed to read an object, the r/w priority ceiling is set equal to the write priority ceiling. This prevents any transactions from writing the object, however, this value will possibly allow higher priority transactions to read the object. If a transaction is allowed to write an object, the r/w priority ceiling is set equal to the absolute priority ceiling to prevent all other transactions from reading or writing the object.

Once again, consider the four transactions, T_1, T_2, T_3, and T_4, sharing two objects O_A and O_B. However, we can now use the additional information regarding which transactions are reading and which are writing. For the sake of this example, the transactions will execute as follows:

T_1 : . . . read_lock(O_B) . . . read_lock(O_A) . . . release locks . . .
T_2 : . . . write_lock(O_A) . . . write_lock(O_B) . . . release locks . . .
T_3 : . . . write_lock(O_A) . . . release lock . . .
T_4 : . . . read_lock(O_A) . . . read_lock(O_B) . . . release locks . . .

Figure 2.4: Example Read/Write Priority Ceiling Protocol Transaction Definition

We can now determine the static priority ceilings for each object (PC Step 1). The absolute priority ceiling of object O_A is set equal to the priority of T_4 since it is the highest priority transaction accessing that object. The absolute priority ceiling of object O_B is also set equal to the priority of T_4. The write priority ceiling of object O_A is set equal to T_3 because it is the highest priority transaction that will write O_A. Likewise, the write priority ceiling of object O_B is set equal to T_2. The priority ceilings are shown in Figure 2.5.

Object O_A	
Abs PC	4
Write PC	3

Object O_B	
Abs PC	4
Write PC	2

Figure 2.5: Example Read/Write Priority Ceilings

In the read/write priority ceiling protocol, a critical section is a read/write lock. A transaction T can lock a critical section only if it passes the following test (PC Step 2):

The priority of transaction T must be strictly higher than the r/w priority ceiling of locks held by all other transactions.

Using the same sequence of events as in the previous protocol example:

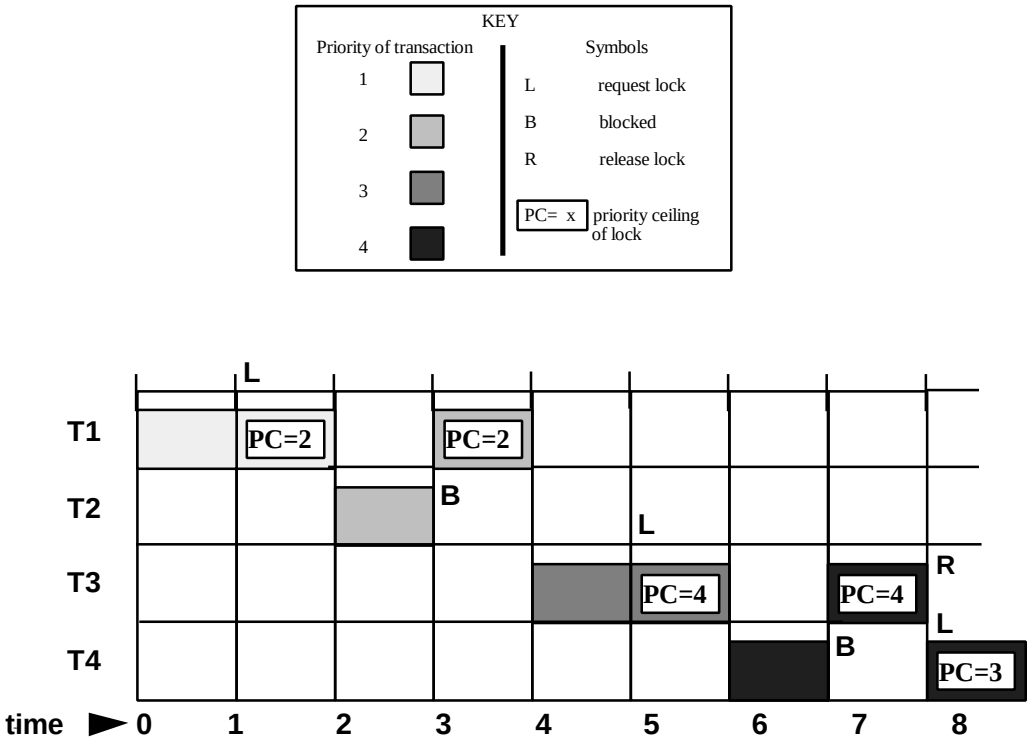

Figure 2.6: Example Read/Write Priority Ceiling Protocol Transaction Timeline

Time	Event	PC Step applied	Explanation/Action

0	Transaction T_1 begins.	(PC Step 2)	--	The lock is granted.
1	Transaction T_1 read_locks O_B		--	r/w PC of O_B = write PC of O_B = 2
2	T_2 preempts T_1.			
3	T_2 attempts to write_lock O_A	(PC Step 2)	--	The priority of T_2 is not greater than the r/w PC of O_B.
	T_2 is blocked and		--	Deadlock is avoided.
	T_1 resumes at priority 2.	(PC Step 3)	--	Priority inversion is limited.
4	T_3 preempts T_1.			
5	T_3 attempts to write_lock O_A	(PC Step 2)	--	The priority of T_3 is greater than the r/w PC of O_B = 2.
	T_3 is granted the write_lock on O_A		--	r/w PC of O_A = absolute PC of O_A = 4

At this point the read/write protocol allowed more concurrency than the original protocol by granting T_3 the lock on O_A. Continuing the example,

6	T_4 preempts T_3.			
7	T_4 attempts to read_lock O_A	(PC Step 2)	--	The priority of T_4 is not greater than the r/w PC of O_A = 4.
	T_4 is blocked.			
	T_3 resumes at priority 4.	(PC Step 3)		
8	T_3 releases the write_lock on O_A and lowers its priority.	(PC Step 3)		
	T_4 is granted the read_lock on O_A		--	r/w PC of O_A = write PC of of O_A = 3

In this example, the read/write locks were executed in a way which allowed more concurrency, while still preventing deadlock. Although this protocol allows more concurrency, it loses effectiveness when individual functions performed on the object can both read and write an object. If each method wrote some piece of information in the object, all locks on the object would be write locks, and this protocol would provide no more concurrency than the previous one.

2.5 Summary of Lock-based Priority Inversion Work

As a review, we can associate each locking technique with the priority ceiling protocol that provides the same level of locking granularity, as shown in Figure 2.7. The original priority ceiling protocol works by placing a single ceiling on an entire object, thereby placing an exclusive lock on that object. The r/w priority ceiling protocol places two ceilings on an object, thus possibly allowing many readers for an object at any given time and only one writer.

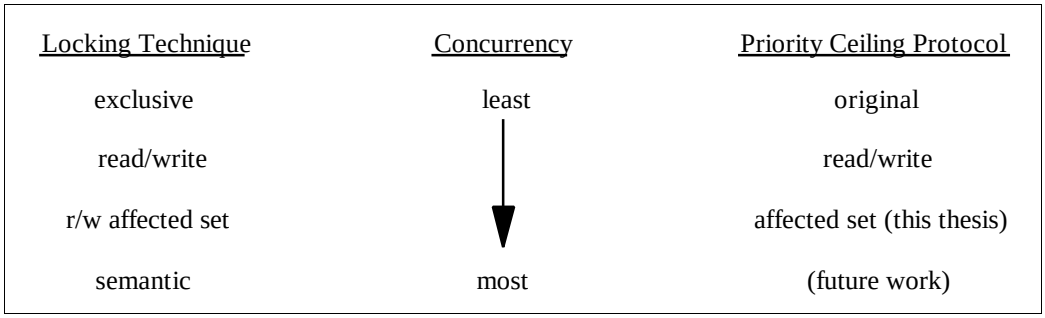

Figure 2.7: Locking and Priority Ceiling Techniques

The existing priority ceiling protocols may be used with affected set locking, however, the concurrency level provided by affected sets will be reduced to that of the priority ceiling protocol being used. The priority ceiling protocol must be redefined in order to capture the semantics of the read/write affected sets.

The research of this thesis has developed the affected set priority ceiling protocol, which uses the static affected set information of an object to achieve a finer granularity of locking than either exclusive or read/write locking. Arbitrary semantic locking is considered to be future work, and is examined in the conclusion of this thesis.

Chapter 3

The Affected Set Priority Ceiling Protocol

The previous priority ceiling protocols place a priority ceiling on an entire data object and therefore allow less potential concurrency than semantic-based techniques, such as that described in Chapter 2, that use locks on methods of database objects.

The ASPC protocol uses the affected sets [BR92] of each method of each object to determine the compatibilities of the methods of an object. The semantic locking technique [DiP95] uses affected set information, but also allows the object designer to specify additional conditions under which methods may execute concurrently. Because priority ceiling protocols are based on static information, establishing priority ceilings where arbitrary semantics are allowed is not straightforward. Thus, the approach in this thesis focuses on affected set semantics.

Using affected set semantics, the critical section is a method lock. Thus, the ASPC protocol assigns a *conflict priority ceiling* to each method of each object. The conflict priority ceiling of a method *m* is the priority of the highest priority transaction that will ever lock a method that is not compatible with method *m* (based on affected set semantics - see Section 2.1).

This protocol requires more information than the previous protocols in order to determine the priority ceilings. Since more information is required about each object, the process of assigning the ceilings is more involved than in the previous protocols. Therefore, PC Step 1 is divided into 4 sub-steps to determine the ceilings for the ASPC protocol:

a. Determine the read/write affected sets for each method.
b. Determine the compatibilities of the methods using the affected sets.
c. Determine the highest priority transaction that will access each method.
d. Determine a priority ceiling for each method using the information from sub-steps b and c.

At run-time the priority ceilings are used the same way as in the original and read/write priority ceiling protocols.

The first section of this chapter illustrates the ASPC protocol by using the same example that was used for the original and read/write protocols. In the second section, the ASPC protocol is proven to prevent deadlock and bound priority inversion of a high priority transaction to at most one critical section (method lock) held by a lower priority transaction.

3.1 The Algorithm by Example

The ASPC protocol will be explained by using the ongoing example in this thesis. Step 1.a is to determine the affected sets for each method of an object. We can define the two objects O_A and O_B, and determine the affected sets:

```
Object O_A :
    Attribute   speed;
    Attribute   altitude;

    method      read_speed();           /* RAS = speed */
    method      write_speed();          /* WAS = speed */
    method      read_altitude();        /* RAS = altitude */
    method      write_altitude();       /* WAS = altitude */

Object O_B :
    Attribute   speed;
    Attribute   depth;

    method      read_speed();           /* RAS = speed */
    method      read_depth();           /* RAS = depth */
    method      write_speed_depth();    /* WAS = speed, depth */
```

Figure 3.1: Example Objects

Transactions are required to access attributes by using the appropriate methods. For simplicity these objects were defined to have distinct read and write methods, however, methods are not restricted to this behavior. Notice that object O_A has separate methods to write each attribute, while O_B has a method that writes to two attributes. Each object is analyzed to determine the read/write affected set for each of its methods, as shown by the RAS and WAS annotations.

Once the read/write affected sets have been determined, PC Step 1.b is to determine the method compatibilities using affected set semantics. The method compatibilities can be evaluated and expressed in a matrix of YES and NO values, indicating whether two methods in an object can or cannot execute concurrently. Figure 3.2 displays the method compatibilities for objects O_A and O_B.

Object O_A				
	read_speed	write_speed	read_altitude	write_altitude
read_speed	YES	NO	YES	YES
write_speed	NO	NO	YES	YES
read_altitude	YES	YES	YES	NO
write_altitude	YES	YES	NO	NO

Object O_B			
	read_speed	read_depth	write_speed_depth
read_speed	YES	YES	NO
read_depth	YES	YES	NO
write_speed_depth	NO	NO	NO

Figure 3.2: Example Compatibility Tables

Notice that two methods may interact concurrently if they are only reading attributes, or if they are accessing different attributes. Methods that write to the same attributes may not execute concurrently.

Now that we have established the method compatibilities, we need to know how a transaction will obtain a method lock. A method lock will be obtained by identifying both the object and method within that object. In an implementation, enumerated types may be used to associate numerical identifiers to each object and method. The next piece of information that is needed is how the transactions are going to interact. Using the ongoing example, we can change the read and write locks to specific method locks (m_lock).

T_1 : ... m_lock(O_B, read_speed) ... m_lock(O_A, read_speed) ... release locks
T_2 : ... m_lock(O_A, write_speed) ... m_lock(O_B, write_speed_depth) ... release locks
T_3 : ... m_lock(O_A, write_speed) ... m_lock(O_A, write_altitude) ... release locks
T_4 : ... m_lock(O_A, read_altitude) ... m_lock(O_B, read_depth) ... release locks

Figure 3.3: Example ASPC Protocol Transaction Definition

Step 1.c is to determine the highest priority transaction that will lock each method. This is done by examining the transaction definitions (Figure 3.3). Step 1.d is to determine the conflict priority ceiling for each method using the definition of the conflict priority ceiling stated at the beginning of this chapter. To see the process more easily, another matrix can be created which will be used in conjunction with the compatibility matrices determined earlier. First, object O_A will be evaluated.

Object O_A				
method ->	read_speed	write_speed	read_altitude	write_altitude
Highest Priority Transaction	T_1	T_3	T_4	T_3
Conflict Priority Ceiling	3	3	3	4

Figure 3.4: Example Affected Set Priority Ceilings for Object O_A

To obtain the conflict priority ceiling of *read_altitude*, identify all methods in the compatibility matrix that conflict with this method. Looking at the *read_altitude* column (or row) in the compatibility matrix of object O_A, we see that only the *write_altitude* method conflicts with the *read_altitude* method. The conflict priority ceiling of *read_altitude* is therefore set to the priority of the highest priority transaction that will use *write_altitude*, which is 3. The conflict ceiling of *write_altitude* is set equal to the priority of the highest priority transaction that will use either *read_altitude* or *write_altitude*, which is 4. The other conflict priority ceilings are set in a similar manner.

We will now repeat this process for object O_B.

Object O_B			
method ->	read_speed	read_depth	write_speed_depth
Highest Priority Transaction	T_1	T_4	T_2
Conflict Priority Ceiling	2	2	4

Figure 3.5: Example Affected Set Priority Ceilings for Object O_B

The highest priority transaction to conflict with *read_speed* and *read_depth* is T_2, since only T_2 uses *write_speed_depth*. Therefore, the conflict priority ceilings of *read_speed* and *read_depth* are set equal to 2. Finally, the conflict priority ceiling of *write_speed_depth* is set equal to 4.

The ASPC protocol allows a transaction T to receive a lock on a critical section if and only if (PC Step 2):

> *The priority of transaction T must be strictly higher than the conflict priority ceiling of locks held by all other transactions.*

Using the same sequence of events as in the previous protocol examples:

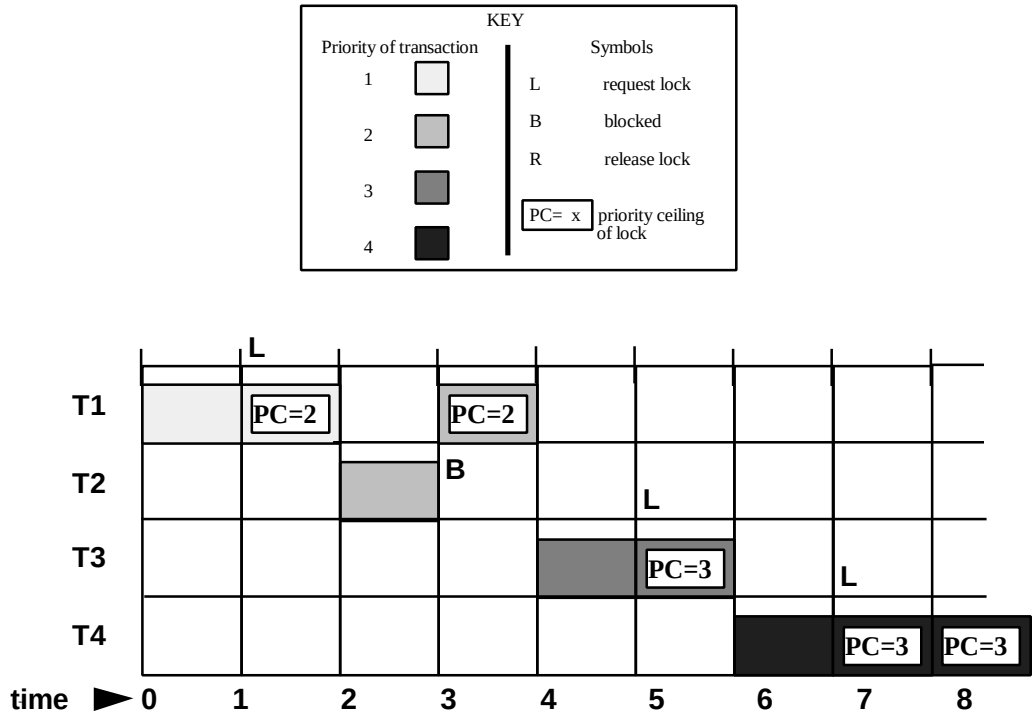

Figure 3.6: Example ASPC Protocol Transaction Timeline

Time	Event	PC Step applied		Explanation
0	Transaction T_1 begins.	(PC Step 2)		
1	T_1 m_lock(O_B, read_speed)	(PC Step 2)	--	The lock is granted.
2	T_2 preempts T_1.			
3	T_2 attempts m_lock(O_A, write_speed)	(PC Step 2)	--	The priority of T_2 is not greater than the conflict PC of (O_B, read_speed) = 2.
	T_2 is blocked.			
	T_1 resumes at priority 2.	(PC Step 3)	--	Deadlock is avoided.
4	T_3 preempts T_1.			
5	T_3 attempts m_lock(O_A, write_speed)	(PC Step 2)	--	The priority of T_3 is greater than the conflict PC of (O_B, read_speed) = 2.
	T_3 is granted the m_lock.			
6	T_4 preempts T_3.			
7	T_4 attempts m_lock(O_A, read_altitude)	(PC Step 2)	--	The priority of T_4 is greater than the conflict PC of (O_B, read_speed) = 2 and (O_A, read_speed) = 3.
	T_4 is granted the m_lock.			

Once T_4 completes, T_3 will resume and eventually release its locks. After T_3 completes, T_1 will resume and after releasing *m_lock(O_B, read_speed)*, will revert back to priority 1 (PC Step 3). This allows T_2 to preempt and run to completion. Finally, T_1 will complete. Note that in this example, the ASPC protocol allows two more locks to be granted than the original protocol and one more than the read/write protocol.

Furthermore, the blocking time for the high priority transaction, T_4, is reduced. In the example of the original protocol, T_4 is blocked as long as T1 holds its locks. With the ASPC protocol, T_4 is not blocked at all.

This example provides the intuition for the effectiveness of the ASPC protocol. It indicates that the finer granularity ceilings can provide more concurrency than the other protocols, and that blocking time for high priority transactions can be reduced. This possible reduction in blocking time is the result of the potentially shorter critical sections (method vs. object locks) of the ASPC protocol.

3.2 Analytical Results

Now that the ASPC protocol has been illustrated, it must be shown that it does in fact prevent deadlock and bound priority inversion to one method lock held by a lower priority transaction. The following proofs were adapted from [SRL90, SRSC91] and have been modified for the ASPC protocol. To prove deadlock prevention, Theorem 1 uses Lemma 1 to prove that a circular wait cannot occur using the ASPC protocol. Since a circular wait is one of the necessary conditions for a deadlock to occur, deadlock is prevented by the ASPC protocol.

Assumption. *A transaction cannot deadlock with itself.*

Lemma 1 *Under the ASPC protocol, each transaction will execute at a higher priority level than the level that the preempted transaction can inherit.*

Proof:

By the definition of the ASPC protocol, when a transaction T locks a set of methods, the highest priority level T can inherit is equal to the highest conflict priority ceiling of the methods locked by T. Hence, when a transaction T_H's priority is higher than the highest conflict priority ceiling of the methods locked by T, the transaction T_H will execute at a priority that is higher than the priority that the preempted transaction T can inherit. QED

Theorem 1 *The ASPC protocol prevents deadlock.*

Proof:

First, by the assumption stated above, a transaction cannot deadlock with itself. Thus, a deadlock can only be formed by a cycle of transactions waiting for each other. Let the *n* transactions involved in the deadlock be $\{T_1, ..., T_n, n > 1\}$. Note that each of these n transactions must hold a lock, since a transaction that does not hold any lock cannot contribute to the deadlock. In order to have more than one transaction hold a lock, a transaction still holding a lock must have been preempted by a higher priority transaction that acquires a lock itself. Suppose that while transaction T_j held a lock, transaction T_i preempts T_j and acquires a lock. By Lemma 1, transaction T_j can never inherit a priority which is higher than or equal to that of transaction T_i, before transaction T_i completes. However, if a deadlock is formed, then by the transitivity property of priority inheritance (PC Step 3) in the ASPC protocol, all the transactions in the deadlock will eventually inherit the highest priority of all the transactions in the deadlock. This is a contradiction of Lemma 1, which states that each transaction will execute at a higher priority level than the level that the preempted transaction can inherit. QED

The following proofs show that the ASPC protocol bounds priority inversion to one method lock held by a lower priority transaction.

Lemma 2 *Under the ASPC protocol, a transaction T_H can be blocked by a lower priority transaction T_L, only if T_L has obtained a lock when T_H arrives.*

Proof:

Consider the two cases when T_H arrives:

Case 1. T_L is running but has not acquired a lock when T_H arrives and attempts to acquire a lock. Applying PC Step 2, T_H is granted the lock and continues since no locks are held by other transactions.

Case 2. T_L is running and has acquired a lock L when T_H arrives and attempts to acquire a lock. Applying PC Step 2, T_H is granted the lock if and only if the priority of T_H is higher than the conflict priority ceiling of lock L.

It follows from PC Step 2 of the priority ceiling protocols that if T_L does not hold a lock, it cannot block a higher priority transaction T_H. QED

Lemma 3 *Under the ASPC protocol, a transaction T that does not hold a lock cannot continue execution until all running and blocked higher priority transactions complete.*

Proof:

Assume that a low priority transaction T is running and does not hold a lock when a series of higher priority transactions $\{T_1, ... T_n, n > 1\}$ begin executing. Since transaction T does not hold a lock, it follows from Lemma 2 that it cannot block a higher priority transaction. Since transactions $\{T_1, ... T_n\}$ have a higher priority than T, T will not be allowed to run until transactions $\{T_1, ... T_n\}$, running and blocked, have completed. QED

Lemma 4 *Under the ASPC protocol, a transaction T can be blocked by at most a single lock of a lower priority transaction T_L.*

Proof:

Suppose that transaction T is blocked by a lower priority transaction T_L which has two or more locks that do not overlap. By Theorem 1, there is no deadlock and transaction T_L will release its current lock at some instant t_1. Once transaction T_L releases its current lock at time t_1, transaction T_L is preempted by T by using PC Step 3. By Lemma 3, T_L cannot acquire another lock until transaction T has completed its execution. By Lemma 2, T_L can no longer block transaction T and it follows that transaction T can be blocked for at most a single lock of a lower priority transaction T_L. QED

Theorem 2 *Under the ASPC protocol, a transaction T can be blocked by at most a single lock of one lower priority transaction.*

Proof:

Suppose that T can be blocked by *n* locks held by lower priority transactions, where $n > 1$. By Lemma 4, T must be blocked by *n* different lower priority transactions. Suppose that the first two lower priority transactions that block T are T_1 and T_2. By Lemma 2, in order for both of these transactions to block T, both of them must hold locks when T becomes ready for execution. Let the lowest priority transaction T_1 obtain its lock first. And let the highest priority ceiling of all locks held by T_1 be P_1. By PC Step 2, in order for transaction T_2 to obtain a lock while T_1 already has one, the priority of T_2 must be higher than the priority ceiling P_1.

Since we assume that transaction T can be blocked by T_1, by PC Step 2, the priority of T cannot be higher than the priority ceiling P_1. Since the priority of T_2 is higher than P_1 and the priority of T is no higher than P_1, transaction T's priority must be lower than the priority of T_2. This contradicts the assumption that the priority of transaction T is higher than both T_1 and T_2. Thus, it is impossible for T to have a priority higher than both T_1 and T_2 and be blocked by both of them under the ASPC protocol. Therefore, under the ASPC protocol, a transaction T can be blocked by at most a single lock of one lower priority transaction. QED

Summary. Notice that the proofs do not rely on how the priority ceilings are determined. The ceilings are used to enforce the semantics of the concurrency control, and at the same time prevent deadlock and bound priority inversion. The results of this section show that the ASPC protocol solves two problems with the semantic locking technique [DiP95]; deadlock and unbounded priority inversion.

Chapter 4

Implementation

The ASPC protocol was designed and implemented as part of a prototype real-time data manager being developed at MITRE. This implementation was used as a testbed for evaluating the ASPC protocol. The entire prototype design developed at MITRE is comprised of many components, including a real-time infrastructure class library, real-time monitoring and control processes, a *data manager*, a real-time database, and a test application. This thesis was concerned with the implementation of the data manager design, which consists of a user interface, query manager, *meta data manager*, *transaction manager*, constraint manager, *object manager*, storage manager, and a persistent database. The data manager in this prototype is responsible for controlling the concurrent access of the objects in the database. This thesis was concerned with the design of the meta data manager, transaction manager, and the object manager. The meta data manager stores and controls access to the meta data for all of the objects and transactions in the database. The transaction manager uses the meta data manager to determine the concurrent interaction of transactions. Transactions are executable code that access the objects in the database. The object manager stores and retrieves objects from the database.

The implementation of the MITRE prototype is currently still underway. The prototype is being developed on 486DX2 66 computers running the Lynx 2.3 operating system. Lynx is a POSIX [Gal95] compliant operating system, having the features required by POSIX.1, POSIX.4, and POSIX.4a standards. The objects and meta data manager are implemented in shared memory. Shared memory is a POSIX.4 feature, and allows multiple processes to access the objects and meta data as if the memory were in their own address space.

The prototype is currently designed for one application running as a single process. This was done so the implementation could use the Lynx priority inheritance mutex and

condition variable, which can only be used within a single process. Within this process, multiple transactions may run and have access to the shared memory objects and meta data. Each transaction is a thread as described in the POSIX.4a standard. A thread can be thought of as a light-weight process, each thread in a process having access to that process's memory.

A process gains access to the shared memory by instantiating a transaction manager in its own address space. The transaction manager class maps in the shared memory. The process and transactions have no direct access to the shared objects or meta data. A transaction must acquire an exclusive, read, write, or method lock. The transaction is given a shadow copy of the attributes in the shared object which are specified by the lock's read/write affected sets. Once the method or methods have finished, the transaction releases the lock, but must commit any writes if the changes are to be reflected in the shared memory object.

4.1 System Description

Two models were used in the overall design and implementation for this thesis. The RTSORAC (**R**eal **T**ime **S**emantic **O**bjects, **R**elationships **A**nd **C**onstraints) model [PDPW94] was used as the basis for the object and transaction descriptions. The ASSET (**A** **S**ystem for **S**upporting **E**xtended **T**ransactions) [BDGJR95] facility was used as the basis for the transaction manager design.

4.1.1 RTSORAC Model

The RTSORAC model was developed at the University of Rhode Island specifically for real-time databases. The model consists of a data manager, a set of object types that describe the structure of database objects, a set of transactions, and a set of relationship types which describe interactions between objects. The MITRE prototype has its own design for a data manager, and, in addition, relationships are currently not addressed in the MITRE design.

Object Types. An object type in the RTSORAC model allows an object to address both logical and temporal constraints of the objects attributes. The MITRE prototype is currently only concerned with logical consistency of data, and therefore, this thesis will use a subset of the RTSORAC object type by excluding the temporal aspects.

For this thesis, an object type is defined by ‹ N, A, M ›. The field N is the name of the object type. The field A is the set of attributes, or data members, each of which has a value. The M field is the set of methods that are used by transactions to access the object's attributes. A method is defined by ‹ Arg, Op ›. Arg is the set of arguments and has the same structure as an attribute. Op is a sequence of programming language operations that represents the executable code of a method.

Transactions. A transaction is defined by ‹ MI, L, C/A, P ›. MI is the set of method invocations a transaction performs. The L field is the set of lock requests and releases. A transaction is required to request a lock before each method invocation on an object in the database. Two-phase locking (2PL) is supported by this model, but it is not a requirement.

The C/A field specifies whether the transaction is to be committed or aborted. This field is not present in the RTSORAC model. A transaction must commit to make writes to the database, otherwise the changes will be lost. A transaction that reads from the database is not required to commit or abort, except to notify other transactions of its status.

The P field represents the priority of the transaction. This priority is used by both the scheduler and the transaction manager. The scheduler will allow the highest priority transaction to execute until it completes or suspends itself. The transaction manager uses the priority in the priority ceiling protocol analysis when a transaction makes a lock request.

4.1.2 ASSET Facility

The transaction manager was modeled after the ASSET [BDGJR95] design. Reference [BDGJR95] specifies that ASSET is not a model, but a flexible transaction facility. This philosophy allows a user of ASSET to create a model for a particular application.

The transaction manager basic primitives in ASSET, that were used in the MITRE design, are as follows: initiate(*func*, *args*), begin(*txnid*), commit(*txnid*), and abort(*txnid*). Initiate initializes meta data structures and reserves system resources for a transaction that will execute the function *func* with arguments *args*. If successful, initiate returns a transaction identifier; otherwise, initiate returns null. Begin actually starts the execution of a transaction's function. If successful, begin returns 1; otherwise begin returns 0. Transactions are atomic, meaning that either all of a transaction's changes are made to the database, or none of them are. Commit must be used by a transaction to make changes to

the database. Commit returns 1 if the specified transaction commits or has committed; otherwise, if that txnid has aborted, commit returns 0. Abort is used to disregard all of the changes that a transaction made. Abort returns 1 if the specified transaction is aborted; otherwise, if the transaction has committed, abort returns 0.

Two additional primitives were added to allow transactions access to the database objects. Request_lock(*txnid*, *obid*, *lock_mode*) is used to obtain access to a database object *obid*. The *lock_mode* parameter refers to the type of concurrency being used, i.e., exclusive, read, write, or affected set. Release_lock(*txnid*, *obid*, *lock_mode*) changes the status of the lock, which may allow another transaction to execute.

4.2 Shared Memory Management

Shared memory enables different processes to share data in a common address space. Objects, object meta data, and structures used by the meta data manager are stored in shared memory. To ease the use of the POSIX shared memory feature, the MITRE prototype used a dynamic shared memory manager developed by John K. Black at the University of Rhode Island. This C++ class allows the dynamic allocation and deallocation of shared memory objects. Each object is retrieved by use of an application assigned object identification number.

Since the priority ceiling protocols require *a priori* knowledge of the objects and transactions, the objects and object meta data structures are statically instantiated in shared memory. An overloaded *new* operator is used for this purpose. In addition, all structures used by the meta data manager are also placed in shared memory at this time. The correct number of meta data manager structures is determined by using the knowledge of the maximum number of objects, transactions, and locks that will be used in the worst case.

When the MITRE project was started, it was not known that static allocation of the shared memory would be sufficient. The dynamic memory manager class is currently only being used during allocation of the shared memory, at which time the meta data manager obtains pointers to all shared objects. No overhead is added while the application is running, and the implementation has the capability to add dynamic objects if they are required at a later date.

4.3 Transaction Implementation

Transactions request locks on objects or the methods of objects. These requests are either granted or denied by using the priority ceiling protocol. Transactions in the prototype are C++ programs that execute as threads. The MITRE prototype requires that all threads be instantiated by using an infrastructure thread class. An infrastructure thread is a C++ class used to encapsulate the operating system calls needed to execute a thread. Use of the infrastructure class ensures that the thread is assigned an appropriate priority, and that the thread can be monitored if required. The infrastructure classes, including threads, mutexes, and semaphores, as well as the priority server, were implemented by Ruth Sigel of MITRE. Each transaction has access to the transaction manager instantiated in the process, and thus has access to the shared memory objects.

4.4 Transaction Manager Implementation

The transaction manager uses the priority ceiling protocol to control the concurrent execution of the transactions. The transaction manager is implemented as a C++ class and contains the primitives described in section 4.1.2 as public member functions. The class has a private meta data manager, which allows the transaction manager access to the shared memory objects. The request_lock and release_lock methods execute the priority ceiling protocols. The explanation of the pseudocode may be found in section 4.6.

4.5 Meta Data Manager Implementation

The meta data manager contains all of the structures required to implement the priority ceiling protocol. The meta data manager is implemented as a C++ class and is based, in part, on the implementation described in [BDGJR95]. The structures and implementation required for the priority ceiling protocols was obtained from [BR89].

4.5.1 Support Structures

The ASSET [BDGJR95] implementation describes three major structures: the Transaction Descriptor (TD), the Lock Request Descriptor (LRD), and the Object Descriptor (OD). Each of these C++ structures required additional fields in order to implement the priority ceiling protocols. The structures are listed below with their major fields, indicating the applicable reference.

TD:

id [BDGJR95]

status [BDGJR95] {initiated, running, committed, aborted}
list of LRDs currently locked [BDGJR95]
base priority [BR89] // the normal (non-inherited) priority for a transaction
blocker's old priority [BR89] // before the blocking low priority transaction inherited the higher priority

LRD:
a pointer to the TD of the transaction that holds this lock [BDGJR95]
a pointer to the OD of the object held by this lock [BDGJR95]
the lock mode {exclusive, read [BDGJR95], write [BDGJR95], method number}
status {granted, pending} [BDGJR95]

OD: // implemented as a C++ templated class
id [BDGJR95]
shared memory object
an array of priority ceilings to support all three protocols [BR89]

Since the priority ceiling protocol is being implemented within the data manager, and not the objects, it made sense to store the meta data required for the priority ceiling protocols within the meta data manager structures and not directly in the object.

4.5.2 The Meta Data Class

The meta data manager has as private members, a hash table for TDs, an array of OD pointers (one for each object in shared memory), a priority queue for granted requested locks (GRL), and a last-in-first-out (LIFO) list for pending transactions (PTL). The user of the prototype initializes constants which enable the meta data manager to allocate the proper number of TD, LRD, and OD structures.

The hash table is used to locate a transaction by its id, which is the same as the thread id assigned by the operating system. The ASSET [BDGJR95] implementation, which was not being used for real-time, suggests the use of a hash table for locating the TDs because of its excellent average case performance. Although a hash table has poor worst case performance, it can be bounded given a certain number of transactions, and it can be easily replaced if the performance is not acceptable. The array of OD pointers is used to locate an object in shared memory. The array indices are the same as the object ids.

The GRL queue and LIFO PTL are described in the implementation of [BR89]. The GRL is a priority queue of currently held locks (LRD structures) enqueued in highest to lowest order based on the priority ceiling of the lock. This queue is used to determine if a transaction may be granted a lock. The LIFO PTL is a list of pending transactions (TD structures). Each time a high priority transaction is blocked by a lock held by a lower priority transaction, the high priority transaction is placed in this LIFO list until the lower priority transaction releases the blocking lock. Use of the GRL and PTL structures will be explained in detail in the next section since they are directly related to the ASPC protocol.

The meta data manager also has a mutex and condition variable which are used to control access to the meta data. The mutex provides mutual exclusion access of the meta data, and the condition variable is used to signal other transactions when a lock has been released.

4.6 Object Type Implementation

The object type in the MITRE prototype is a subset of the RTSORAC object type. The schema is specified by C++ classes that are of the handle/body idiom [Cop92]. The body contains the object's attributes and meta data, and is stored in shared memory. In the conventional handle/body design, the handle contains a pointer to a body object, which is assigned when the handle is instantiated, along with the methods used to access the body. The RTSORAC model uses this idiom, however, the MITRE implementation required atomic transactions. Therefore, the handles are given a shadow copy of the object to enforce atomicity, and a pointer to the object's meta data in shared memory. Currently, the copy is instantiated in the process's own address space. This could be changed so that the copy is instantiated in shared memory by using the dynamic shared memory manager.

Any shared memory object or shadow copy object class that is to be used by the prototype must be derived from a base class. This base class contains fields that are required for all schema classes, such as the number of attributes. Attributes are themselves specified by C++ classes, and are parameterized (C++ templates) so that any C++ basic type can be used. The Attribute class has a private value variable that may be accessed in two ways. The first way is with two public methods, one to read and one to write the actual private value variable. The second way to access the value member is by using the overloaded equal operator for the class.

The meta data is stored in shared memory at a known location for each class. Therefore, each object instantiated for a class shares the meta data, such as the number of methods and attributes, and the read/write affected sets. In addition, the transaction

manager needs to know the offsets of the attributes within the body to transfer values between the shadow copy object and the shared memory object. The user is required to specify a constructor which takes as arguments, pointers to the attributes within the body. When the meta data is placed in shared memory, this constructor is called, and an offset table is initialized that contains each attributes' offset with respect to the object's location. The *get_attr(object pointer, attribute number)* method of the base class is used to convert the relative offset into a pointer to an attribute.

4.6.1 Instantiating an Object

First, the user must specify a schema object in the handle/body form, along with the class meta data (number of attributes, number of methods, and the read/write affected sets). Base classes were implemented for attributes, bodies, and meta data. The handle does not need to be derived at this time. Second, since the priority ceiling protocols need both affected set information and transaction priority information, the user must also specify the highest priority transaction that will access each method of the object.

The class meta data is determined by using a parser, or in the case of the testing analysis, the class meta data was generated automatically. The parser was written using GNU flex (lexical analyzer) and bison (parser), and takes as input the object schema files (both the header and source files). The only requirement is that the attributes be accessed using the read and write member functions of the Attr class. The parser determines the RAS and WAS by identifying the read/write Attr method calls within the object's methods.

The meta data for the objects is placed in shared memory first. When the meta data constructor is called, a temporary object is instantiated, and the attribute offset table is created. The object can now be instantiated in shared memory using the parameterized OD class. The OD class takes as parameters, the type of the body class, and the number of methods in the handle class. The constructor of the OD class takes as arguments, a pointer to the class meta data, and an array of the highest priority transactions that will access the methods. Each array index/location is associated with a method. Presently, the user determines the highest priority transaction that accesses each method. In the future, a parser can be implemented which could analyze the transactions to determine these priorities. The OD constructor computes the priority ceilings as described in section 4.7.1.

4.7 Affected Set Priority Ceiling Protocol Implementation

All three priority ceiling protocols were implemented as part of the data manager using the same functions and meta data structures. Once the implementation was completed for the ASPC protocol, the implementation was able to support all three protocols. Each object sets a conflict priority ceiling for each method, an exclusive priority ceiling, a read priority ceiling, and a write priority ceiling. A transaction then has the flexibility to request an exclusive lock, read lock, write lock, or method lock.

4.7.1 Calculating the Ceilings

The constructor of the OD class uses the read/write affected sets and an array of transaction priorities to calculate the priority ceilings. Since the ceilings are calculated each time an object is instantiated, objects of the same class type may have different ceilings. The method ceilings are calculated by comparing both the RAS and WAS of each method against the RAS and WAS of all methods in the object. The priority ceiling of method *m* is the priority of the highest priority transaction that will ever lock a method that conflicts with method *m*.. The pseudocode for calculating the conflict priority ceilings follows:

```
while  ( n  <  number of methods )
    priority ceiling of method[n]  =  -1
    while  ( m  <  number of methods )
        term1 = method[n].WAS  ∩  ( method[m].RAS  ∪  method[m].WAS )
        term2 = method[n].RAS  ∩  method[m].WAS

        if  ( ( term1 OR term2 )  AND  PC of method[n]  <  hi_prio_txn_array[m] )
            PC of method[n]  =  hi_prio_txn_array[m]

    end inner while.
end outer while.
```

Figure 4.1: Pseudocode for Calculating the Conflict Priority Ceiling

The priority ceilings for the exclusive and read/write protocols are also calculated. The exclusive lock and write lock (absolute priority) ceilings are the same, and are equal to the highest priority in the hi_prio_txn_array. The read lock (write priority) ceiling is determined by using the highest priority in the inputted array that corresponds to a method with a write affected set that is not null:

```
while  ( n  <  number of methods )
    if  ( method[n].WAS  !=  0  AND  PC of method[n]  <  hi_prio_txn_array[n] )
        PC of method[n]  =  hi_prio_txn_array[n]
end while.
```

Figure 4.2: Pseudocode for Calculating the Write Priority Ceiling

The ceilings are stored in a priority ceiling array that has a size equal to the number of methods plus three locations. The first three locations in the array are used for the exclusive, read, and write lock priority ceilings. The remaining locations are used for the method lock conflict priority ceilings, and are accessed using an enumerated type.

4.7.2 Using the Ceilings

The request_lock and release_lock public member functions of the transaction manager class execute the priority ceiling protocols. When a lock is requested, it is either granted or blocked. If the lock is granted, it is placed in the GRL priority queue. If the transaction is blocked by a priority ceiling, the blocked transaction is placed in the PTL in LIFO order and the priority of the blocking transaction is raised to the priority of the blocked transaction. The pseudocode algorithm for requesting a lock is as follows:

```
Lock the mutex.
if  ( the running transaction's id  ==  the GRL transaction's id  OR
      the running transaction's priority  >  the GRL priority ceiling )
    Enqueue the LRD in the GRL queue.          // grant the lock.
else
    Place the running transaction's TD in the PTL.
    Store the blocking transaction's current priority in this TD's blockers_prio field.
    Raise the priority of the blocking transaction.
    Wait on a condition variable.              // suspend until awakened
Unlock the mutex.
```

Figure 4.3: Pseudocode for Requesting a Lock

When a lock is released, the lock is removed from the GRL priority queue. The running transaction then checks to see if it is blocking the first PTL transaction. If it is

blocking, it tests if the first PTL transaction's priority is higher than the priority ceiling of the lock at the front of the GRL queue (PC Step 2). If the PTL transaction can run, the currently running transaction lowers its priority, allowing the blocked transaction to run (PC Step 3). If the PTL transaction cannot run, or there is no PTL transaction waiting, the current transaction continues running. The pseudocode algorithm for releasing a lock is shown in Figure 4.4. Since any given lock may block several higher priority transactions, the blocking transaction must stay in the while loop until all blocked higher priority transactions that can run have been signaled.

```
Lock mutex.
if ( the lock is found in the transaction's TD lock list )
    Dequeue the LRD from the GRL.
    while ( the running transaction's id == the first PTL transaction's blockers_id
          AND the first PTL transaction's priority > the GRL priority ceiling )
        Remove the first TD from the PTL.
        Broadcast the condition variable. // wake up the blocked transaction.
        Lower the running transaction's priority to the priority stored in this TD's
        blockers_prio field.
        Wait on the condition variable.          // suspend until awakened.
    end while.
end if.
Unlock mutex.
```

Figure 4.4: Pseudocode for Releasing a Lock

All three protocols use the same request_lock and release_lock member functions, however, there is a difference between the exclusive priority ceiling protocol and the other two protocols. The exclusive priority ceiling protocol assigns the ceilings such that a transaction will never acquire a lock that has a priority ceiling that is less than the transaction's priority. The difference is that the r/w and ASPC protocols allow a transaction to acquire a lock that has a priority ceiling that is less than its priority. This is due to how the ceilings are assigned in order to reduce conflict.

Since a given transaction will run before transactions with priorities lower than its own, it is only necessary to insert locks into the GRL queue that have a priority ceiling that is greater than or equal to a transaction's base priority. The transaction's priority

prevents lower priority transactions from running, while the priority ceiling prevents higher priority transactions from executing.

Chapter 5

Evaluation

The three priority ceiling protocols were compared using the prototype system described in Chapter 4. Each test involved generating a set of synthetic system configurations and a set of synthetic workloads. On each system configuration, the corresponding workload was executed using each of the priority ceiling protocols. This chapter begins with a description of the construction of the testbed. Next, the performance model and performance parameters are discussed. This chapter then describes the measurements used to compare the protocols and how the testing was performed. Finally, the results are presented and analyzed.

The test model used in this thesis was developed and implemented by Lisa Cingiser DiPippo. Dr. DiPippo used the testbed to evaluate the semantic locking mechanism for her Ph.D. dissertation [DiP95].

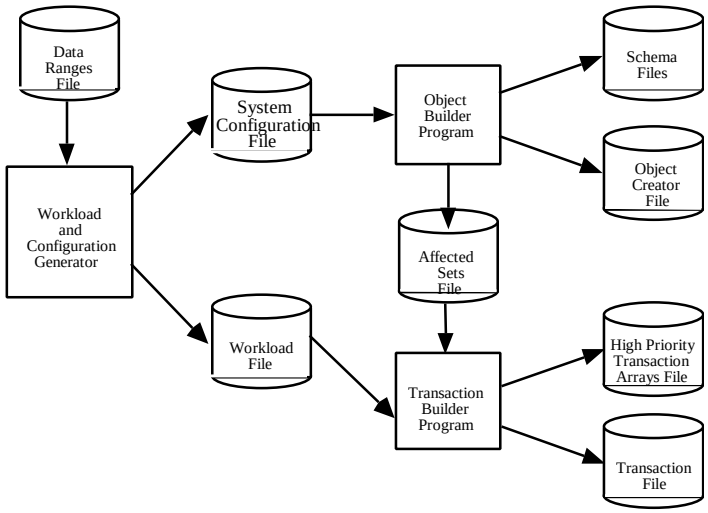

Figure 5.1: Construction of Testbed Configuration

5.1 Testbed Construction

The testbed was implemented by Lisa DiPippo, except where modifications are identified. Figure 5.1 illustrates how a system configuration and workload is generated. The range file stores the data ranges from which the parameters are randomly generated. The workload and configuration generation program reads from the data ranges file and uses a seed value to produce a random number within the specified range for each parameter (see Section 5.3 for performance parameters). The workload and configuration program produces the object parameters in the system configuration file and the transaction parameters in the workload file. The object builder program reads from the system configuration file and produces schema files that contain the C++ specifications of the objects and the objects' meta data in the system. In addition, the object builder program produces a file containing information for storing the objects in shared memory (object creator file). Since transactions require read/write information of the methods for read/write locking, the object builder program was modified to produce an affected set file to be used by the transaction builder program. The transaction builder program reads from the workload file and affected set file and produces a file containing C++ code for the transactions of the workload specification. The transaction builder program was modified to produce a file containing arrays initialized with the highest priority transactions that will access each object's methods.

Once the system configuration and the workload are generated, the test is run using the prototype system described in Chapter 4, with one exception. The prototype system was designed to assign priorities based on the period of a thread (rate monotonic), however, the tests used non-periodic threads. Therefore, the threads were scheduled using least slack time analysis, as was done in [DiP95], and the priorities were set accordingly. Figure 5.2 shows how a test is run. First, the object creator program is compiled including the schema files and the highest priority transaction array file. The object creator program is run, placing the objects of the system configuration into shared memory. Next, the controller program is compiled including the schema files, and the transaction file. The controller process is run and maps the shared memory segment into its own address space. The controller process spawns transactions which execute as threads. The transactions run, accessing objects using the transaction manager and the chosen priority ceiling protocol. The controller process records the results of the test in a statistics file. This procedure was repeated for each test that was performed, changing the range file to vary specific parameters, and changing the seed value to get different random configurations.

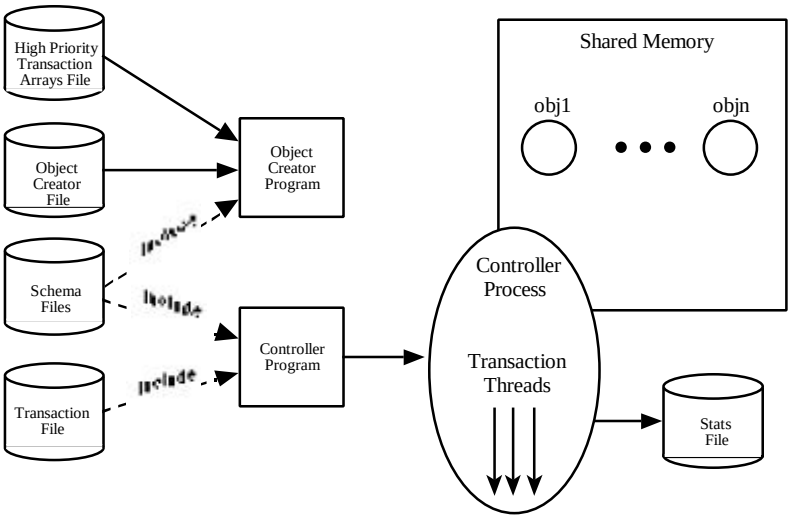

Figure 5.2: Running a Test

5.2 Performance Model

The performance model was taken from [DiP95], and is based on the canonical concurrency control simulation model of [ACL87], with some modifications. Figure 5.3 shows the logical queuing model of [ACL87] which is central to their simulation model for concurrency control algorithm performance.

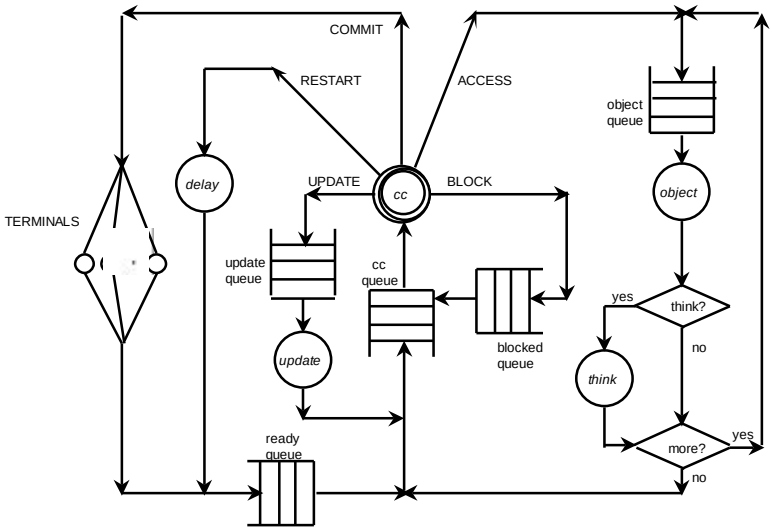

Figure 5.3: Agrawal Performance Model

The terminals in Figure 5.3 represent sources of transactions. When a transaction originates at a terminal and the maximum number of transactions are active, the new transaction enters the ready queue. When the transaction comes off the ready queue, it enters the concurrency control queue (cc queue) and makes its concurrency control requests to the concurrency control module. If the request is granted, the transaction goes to the object queue to access the requested objects, cycling through all of the objects in the request. The transaction returns to the cc queue to make its next request. If a request is denied, the transaction enters the blocked queue where it waits to reenter its request.

The performance model used for this thesis does not conform exactly with the model of [ACL87]. The transactions are started at specified times, and not placed in a ready queue. The range of start times represents the inter arrival time, which is a better measure of system load in real-time databases [DiP95, AGM88, HSTR89] than placing the maximum number of transactions in the system.

In the performance model of this thesis, the cc queue of Figure 5.3 is represented by the GRL priority queue, the blocked queue is represented by the PTL LIFO list, and the cc control module is represented by the priority ceiling protocol. The object queue and the cycle for requests of Figure 5.3 are not used. When a transaction is granted a lock, the transaction executes on an object immediately. The control module of the performance model has three actions, BLOCK, ACCESS, and COMMIT. Transactions request commits through the transaction manager, at which time, changes are transferred to the shared memory objects. Transactions do not restart once they commit.

5.3 Performance Parameters

The parameters of the performance model are based on the parameters of the Agrawal model and are displayed in Table 5.1. Several parameters were not used for testing the priority ceiling protocols. The restart_delay is not used, since transactions are not restarted. The *obj_io* and *num_disks* parameters also were not used since the objects are in shared main memory. Other parameters were not varied, such as the *num_terms*, which is represented by the controller program as the only source of transactions. The number of CPUs (*num_cpus*) is always one. There are four additional parameters that were required during the testing of the priority ceiling protocols. These parameters appear in Table 5.2. Since the testing is for a real-time database, transactions were given deadlines (*trans_dl*). Since the ASPC protocol uses the semantics of an object to determine the conflict priority ceiling, the number of attributes and number of methods, as well as the number of attributes accessed by each method will have a bearing on the eventual value of the

conflict priority ceiling. The more attributes of an object a method accesses, the more likely it will conflict with other methods in the object.

Parameter	Meaning
db_size	number of objects
tran_size	Mean size of transactions
max_size	Size of largest transaction
min_size	Size of smallest transaction
write_prob	Probability that transaction writes object
int_think_time	Mean intratransaction think time
restart_delay	Mean transaction restart delay
num_terms	Number of terminals
mpl	Multiprogramming level
ext_think_time	Mean time between transaction
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
num_cpus	Number of CPUs
num_disks	Number of disks

Table 5.1: Performance Parameters for Agrawal Performance Model

Parameter	Meaning
trans_dl	Transaction deadline
nattrs	Number of attributes per object
nmethods	Number of methods per object
nattrs_method	Number of attributes accessed by each method

Table 5.2: Additional Performance Parameters for Thesis Performance Model

System Configuration. The system configurations that were used in the testing consisted of groups of objects. The configurations were designed in order to vary the data contention between the transactions, thereby testing the performance of the ASPC protocol against the previously existing protocols under various conditions which may or may not benefit a particular protocol.

The number of objects was 10 (*db_size* in Table 5.1), unless otherwise specified. The number of attributes in an object was also 10. The number of methods per object was between 10 and 20, and was picked randomly for each object. The number of attributes per method was 10% of the available attributes in the object, or 20% to 30% of available attributes, in order to vary the data contention for each object. Random attributes were generated for each method. The read/write affected sets are also determined at this time by randomly choosing to read or write, or read and write each attribute touched by a method. The execution time for each method was generated as an integer number of KiloWhetstones [DSW90]. The execution time was converted into seconds and nanoseconds based on the prototype implementation. This execution time is analogous to the *int_think_time* parameter in Table 5.1.

Workload. The tests used 20 transactions accessing a single system configuration. Because the transactions accessed the objects by invoking methods, each workload was dependent on the system configuration. The transaction size (*tran_size*, *max_size*, and *min_size* of the Agrawal model) was varied by randomly choosing the number of methods from a range of 1 to 4, unless otherwise specified. The start time range of the transactions was varied from 5 to 35 seconds, unless otherwise specified. The start time range was used to represent the system load, so the *mpl* parameter of the Agrawal model was not used. The deadlines of the transactions was varied depending on the particular workload. The execution time of a transaction was calculated by adding the execution times of each of the methods that the transaction invoked. The execution time of the transaction manager methods (request_lock, release_lock, commit, and abort) was not considered in the calculation of the transaction execution time. The slack time was calculated by subtracting the execution time from the relative deadline. The priority of the transaction was determined based on a least slack time priority assignment scheme, which has been shown to be optimal under certain circumstances [CSK88].

Each method invocation was generated by first randomly selecting an object from the system configuration, and then randomly selecting a method for the selected object. Each transaction requested locks using two-phase locking. The transaction requested a lock when it was needed, and the transaction held the lock until the end of its execution. Transactions that missed their deadlines were aborted and not restarted.

5.4 Comparison Techniques

The implementation of the meta data manager's OD support structure calculates the priority ceilings for the original, read/write, and ASPC protocols when an object is created. Since the dynamic use of the priority ceilings is the same for each priority ceiling protocol, the same implementation was used to compare the three protocols discussed in this thesis.

When a workload is generated, the transactions are adjusted to use the appropriate lock granularity, i.e., exclusive, read/write, or affected set (method). The transactions were also adjusted so that no unnecessary locking was performed. For example, if a transaction requested 4 methods of an object, it would only request one lock for the original protocol, while 4 method locks were requested for the ASPC protocol. The same analysis was performed for the read/write protocol.

5.5 Performance Measurements

Traditionally the measure of a concurrency control protocol is the throughput of transactions [ACL87]. However, because the priority ceiling protocols are used in real-time systems, it is more important to measure temporal consistency than it is to measure throughput. One way to measure temporal consistency of transactions in a real-time system is through the percentage of transactions that miss their deadlines [HSTR89, AGM88].

5.6 Testing

Each test that was performed generated 15 system configurations and 15 corresponding transaction sets. The results of each test were averaged over these 15 trials producing an error of at most 1% in most cases. A test was conducted for each of the three protocols. The interarrival time of transactions was varied to illustrate how the protocols perform under different system loads. A range of start times was used for a transaction as a measure of interarrival time. The smaller the range of start times for a set of transactions, the closer the interarrival time, and hence the heavier the load.

Test Suite T1: Average Case. These tests were chosen to see how the protocols perform on average with random transactions. The deadlines for transactions were randomly chosen from a range of 1 to 10 seconds for short deadlines, and 4 to 20 seconds for long deadlines. Each test randomly chose 1 to 4 method invocations per transaction, 10 to 20 methods per object, and 1 of the 10 attributes for each method in an object. The

method execution was randomly chosen from a range of 5 to 10 KiloWhetstones for short deadlines and 10 to 15 KiloWhetstones for long deadlines. Both tests used 10 objects.

Having 10 to 20 methods, and one attribute per method may seem low, but only the read/write and ASPC protocols are impacted by these parameters. Having one attribute simulates select, insert, and update calls in a database. Each method randomly chooses between reading (select), writing (insert), or reading and writing (update) an attribute. Since the read/write protocol was designed for use in real-time databases, these parameters should create an unbiased environment in order to compare it with the ASPC protocol.

Test Suite T2: Data Contention. The number of objects, the number of methods, as well as the number of attributes accessed by each method was varied. These parameters contribute to the amount of conflict a transaction will encounter when attempting to request a lock. The number of methods and the number of attributes per method having an impact on the read/write and ASPC protocols, since the ceilings of these protocols depend on the data contention within an object.

Test Suite T3: Priority Inversion. This test was designed to give an indication of the performance of the protocols under a worst case priority inversion situation. The transactions were started in pairs of a low and high priority transactions with a random relative difference in start times of 100, 200, or 300 *ms*. The low priority transaction began execution before the high priority transaction, and was guaranteed to meet its deadline. The difference in the start times within a pair of transactions gave the low priority transaction enough time to acquire a lock. Each pair started after the deadline of the low priority transaction in the previous pair had expired.

5.7 Results

The results show in general that the ASPC protocol performs as well as the previous protocols, with the potential of performing slightly better under certain system loads.

Test Suite T1: Average Case. Taking the error into account, there was no difference between the original and read/write priority ceiling protocols for either short or long deadlines. The ASPC protocol did 2% better (43% compared to 46% with 0.5% error) in the 10 second start range for short deadlines, and 2% better in the 20 second start range, where transactions missed 18% of their deadlines. The ASPC protocol did at

least 3% better than the other protocols for long deadlines in the start ranges of 5, 10, and 15 seconds. Once again, many deadlines were missed in these ranges, 44%, 35.5%, and 27% respectively for each start range.

The numbers indicate that the ASPC protocol allows up to 3% more transactions to meet their deadlines than the other two protocols. These results also indicate that on average, priority inversion is not a problem. However, the important thing to remember is that in a real-time system, worst-case situations are of paramount concern. It doesn't matter that priority inversion doesn't occur often in the average case, the fact that it occurs at all may cause a high priority transaction to miss its deadline and cause a catastrophic failure in the system.

One test that was not performed was when transactions had the same start time. This was proven to be the worst-case situation for independent transactions [Liu73]. In this situation the transactions execute in priority order, highest to lowest, and no blocking occurs. Since the ASPC protocol has the potential of turning a single object lock into as many locks as there are methods in that object, any added execution time will potentially cause a transaction to miss its deadline. A scheduling analysis of the transactions would give a better indication as to the impact of added execution time in both the average random start-time case and the average simultaneous start-time case. This analysis was done for the original priority ceiling protocol [Raj91] and will be discussed in section 6.2.

Test Suite T2: Data Contention. Two test suites were performed to measure the affect of increasing the data contention. The first test used the ranges for the Average Case long deadlines, except that each object had 5 to 10 methods, and each method accessed 2 to 3 attributes. Decreasing the number of methods per object, and increasing the number of attributes touched by each method greatly increased the contention for the data. All three protocols performed equally as well under these conditions. One way to tell if there will be any difference between the protocols is to examine the priority ceilings of the locks. If there is absolutely no difference between the ceilings, then there will be no advantage to using the read/write or ASPC protocols instead of the original protocol. Figure 5.4 shows a sample object from this test and its priority ceilings.

Object O: method conflict priority ceilings

Original PC = 38	M0 PC = 28	M1 PC = 30	M2 PC = 30

Absolute PC = 38	M3 PC = 38	M4 PC = 30	M5 PC = 14
Write PC = 38	M6 PC = 30		

Figure 5.4: Sample Test Object With Priority Ceilings

The priorities used in the test ranged from 2 to 40 (40 being the highest priority). Since the absolute and write priority ceilings are both equal to the original priority ceiling, nothing is gained by using the read/write protocol. The ceilings on the methods show that there is still potential for the ASPC protocol to perform better in certain situations. Once again, just because the average results of this test indicate that there is no difference between the protocols, the ceilings of the methods show that there is still a potential for less priority inversion.

The second test used one object and 4 to 8 method invocations per transaction. The method execution was 2 to 7 KiloWhetstones so that the 4 to 20 second deadlines could still be used. An important measure that this test will indicate is the overhead of locking in the ASPC protocol. When the transactions are using the original protocol, they will only need to request one lock. When the transactions are using the read/write protocol, they may need to request two locks, a read lock followed by a write lock. Since the ASPC protocol requires a lock for each method, transactions may need to request as many as 8 locks.

Examination of the priority ceilings for this test showed that there were some method ceilings lower than the absolute and write priority ceilings. However, all three protocols performed equally as well. This means that requesting extra locks will not have an adverse effect in the average case. Since the protocols are all implemented the same way, replacing one protocol with another will not affect performance in a lock for lock replacement. However, adding extra locks will add some execution time to a transaction. The locks will provide a benefit to the system if they reduce blocking times by lowering the priority ceilings.

In systems with high data contention, for example, in Figure 5.4, if all of the method conflict priority ceilings were 38, there would be no benefit in using the ASPC protocol. Transactions would request more locks, adding execution time, without a decrease in blocking times (lowered ceilings). A scheduling analysis will indicate the situations where a transaction can add extra execution time and still be schedulable.

Test Suite T3: Priority Inversion. Two tests were performed using 15 seeds each. The first test suite consisted of 5 objects, each with 10 methods (1 to 2

KiloWhetstones), each method accessing 1 attribute randomly chosen from the object. Each transaction had 4 method invocations randomly chosen from the objects. The second test consisted of 5 objects, each with 5 to 10 methods (4 to 8 KiloWhetstones), each method accessing 2 to 3 attributes randomly chosen from the object. Each transaction had one method invocation. High priority transactions in both tests had deadlines of 200 to 600 *ms*.

The results were analyzed to determine the case where the protocols did the best. Discarding the high and low data values, the ASPC protocol allowed 19 transactions to meet their deadlines in both tests, where the other protocols only allowed 17. The ASPC protocol allowed the same transactions to finish as the other protocols, plus two additional transactions. The data from these tests was also analyzed using the standard statistical methods for testing the hypothesis:

H_0: original PC protocol missed deadlines <= ASPC protocol missed deadlines
H_1: original PC protocol missed deadlines > ASPC protocol missed deadlines

where H_0 is the original hypothesis, and H_1 is the alternate hypothesis. There is sufficient evidence to reject the original hypothesis within a 2.5% confidence interval. This also applies to the read/write protocol.

This result supports the claim made for the Data Contention test that the conflict priority ceilings of the methods indicated a potential for less priority inversion, even though the protocols performed equally as well on average.

Chapter 6

Conclusion

6.1 Contributions

The goal of this thesis was to develop a priority ceiling protocol that could use the information of affected set semantics. This thesis has presented the ASPC protocol which prevents deadlock and bounds priority inversion, while at the same time preserving the concurrency level of the affected set semantics. It is also an important step towards applying priority ceiling techniques to real-time object-oriented databases. Furthermore, the generality of the ASPC protocol makes it a natural step in extending priority ceiling techniques to control concurrent access to objects.

The test results show that the ASPC protocol can improve average performance of a real-time system by 2% with respect to the original and read/write priority ceiling protocols. In the situation for a worst case priority inversion, the gain in performance can be seen as well.

6.2 Comparison with Related Work

The previous priority ceiling protocols were tested in simulations in [Raj91]. These simulations compared basic priority inheritance (with ordered locks to prevent deadlock) with the original priority ceiling protocol. The simulations also compared other protocols in the priority ceiling family that were not previously mentioned in this thesis. The simulations were performed to analyze the effects of the worst-case blocking time of a task on the actual schedulability of randomly chosen task sets. The tasks were scheduled and then their execution times were increased until the *breakdown utilization* was reached. Breakdown utilization is the point when the system is overloaded and a task will miss its

deadline. The protocol that performs the best is the one with the highest breakdown utilization percentage.

Two experiments were conducted, one using 5 locks (semaphores) and the other using 10 locks. Each transaction executed for a certain duration t_i, acquired a lock and entered a critical section for a time t_j, released the lock, and then executed for an additional time t_k. The tests allowed nested locks, and the locks were ordered for the protocols that did not prevent deadlock. Additionally, the duration of time a transaction spent in each phase (t_i, t_j, and t_k) was equal. Random task sets were generated, and scheduled under the different protocols.

The breakdown utilization for each protocol was averaged over multiple task sets. The results showed that the original priority ceiling protocol performed best. The original priority ceiling protocol had a breakdown utilization of 83.45%, while the basic priority inheritance protocol had a breakdown utilization of 82.19%. This is a difference of 1.26% for average utilization with random phasing (start times). Unfortunately, the read/write protocol was not tested in these simulations, nor was it tested against other priority ceiling protocols in [SRSC91].

The results in [Raj91] indicate that there is not much difference between the protocols in the average case. The simulations in this thesis were not conducted in the same manner because scheduling was not performed, however, with random transactions, and random start times for a given workload, the ASPC protocol performed 2% to 3% better than either the original or read/write protocols. These numbers cannot be compared directly to the numbers obtained in [Raj91], but they are an indication that ASPC protocol is capable of achieving a higher average breakdown utilization for transactions with random start times.

The average breakdown utilization for transactions with simultaneous start times was also determined in [Raj91]. This test showed that the original priority ceiling protocol performed 1.3% better than the basic priority inheritance protocol. The ASPC protocol was not tested for this case. This test will give an indication of the overhead involved when extra method locks are added to a transaction.

Comparing the protocols from an implementation point of view, all three of the protocols tested in this thesis use the same algorithm to request and release locks. The difference in the protocols arises in how the ceilings are determined. As was shown in section 4.7.1, the process of determining the method ceilings is not difficult, assuming the user has access to a parser or tool that can determine the read/write affected sets for each method. The fact that the computation is on the order of $O(n^2)$ is not an issue because the calculations can be done off-line, before the system is started.

6.3 Limitations and Future Work

There are several drawbacks to the ASPC protocol for real-time databases. First, like all priority ceiling protocols, it requires a substantial amount of static, *a priori* information about the system. It is this extra static information that allows priority inversion bounding and deadlock prevention, but it can be a prohibitive assumption for dynamic real-time systems. Second, the ASPC protocol enforces serializability of methods, which may be overly restrictive for real-time databases. Finally, the ASPC protocol does not address temporal consistency nor does it explicitly handle the trade-off between temporal and logical consistency.

Future work will be performing simulations in a manner similar to [Raj91] in order to obtain a more accurate assessment of the ASPC protocol. This will involve a scheduling analysis of the transactions, and a knowledge of the execution times to request, release, commit, and abort a transaction.

One use of the ASPC protocol could be for attribute locking within a method. Recall that for this thesis, attributes were objects themselves with a private value variable accessible by two public methods, one to read and one to write the private value. The conflict priority ceilings could be applied to the read/write methods of the attributes. Methods of the object containing the attributes could request locks on the attributes, thereby further reducing the granularity of the locks. The methods would most likely have to use two phase locking to maintain serializability of the methods, and the number of locks would be substantially increased over method locking described in this thesis. This application of the ASPC protocol could be useful if the execution time of the extra locks was outweighed by the reduction in blocking times (lowered ceilings).

Extending the ASPC protocol further to address any static semantics is also an interesting area of research. Recall that, in general, the priority ceiling protocol will prevent deadlock and bound priority inversion regardless of how the priority ceilings are determined. The ceilings maintain the semantics of the concurrency control technique while preventing deadlock, and priority inheritance bounds the priority inversion.

For example, in lieu of using affected set semantics, a system may want to use security semantics, where there are three levels of security: secret, confidential, and unclassified. For these semantics, a critical section is a security lock. A compatibility matrix can be determined based on the system semantics for the security levels. The highest priority transaction to access an object for each security level is then determined (analogous to determining the highest priority transaction to use each method of an object). The conflict priority ceiling of each security level for an object can then be determined.

The extension of the ASPC protocol to allow the arbitrary semantics of the semantic locking technique is complicated. The fundamental problem is that priority ceiling protocols rely on static knowledge to determine ceilings, while arbitrary semantic conditions, such as current temporal consistency status, are dynamic in nature. An examination of [CL89] may give insights on how to apply dynamic priority ceilings to the semantic locking technique. The protocol described in [CL89] can be used with the dynamic earliest deadline first scheduling algorithm [Raj91].

The ASPC protocol is a compromise that allows more concurrency than previous priority ceiling techniques, and solves the deadlock and priority inversion problems of the semantic locking technique. To bound priority inversion and prevent deadlock in full semantic locking, further research is required. The ASPC protocol is a step towards this future work.

List of References

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609-654, December 1987.

[AGM88] Robert Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *14th VLDB Conference*, August 1988.

[BDGJR95] A. Biliris, S. Dar, N. Gehani, H. Jagadish, K. Ramamritham. ASSET: A System for Supporting Extended Transactions. SIGMOD, 1995.

[BHG86] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, New York, 1986.

[BR89] M. Borger, R. Rajkumar. Implementing Priority Inheritance Algorithms in an Ada Runtime System. Tech. Rep., Software Engineering Institute, CMU, 1989.

[BR92] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163-199, March 1992.

[CL89] K. I. Chen, and K-J Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. Technical Report, UIUCDCS-R89-1511, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[Cop92] J. O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison Wesley, Reading, MA, 1992.

[CSK88] S. Cheng, J. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. In *IEEE Real-Time Systems Symposium*, pages 150-173, 1988.

[DiP95] L. C. DiPippo. Object-based semantic real-time concurrency control. *PhD dissertation University of Rhode Island Department of Computer Science and Statistics*, May, 1995.

[DSW90] Patrick Donohoe, Ruth Shapiro, and Nelson Weiderman. *Hartstone Benchmark User's Guide, Version 1.0.* Carnegie Mellon University, Software Engineering Institute, March 1990.

[Gal95] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

[HSTR89] Jiandong Huang, John Stankovic, D. Towsley, and Krithi Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1992.

[Liu73] C.L. Lui, and J. W. Layland. Scheduling Algorithms for Multi-programming in a Hard Real-Time Enviornment. *Journal of the ACM*, 20(1):46-61, 1973.

[PDPW94] J. Prichard, L.C. DiPippo, J. Peckham, and V. F. Wolfe. RTSORAC: A real-time object-oriented database model. in *The 5th International Conference on Database and Expert Systems Applications*, Sept 1994.

[Raj91] Ragunathan Rajkumar. *Synchronization in Reai-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Boston, MA, 1991.

[Ram93] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

[SPG92] A. Silberschatz, J. Peterson, P. Galvin. *Operating System Concepts*. Addison Wesley, Reading, MA, 1992.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Tech. Rep., Department of Computer Science., CMU, 1987. IEEE *Trans. Comput.*, vol. 39, pp. 1175-1185, Sept. 1990.

[SRSC91] L. Sha, R. Rajkumar, S. Son, and C. Chang. A Real-Time Locking Protocol. IEEE *Trans. Comput.*, vol. 40, pp. 793-800, July 1991.

[YWLS82] Phillip S. Yu, Kun-Lung Wu, Kwei-Jay Lin, and Sang H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140-157, January 1994.

Bibliography

Abbott, Robert, and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *14th VLDB Conference*, August 1988.

Agrawal, Rakesh, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609-654, December 1987.

Badrinath, B. R., and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163-199, March 1992.

Biliris, A., S. Dar, N. Gehani, H. Jagadish, K. Ramamritham. ASSET: A System for Supporting Extended Transactions. SIGMOD, 1995.

Borger, M., R. Rajkumar. Implementing Priority Inheritance Algorithms in an Ada Runtime System. Tech. Rep., Software Engineering Institute, CMU, 1989.

Chen, K. I., and K-J Lin. Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems. Technical Report, UIUCDCS-R89-1511, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989.

Cheng, S., J. Stankovic, and K. Ramamritham. Scheduling algorithms for hard real-time systems - a brief survey. In *IEEE Real-Time Systems Symposium*, pages 150-173, 1988.

Coplien, J. O. *Advanced C++ Programming Styles and Idioms*. Addison Wesley, Reading, MA, 1992.

Dabbene, Danilo, Silverio Damiani. Adding persistence to objects using smart pointers. *The Journal of Object-Oriented Programming*, 8(3):33-39, June 1995.

DiPippo, L. C. Object-based semantic real-time concurrency control. *PhD dissertation University of Rhode Island Department of Computer Science and Statistics*, May, 1995.

Donohoe, Patrick, Ruth Shapiro, and Nelson Weiderman. *Hartstone Benchmark User's Guide, Version 1.0*. Carnegie Mellon University, Software Engineering Institute, March 1990.

Gallmeister, Bill O. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., Sebastopol, CA, 1995.

Huang, Jiandong, John Stankovic, D. Towsley, and Krithi Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of IEEE Real-Time Systems Symposium*, December 1992.

Levine, John R., Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.

Lippman, Stanley B. *C++ Primer*. Addison Wesley, Reading, MA, 1993.

Lui, C. L., and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Enviornment. *Journal of the ACM*, 20(1):46-61, 1973

Prichard, J., L.C. DiPippo, J. Peckham, and V. F. Wolfe. RTSORAC: A real-time object-oriented database model. in *The 5th International Conference on Database and Expert Systems Applications*, Sept 1994.

Rajkumar, Ragunathan. *Synchronization in Reai-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, Boston, MA, 1991.

Ramamritham, Krithi. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

Sha, L., R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Tech. Rep., Department of Computer Science., CMU, 1987. IEEE *Trans. Comput.*, vol. 39, pp. 1175-1185, Sept. 1990.

Sha, L., R. Rajkumar, S. Son, and C. Chang. A Real-Time Locking Protocol. IEEE *Trans. Comput.*, vol. 40, pp. 793-800, July 1991.

Silberschatz, A., J. Peterson, P. Galvin. Operating System Concepts. Addison Wesley, Reading, MA, 1992.

Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison Wesley, Reading, MA, 1994.

Wolfe, V., J. Prichard, L. DiPippo, J. Black, J. Peckham, and P. Fortier. The RTSORAC Real-Time Object-Oriented Database Model and Prototype. Technical Report, University of Rhode Island; a version also published in the *1993 IEEE Real-time Systems Symposium*.

Yu, Phillip S., Kun-Lung Wu, Kwei-Jay Lin, and Sang H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140-157, January 1994.