The Affected Set Priority Ceiling Protocols for Real-Time Object-Oriented Concurrency Control

Michael Squadrito squadrit@cs.uri.edu University of Rhode Island, USA
Levon Esibov esibov@cs.uri.edu University of Rhode Island, USA
Lisa Cingiser DiPippo cingiser@cs.uri.edu University of Rhode Island, USA
Victor Fay Wolfe wolfe@cs.uri.edu University of Rhode Island, USA
Gregory Cooper cooper@cs.uri.edu University of Rhode Island, USA
Bhavani Thurasingham thura@mitre.org MITRE Corporation, Bedford, MA USA
Peter Krupp pck@mitre.org MITRE Corporation, Bedford, MA USA
Michael Milligan miiliganm@hanscom.af.mil Hanscom AFB USA
Russell Johnston russ@nosc.mil U.S. Navy NRaD, San Diego, CA USA
Ramachandra Bethmangalkar bethmang@tripac.com Tri-Pacific Software, Alameda, CA USA

December 15, 1998

Abstract

This paper presents two concurrency control protocols for real-time object-oriented systems. One of the protocols (Affected Set Priority Ceiling Protocol) is designed for single-node systems, and the other (Distributed Affected Set Priorityt Ceiling Protocol) is designed for distributed systems. Both protocols combine features of semantic concurrency control for added concurrency, with priority ceiling techniques for deadlock prevention and bounding priority inversion. This paper demonstrates the properties of increased concurrency, deadlock prevention, and bounded priority inversion of both protocols. It also describes, for each of the new protocols, an implementation that demonstrates its usefulness in real-time object-oriented systems.

1 Introduction

The advent of real-time object-oriented (RTOO) systems, such as Real-Time CORBA (RT CORBA) middleware [11, 7] and RTOO databases [4], poses the need to control concurrent access to objects under real-time requirements. In a real-time database, the concurrency control technique manages concurrent access by transactions to data objects. In a CORBA system, the middleware must control concurrent access by remote clients to CORBA objects.

Concurrency control techniques for RTOO systems must satisfy more requirements than traditional non-real-time concurrency control techniques because they must also meet timing constraints. Among the most important traditional non-real-time requirements are that the technique provides: *high concurrency* to maximize average throughput; *deadlock treatment* that either prevents, avoids, or breaks deadlocks; and *logical consistency*, such as mutual exclusion or serializability, so that all constraints on the values of object attributes are met. In real-time concurrency control, there are similar requirements, along with the requirement that the technique should support *predictable execution*, such as bounded blocking times for locks. Providing predictable blocking times involves, among other things, bounding *priority inversion* that occurs when a lower priority task blocks a higher priority task [8].

In this paper we describe two new techniques for concurrency control in RTOO systems: one for single-node RTOO systems, such as real-time databases; and one for distributed RTOO systems, such as RT CORBA middleware. Both techniques are based on the *priority ceiling* (PCP) family of protocols [8]. Our single-node RTOO concurrency control technique is called the *Affected Set Priority Ceiling Protocol* (ASPCP) [9, 10], and the multi-node technique is called the *Distributed ASPCP* (DASPCP).

Both protocols use PCP techniques while exploiting the semantics of the object-oriented paradigm. They do this through *method-level locking*, where a transaction or client locks a particular method on an object. Method locking is a finer granularity of object lock than *exclusive locking*, where the transaction/client locks the entire object exclusively; or *read/write locking*, in which the transaction/client locks the ability to read or write the entire object. Our protocols are a form of *semantic real-time object-based concurrency control* [4] that uses the semantics of the object to determine the compatibility of method locks. In particular, these two protocols use the semantics of the *affected sets* of methods [2] to determine the compatibility of method locks. The result is a pair of protocols that, like the other PCPs, prevent deadlock and bound priority inversion [8], and do so while allowing more potential concurrency in RTOO systems than other PCPs.

We have designed implementations that use each of the ASPCP protocols. We have implemented a prototype system designed to test the performance of the ASPCP against other single-node PCPs. The prototype places real-time objected-oriented database objects in shared main memory for predictable access of data by transactions. The results of the performance tests are presented in Section 3. We have developed a technique for analyzing the schedulability of RT CORBA systems that use the DASPCP. We have extended the implementation of the PERTS (Prototyping Environment for Real-Time Systems) [5] real-time analysis tool to accept RT CORBA constructs as input, and to analyze the system using the DASPCP.

Section 2 presents the RTOO model that we assume. It also summarizes the original work on single-node and distributed PCP techniques to establish the framework for our techniques. Section 3 presents our ASPCP protocol for single-node RTOO systems, such as real-time object-oriented databases. Section 3 also shows how the ASPCP can lower priority ceilings for objects and therefore increase potential concurrency compared to the original PCP techniques. Furthermore, the section shows that ASPCP still prevents deadlock and tightly bounds priority inversion. Section 3 also describes the prototype in which the ASPCP has been implemented, and the performance tests that compare the ASPCP to other single-node PCP techniques. Section 4 presents the DASPCP for distributed RTOO systems, such as RT CORBA middleware. Like Section 3, this section demonstrates DASPCP's increased potential concurrency and maintenance of deadlock freedom and priority inversion bounds. Section 4 also describes how we have modified the PERTS real-time analysis tool to model and analyze a RT CORBA system under the DASPCP. Section 5 summarizes.

2 Background

Previous work in object-based semantic real-time concurrency control [4] and in priority ceiling protocols [8] has led to our development of the Affected Set Priority Ceiling Protocols. Our previous work in semantic concurrency control [4] indicates that using object semantics to increase concurrency in a real-time database can enhance real-time performance. However, in general, the semantic concurrency control techniques can be complex and do not necessarily bound priority inversion nor prevent deadlock. Fortunately, priority ceiling protocols have been proven to bound priority inversion and prevent deadlock [8] in certain systems. By combining object semantics with priority ceiling techniques, we developed the ASPC protocols presented in this paper.

This section first describes the model of RTOO systems that we assume and also indicates how the model supports semantic real-time concurrency control for object-oriented systems. We then summarize previous work by Rajkumar, Sha, et. al, in developing the priority ceiling protocol and the distributed priority ceiling protocol - both of which provide the framework for our object-based ASPC protocols.

2.1 Real-Time Object-Oriented System Model

A RTOO system consists of *objects*, some of which manage shared resources. The model of a real-time object that we use in this paper is derived from the *RTSORAC* model [4] for real-time object-oriented databases.

Our RTOO system object model extends the traditional object-oriented notion of an object to include attributes that have a value, a timestamp and an amount of accumulated imprecision. The imprecision that is recorded accumulates due to the potential relaxation of serializability by semantic concurrency control [4]. 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* [2] to determine compatibility. A method's *Read Affected Set* (RA) is the set of the object's attributes that the method reads. A method's *Write Affected Set* (WA) is the set of the object's attributes that the method writes. Under affected set semantics, two methods m_1 and m_2 are compatible if and only if:

$$(WA(m_1) \cap WA(m_2) = \emptyset) \wedge (WA(m_1) \cap RA(m_2) = \emptyset) \wedge$$

$$(RA(m_1) \cap WA(m_2) = \emptyset)$$

Note that defining lock compatibility based on these affected set semantics has been proven to produce serializable object schedules [2].

A less restrictive form of correctness may be needed to express the trade-off between temporal and logical consistency. In such cases, 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 on 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.

We developed a semantic locking concurrency control technique [4] that utilizes the full semantics of the compatibility function to express the trade-off between temporal and logical consistency. It has been shown to bound the imprecision that is accumulated due to non-serializable method interleavings [4]. While this semantic locking concurrency control technique provides the potential for increased concurrency for meeting more transaction deadlines, it suffers from unbounded priority inversion and the possibility of deadlock, both of which can affect the system's predictability and its ability to meet timing constraints.

2.2 Priority Ceiling Protocols

A priority ceiling protocol [8] uses information about the way in which transactions intend to use the resources of the system to bound priority inversion and to prevent deadlock. It is based on the assumption about the system that every object and every transaction in the system is known *a priori*. Thus, no dynamic information may be used to determine the semantics of concurrency control.

There are three basic steps to any of the priority ceiling protocols:

1. Before running, the protocol defines a priority ceiling 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's lock request is denied because T_{low} (a lower priority transaction) holds a lock with priority ceiling equal to or greater than T's priority, T_{low} inherits the priority of T until T_{low}'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 from Rajkumar, Sha et al. that we describe below follow these basic steps. The difference among them arises in how conflict is defined among locks and thus, how priority ceiling is defined. We will describe how priority ceiling is defined in each protocol.

The Basic Priority Ceiling Protocol. In the basic priority ceiling protocol (BPCP) [8], exclusive locks are placed on entire objects. Thus, the critical section requires a lock on the entire object. The priority ceiling of a lock is defined as the priority of the highest priority transaction that will ever use this lock. A transaction T can lock a critical section only if it passes the test of Step 2 (above): The priority of transaction T must be strictly higher than the priority ceiling of locks held by all other transactions.

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 [8].

In the Read/Write priority ceiling protocol (R/W PCP), since each object can allow both readers and writers, each object requires two static priority ceilings, and the system dynamically determines which of these two priority ceilings to use as the overall read/write priority ceiling for the object as follows:

1. The *write priority ceiling* is set equal to the highest priority transaction that will ever write the object.
2. The *absolute priority ceiling* is set equal to the highest priority transaction that will ever read or write the object.
3. The *read/write priority ceiling* is set at run-time. If a transaction is allowed to read an object, the read/write priority ceiling is set equal to the write priority ceiling. If a transaction is allowed to write an object, the read/write priority ceiling is set equal to the absolute priority ceiling.

In the R/W PCP, a critical section is a read/write lock. A transaction T can lock a critical section only if it passes the following test:

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

2.3 Distributed Priority Ceiling Protocol.

The Distributed Priority Ceiling Protocol (DPCP) [8] allows tasks to lock objects on remote nodes.

DPCP Terminolgy and Assumptions. An object lock that is accessed by tasks from remote processors is referred to as a *global lock*. If the lock is accessed only by tasks on its node, it is referred to as a *local lock*. A critical section guarded by a global lock is referred to as a *global critical section* (GCS). A critical section guarded by a local lock is referred to as a *local critical section* (LCS). A task T executes its non-critical-section code and LCS's on its host processor. A task's GCS's may be bound and executed on a processor(s) different than the task's host processor. All GCS's that are controlled by the same lock must be bound to the same processor. DPCP prohibits a mixed nesting of LCSs and GCSs, and GCSs at different nodes within a task.

DPCP Priority Ceiling. The *base priority ceiling*, PG, is a fixed priority, greater than or equal to the priority assigned to the highest priority task in the system (in the examples of this paper we will make PG equal to the highest priority of a task in the system). The priority ceiling of a local lock is the highest priority of all tasks that access it. The priority ceiling of a global lock is the highest priority of all tasks that access it plus PG.

DPCP Priority Assignment. A GCS that is generated by task T, is assigned a priority equal to the sum of the base priority ceiling PG and the priority of T.

Priority Ceiling Protocol. Each processor runs the priority ceiling protocol on the LCSs and GCS's by considering each thread of execution for executing a GCS as a "task". While executing a GCS on another node, a task preempts itself on its own node, allowing other tasks to execute.

DPCP Example. The DPCP is a complicated protocol with many cases to consider. The following example shows some of the cases. For a more detailed example of the application of DPCP, we refer reader to Rajkumar's work [8].

Consider a distributed system with two nodes. The application consists of three tasks and two objects (O_{track1} and O_{track2}), guarded by 2 locks (L_1 and L_2). Task T_3 is bound to Node 1, while tasks T_1 and T_4 are bound to Node 2 (we have no task T_2 in this example, we introduce a task T_2 later in the example in Section 3). P_i is the priority of task T_i. In our notation, the higher the Task's subscript, the higher its priority so $P_1 < P_3 < P_4$.

In the example, tasks T_1, T_3 and T_4 execute the following sequence of steps.

```
T1 :  ... O_track2->read_speed ...
T3 :  ... O_track1->write_speed ...
T4 :  ... O_track1->read_altitude...
            O_track2->read_depth
```

Object O_{track1} and its lock L_1 are bound to Node 1. Object O_{track2} and its lock L_2 are bound to Node 2. The priority ceilings of each lock, and the normal execution priority of each critical section thread are listed in the tables of Figure 1.

The following execution sequences demonstrate several aspects of the DPCP including priority inheritance and several forms of blocking of higher priority tasks by lower priority tasks.

- At time t0, task T_1 arrives on Node 2 and begins execution. Similarly, task T_3 begins execution on Node 1.

Priority Ceilings of Locks	
Lock	**PC**
L_1 (Global)	$4+4=8$
L_2 (Local)	4

Normal Execution Priorities of CSs		
Task	**CS Lock**	**Priority**
T_1	L_2	1
T_3	L_1	$3+4=7$
T_4	L_1	$4+4=8$
	L_2	4

Figure 1: Priority Ceilings and Execution Priorities In DPCP Example

- At time t1, task T_1 gets local lock L_2 on Node 2 and begins execution of LCS at its normal execution priority of P_1. Task T_3 gets the global lock L_1 on Node 1 and begins execution of its GCS at its normal execution priority of $P_3 + PG$.

- At time t2, task T_4 arrives on Node 2 and preempts T_1. Task T_3 continues its execution of its GCS on Node 1.

- At time t3, task T_4 requests global lock L_1. Since the priority of T_4's GCS $(4+4=8)$ is not greater than the priority ceiling of the held lock L_1 (8), T_4 is blocked and T_3 continues its GCS execution at the inherited priority of $4+4=8$. Task T_1 resumes its execution of its LCS at Node 2.

- At time t4, task T_3 completes the execution of its GCS, releases global lock L_1, and resumes its own priority. Task T_4 gets global lock L_1 on Node 1 and begins execution of its GCS at its normal execution priority of $4+4=8$. Task T_3 is preempted by the higher priority T_4's GCS. Task T_1 continues the execution of its LCS at Node 2.

- At time t5, task T_4 completes the execution of its GCS and releases global lock L_1. Task T_3 resumes its execution on Node 1. T_4 attempts to get lock L_2. However, the priority of T_4 (4) is not greater than the priority ceiling of the held lock L_2 (4), so T_4 is blocked and T_1 continues its execution with inherited priority of 4.

- At time t6, task T_1 completes the execution of its LCS and releases the lock L_2 and resumes its own assigned priority of 1. Task T_4 gets the local lock L_2 on Node 2 and begins its execution.

- On completion of execution of task T_4 at t9, task T_1 resumes its execution; it and T_3 complete later.

Note the blocking and priority inheritance that occurred at times t3 and t5. Although the DPCP introduces new sources of blocking [8], for each source that was not present in the PCP protocols, Rajkumar has shown that the blocking is finite and that DPCP prevents deadlock [8].

Object O_{track1}				
method	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_{track2}			
method	read_speed	read_depth	write_speed_depth
read_speed	YES	YES	NO
read_depth	YES	YES	NO
write_speed_depth	NO	NO	NO

Figure 2: Affected Set Compatibilities in Example Objects

Summary of Previous PCPs. In this section we have summarized how the BPCP and the DPCP work by placing a single ceiling on an entire object, thereby placing an exclusive lock on that object. The R/W PCP places two ceilings on an object, thus allowing many readers to an object at any given time and limiting access to only one writer. In the next section we describe how we have introduced affected set semantics to improve concurrency in a single-node object-oriented system by placing multiple priority ceilings on each object - one for each method. Section 4 then describes how we do the same in a distributed system.

3 Affected Set Priority Ceiling Protocol.

This section describes the *Affected Set Priority Ceiling Protocol* (ASPCP), which uses the affected sets [2] of each method of an object to determine the compatibilities of the methods of the object, which in turn establishes priority ceilings for each method.

Using affected set semantics, the critical section requires a method lock. Thus, the ASPCP 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; where compatibility is defined by affected set semantics.

In order to determine the priority ceilings used in the ASPCP, the following four sub-steps to Step 1 in Section 2.2 must be performed:

1a Determine the read/write affected sets for each method.

1b Determine the compatibilities of the methods using the affected sets.

1c Determine the highest priority transaction that will access each method.

1d Determine the conflict priority ceiling for each method using the information from Steps 2 and 3.

At run-time, the priority ceilings are used the same way as in the BPCP and the R/W PCP: The ASPCP allows a transaction T to receive a lock on a method if and only if the priority of transaction T is strictly higher than the conflict priority ceiling of locks held by all other transactions.

3.1 ASPCP Example

Consider how the ASPCP works in the following example of a tracking real-time object-oriented database with two data objects O_{track1} and O_{track2}:

```
Object Otrack1 :
  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 Otrack2 :
  Attribute speed;
  Attribute depth;

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

PC Step 1a establishes the read affected sets (RAS) and write affected sets (WAS) of each method, which are also shown with the objects above. For simplicity, these objects were defined to have distinct read and write methods. However, methods are not restricted to this behavior. They can be any user-defined method on the object. Notice that object O_{track1} has separate methods to write each attribute, while O_{track2} has a method that writes to two attributes.

PC Step 1b establishes the method compatibilities using affected set semantics. These method compatibilities are expressed in the table of YES and NO values of Figure 2. Notice in the table that using affected set semantics, two methods may interact concurrently if they are only reading attributes, or if they are accessing different attributes. Also notice that methods that write to the same attributes may not execute concurrently.

Object O_{track1}				
method →	read_speed	read_altitude	write_speed	write_altitude
Highest Priority Transaction	T1	T4	T3	T3
Conflict Priority Ceiling	3	3	3	4
R/W Priority Ceiling	Abs. PC = 4		Write PC = 3	
Original Priority Ceiling	4			

Object O_{track2}			
method →	read_speed	read_depth	write_speed_depth
Highest Priority Transaction	T1	T4	T2
Conflict Priority Ceiling	2	2	4
R/W Priority Ceiling	Abs. PC = 4		Write PC = 2
Original Priority Ceiling	4		

Figure 3: Priority Ceilings in Tracking Example

To establish the conflict priority ceilings, the transactions must be examined. Consider four transactions, T_1, T_2, T_3, and T_4, where the transaction's subscript indicates its priority (1 = lowest, 4 = highest). The transactions share objects O_{track1} and O_{track2} as follows:

```
T1 :  ... O_track2.read_speed ...
            O_track1.read_speed ...
T2 :  ...O_track1.write_speed ...
            O_track2.write_speed_depth ...
T3 :  ...O_track1.write_speed ...
            O_track1.write_altitude ...
T4 :  ... O_track1.read_altitude...
            O_track2.read_depth ...
```

PC Step 1c establishes the highest priority transaction that will invoke each method. PC Step 1d uses this information to determine the conflict priority ceiling for each method. Figure 3 shows the results of PC Steps 1c and 1d for our example. For comparison, it also displays the priority ceilings that would be used by the previous priority ceiling protocols. The determination of the conflict priority ceiling of object O_{track1}'s method *read_altitude* requires identifying all methods in the compatibility table that conflict with it. From Figure 2 we see that only the *write_altitude* method conflicts with *read_altitude*. 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 other conflict priority ceilings are set in a similar way.

Figure 4 shows one possible concurrent execution of the transactions using each of the three PC protocols. In all three executions, at time t0, T_1 starts executing, and at time t1, is granted a lock, since no other transactions currently hold locks. T_2 enters the system at time t2 and preempts T_1 from the CPU. At time t3, T_2 attempts to acquire a lock. In all three cases, T_2 is denied the request since its priority is not greater than the priority ceiling of the lock held by T_1. Note that this prevents a possible deadlock from occurring

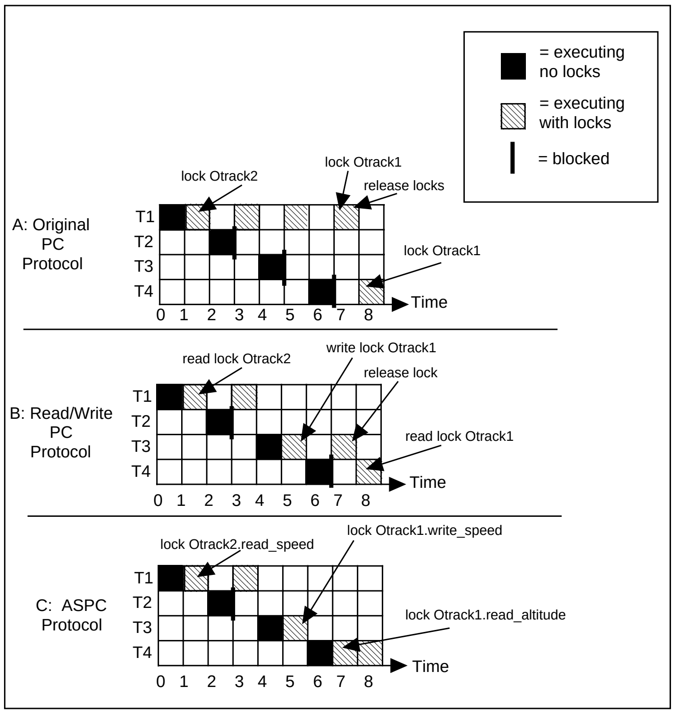


Figure 4: Executions of the Example Transactions Under the Three Priority Ceiling Protocols

between T_1 and T_2. At this point, the three protocols begin to differ in their executions, due to the different ceilings that they use. The BPCP (part A) continues to prevent higher priority transactions from acquiring locks until T_1 releases its locks at time 8. The R/W PCP (part B) and the ASPCP (part C) allow T_3 to acquire its lock at time 5, because T_3's priority is greater than the ceiling of lock held by T_1 (that lock's priority ceiling is 2). When the highest priority transaction, T_4, enters the system and tries to acquire a lock at time 7, it is blocked in the BPCP and in the R/W PCP. On the other hand, the ASPCP allows T_4 to acquire the lock.

3.2 ASPCP Properties

As with the previous priority ceiling protocols, the ASPCP bounds priority inversion, prevents deadlock, and produces serializable schedules of object operations. Here we present informal proofs of these claims, which are based on the analogous proofs of the BPCP. Notice that the proofs do not rely on how the priority ceilings are determined, and so the more formal proofs found in [8] apply to the ASPCP as well.

Deadlock Prevention.

Theorem 3.1 The ASPCP prevents deadlock.

Proof: Informally, our proof of deadlock prevention is based on the fact that the proofs of deadlock prevention for PC protocols in [8] do not rely on how the priority ceilings are determined. Therefore, with minor adjustments for terminology, a similar proof to those given in [8] proves that the ASPCP prevents deadlock.

Basically, since the ASPCP orders the locks, and maintains this order using PC Step 2, a circular wait cannot occur. Since a circular wait is one of the necessary conditions for deadlock in lock-based systems such as we have described, deadlock cannot occur using the ASPCP. The complete proof for deadlock prevention in the ASPCP is given in [9]. □

Bounded Priority Inversion.

Theorem 3.2 Under the ASPCP, a transaction T can be blocked by at most a critical section of one lower priority transaction.

Proof: The proofs of bounded priority inversion in the other PC protocols given in [8] rely on PC Step 2 and PC Step 3 (see Section 2.2), which are common to all priority ceiling protocols, including the ASPCP. Again, since the proofs given in [8] do not rely on how the priority ceilings are determined, this proof is similar to those proofs. The complete proof for priority inversion bound using the ASPCP is given in [9]. □

Increased Potential Concurrency. The example of Figure 4 shows how the ASPCP lowers ceilings, which reduces blocking and increases concurrency.

Theorem 3.3 The ASPCP never decreases concurrency compared to the basic PCP technique.

Proof: The priority ceiling used by the PCP is the maximum of the conflict priority ceilings of the ASPCP. Since lower priority ceilings can only mean less blocking time, concurrency can only increase when ASPCP is used instead of PCP. □

Serializable Execution.

Theorem 3.4 The ASPCP enforces serializable schedules of method operations for each object.

Proof: Under any PCP, a concurrent access is allowed only if the requesting transaction has a priority higher than the priority ceiling of all held locks. In the ASPCP the priority ceiling of a lock is determined by the priority of transactions accessing conflicting locks. Thus, a lock will not be granted if a conflicting lock is currently held. Badrinath and Ramamritham showed that by defining conflict using affected set semantics, an object is ensured a serializable schedule of method operations [2]. Thus, since ASPCP defines conflict with affected set semantics and denies conflicting locks, it produces a serialable schedule of method operations on each object. □

Note that the above theorem discusses serializability of method operations within an object, and not the more global notion of transaction serializability. Two-phase locking of method locks can be used to ensure transaction serializability.

3.3 Implementation

The Prototype System. The ASPCP was designed and implemented as part of a prototype real-time data manager developed at MITRE [9]. This implementation was used as a testbed for evaluating the ASPCP. The data manager design includes a meta data manager, a transaction manager, and an object manager. The data manager in this prototype is responsible for controlling the concurrent access of the objects in the database. 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, which was modeled after the ASSET [3] design, uses the meta data manager to determine the concurrent interaction of transactions. The object manager stores and retrieves objects from the database. The prototype was developed on 486DX2 66 computers running the Lynx 2.3 operating system.

Objects and the meta data manager are implemented in shared memory. Since the ASPCP requires *a priori* knowledge of the objects and transactions, the objects and object meta data structures are statically instantiated in shared memory. In addition, all structures used by the meta data manager are placed in shared memory at this time.

Objects in the system are C++ classes derived from a base class that contains fields required for all objects, such as the number of attributes and number of methods. The objects attributes and meta data are stored in shared memory. Its methods are compiled into the transactions that accesses it.

Transactions in the prototype are C++ programs that execute as threads. Before executing a method, a transaction must request a method lock. The request is either granted or denied based on the execution of the ASPCP. If the request is granted, the transaction is given a shadow copy of the attributes in the shared object that are specified by the requested methods 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. Each transaction has access to the transaction manager instantiated in the process, and thus has access to the shared memory objects.

The meta data manager has as private members, a hash table for transaction descriptors (TDs), an array of object descriptor (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 meta data manager also has a mutex and condition variable which are used to control access to the meta data.

The transaction manager, implemented as a C++ class, uses the priority ceiling protocol to control the concurrent execution of the transactions. 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 of the transaction manager execute the affected set priority ceiling protocol.

ASPCP Implementation. The affected set priority ceiling protocol was implemented as part of the data manager. Each object sets a conflict priority ceiling for each method. The constructor of the OD class uses the read/write affected sets and an array of priorities of transactions that call the objects methods to calculate the priority ceilings. The method conflict priority 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 ceilings are stored in a priority ceiling array in the object's meta data. The pseudocode for calculating the conflict priority ceilings is shown in Figure 5.

The *request_lock* and *release_lock* public member functions of the transaction manager class execute the

```
while  ( n  <  number of methods )
  priority ceiling of method[n]  =  -1
  while  ( m  <  number of methods )
    term1 = method[n].WAS  intersect ( method[m].RAS  union method[m].WAS )
    term2 = method[n].RAS  intersect 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 5: Pseudocode for Calculating the Conflict Priority Ceiling

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

Figure 6: Pseudocode for Requesting a Lock

affected set priority ceiling protocol. 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 for requesting a lock is shown in Figure 6.

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 for releasing a lock is shown in Figure 7. 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 ((running trans's id == first PTL trans's blockers_id) AND
        (first PTL trans's prio > GRL prio ceiling))
    Remove the first TD from the PTL.
    Lower running trans's prio to prio in this TD's blockers_prio field.
    Broadcast the condition variable   // wake up the blocked transaction.
    Wait on the condition variable     // suspend until awakened.
  end while
end if
Unlock mutex
```

Figure 7: Pseudocode for Releasing a Lock

Parameter	**Value(s)**
Num Objects	10
Num Attribs	10
Num Methods	10-20
Method Exec Time	5 to 15 KWhetstones
Attribs/Method	1
Num Trans	20
Methods/Trans	1-4
Trans Deadline	1-20 sec
Trans Start Time	5-35 sec

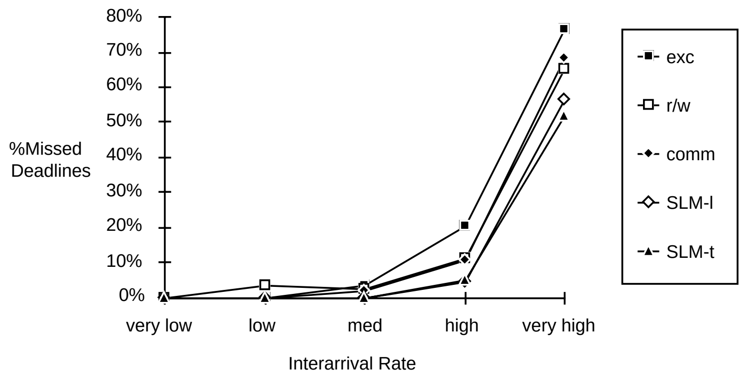
Table 1: Priority Ceiling Performance Parameters

3.4 ASPCP Performance Testing

The ASPCP was compared with the BPCP and the R/W PCP using the prototype system described in Section 3.3, slightly modified to implement the BPCP and the R/W PCP. 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.

3.4.1 Performance Parameters.

Table 1 lists the parameters and values used in the testing. Transactions that missed their deadlines were aborted and not restarted. As is typical in real-time concurrency control performance evaluation, percentage of missed deadlines was used to measure performance [6, 1]. Transactions were started in several different size start time intervals in order to illustrate how the protocols performed under varying system loads.



Interarrival Rate	Start Time Range
very low	4 - 35 sec
low	4 - 30 sec
med	4 - 25 sec
high	4 - 15 sec
very high	4 - 5 sec

Figure 8: Affected Set Performance: Effect of Deadline

3.4.2 Results.

The results of the tests produced an error of at most 1% with a 95% confidence interval. We performed one test suite to examine results when deadline was varied. Based on the results of these tests, we performed a specialized test to indicate the performance of the protocols when priority inversion was likely to occur.

Effect of Deadline. This test suite examined performance with long deadlines (4-20 sec) and with short deadlines (1-10 sec). Figure 8 displays the results of the short deadline test. The long deadline test produced similar results. We can see that the two previous priority ceiling protocols performed very similarly. The ASPCP performed better than the other two, with the difference becoming more pronounced under heavier system load. This result is due to the fact that under high system load, the added concurrency provided by the ASPCP allows more deadlines to be met. When system load is lower, all three protocols are more likely to meet more deadlines.

Priority Inversion. While performing the previous test suite, we found that priority inversion occurred very rarely. However, in a real-time system, if priority inversion occurs at all a high priority transaction may miss its deadline and cause a catastrophic failure in the system. Therefore, we performed the following test suite in order to examine how performance was affected when priority inversion was likely to occur. Because these tests were constructed for a particular purpose, some of the parameters are significantly different from the values displayed in Table 1.

In this test suite, two tests were performed, both to represent high data contention in different ways. Table 2 displays the parameters that were used for this test. The transactions were started in pairs of low and high priority transactions. 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 and to provide the opportunity for priority inversion to occur.

	Test 1	Test 2
Num Objects	5	5
Trans Deadline	100-600ms	100-600ms
Methods/Obj	10	5-10
Method Exec	1-2 KWh	4-8
Attribs/Method	1	2-3
Methods/Trans	4	1

Table 2: Priority Inversion Test Suite Parameter Values

The ASPCP allowed 19 transactions to meet their deadlines in both tests, where the other protocols only allowed 17. This result is due to the lower occurrance of priority inversion with the ASPCP. The increased concurrency provided by the ASPCP allowed it to avoid priority inversion in cases when the other priority ceiling protocols could not.

Overall Results. The results of these tests indicate that on average, the ASPCP misses slightly fewer deadlines than the other priority ceiling protocols. The results also show that the ASPCP avoids priority inversion more than the other protocols. While these results may not seem highly significant, it is important to note that priority ceiling protocols are typically used in systems with hard real-time requirements where performance must be guaranteed. If an *a priori* analysis is done on such a system, whenever priority inversion can be avoided, the analysis will be simpler, and a guarantee of meeting timing constraints will be more likely.

4 Distributed Affected Set Priority Ceiling Protocol

For concurrency control in distributed real-time object-oriented systems, we have developed the DASPCP that combines techniques from Rajkumar's DPCP, and our ASPCP. In particular, just as the ASPCP uses the PCP mechanism but with the lock granularity at the object method level, the DASPCP uses the DPCP mechanism with its lock granularity at the object method level.

A global critical section in the DASPCP is an object method that is accessed by one or more clients on non-local nodes. We will refer to this method as a global method. The definition of priority ceiling for a global method in the DASPCP is a combination of the PC definition in the ASPCP and in the DPCP. The priority ceiling of a global method m is the sum of the base priority ceiling, PG, and the highest priority of a transaction that will ever lock a method that is not compatible with method m; where compatibility is defined by affected set semantics. The DASPCP also uses the DPCP priority assignment so that global methods execute at the priority of the requesting task plus PG. Note that these priority ceilings cause the reduced blocking found in the DASPCP compared to the DPCP. The DASPCP also follows the basic steps

Object O_{track1}				
method $\rightarrow$	read_speed	read_altitude	write_speed	write_altitude
Highest Priority Transaction	T1	T4	T3	T3
DASPCP Priority Ceiling	$3+4=7$	3	$3+4=7$	$4+4=8$
DPCP Priority Ceiling	$4+4=8$			

Object O_{track2}			
method $\rightarrow$	read_speed	read_depth	write_speed_depth
Highest Priority Transaction	T1	T4	T2
DASPCP Priority Ceiling	2	2	4
DPCP Priority Ceiling	4		

Normal Execution Priorities of Methods		
Task	**Method**	**Priority**
T_1	$Otrack1 \rightarrow read\_speed$	$1+4=5$
	$Otrack2 \rightarrow read\_speed$	1
T_3	$Otrack1 \rightarrow write\_speed$	$3+4=7$
	$Otrack1 \rightarrow write\_altitude$	3
T_4	$Otrack1 \rightarrow read\_altitude$	$4+4=8$
	$Otrack2 \rightarrow read\_depth$	4

Figure 9: Priority Ceilings and Execution Priorities in Distributed Tracking Example

of all priority ceiling protocols (see Section 2.2).

4.1 DASPCP Example

To illustrate the DASPCP, consider the example that was introduced in Section 2.3 then augmented in Section 3.1. The DASPCP priority ceilings are shown in Figure 9. In this figure, the priority ceilings of object O_{track1}, which is a globally accessed since it resides on Node 1 and is accessed by T_1 and T_4 on Node 2, are shown as the sum of the highest priority of a task that accesses a conflicting method plus the base priority ceiling, PG (we have chosen PG=4, the highest priority of any task in the system).

Consider the following execution sequence.

- At time t0, task T_1 arrives on Node 2 and begins its execution. Similarly, task T_3 begins execution on Node 1.
- At time t1, task T_1 gets local lock on $O_{track2} \rightarrow$ *read_speed* on Node 2 and begins execution of its LCS at its normal execution priority of 1. Task T_3 gets the global lock on $O_{track1} \rightarrow$ *write_speed* on Node 1 and begins execution of its GCS at its normal execution priority of $3+4=7$.
- At time t2, task T_4 arrives on Node 2 and preempts T_1. Task T_3 continues execution of its GCS.

- At time t3, task T_4 requests the global lock on $O_{track1} \rightarrow \mathit{read\_altitude}$. Since T_4's GCS priority $(4+4=8)$, is higher than the priority ceiling of $O_{track1} \rightarrow \mathit{write\_speed}$ $(3+4=7)$, it gets the lock on $O_{track1} \rightarrow \mathit{read\_altitude}$ and preempts T_3's GCS. Task T_1 continues the execution of its LCS at Node 2.

- At time t4, task T_4 completes the execution of its GCS and releases the global lock on $O_{track1} \rightarrow \mathit{read\_altitude}$. Task T_3 resumes the execution of its GCS with $O_{track1} \rightarrow \mathit{write\_speed}$. Task T_4 requests a local lock on $O_{track2} \rightarrow \mathit{read\_depth}$. Since T_4's priority (priority = 4) is higher than the priority ceiling of $O_{track2} \rightarrow \mathit{read\_speed}$ (PC = 2), T_4 gets lock on $O_{track2} \rightarrow \mathit{read\_depth}$ and preempts task T_1.

- At time t5, task T_3 completes the execution of its GCS. Execution on Node 2 remains unchanged.

- At time t7, task T_4 completes its execution including execution of its LCS with $O_{track2} \rightarrow \mathit{read\_depth}$ and releases that lock. Task T_1 resumes execution of its LCS on $O_{track2} \rightarrow \mathit{read\_speed}$ on Node 2. Tasks T_1 and T_3 complete their executions at some later times.

Notice that two blockings of high priority task T_4 that occurred in the DPCP example of Section 2.3 (the blocking on the global lock at time t3 and the blocking on the local lock at time t5) are alleviated under the DASPCP.

4.2 DASPCP Properties

As we did with the ASPCP, we present informal proofs that the DASPCP bounds priority inversion, prevents deadlock, and never decreases concurrency compared to the DPCP. Again, the first two proofs follow Rajkumar's proofs in [8].

Deadlock Prevention.

Theorem 4.1 The DASPCP prevents deadlock.

Proof: Informally, our proof of deadlock prevention is based on Rajkumar's results [8]. Since a job can not deadlock with itself, it can deadlock with other jobs. Since the nesting of GCSs and LCSs is prohibited, access to gcs's and lcs's cannot occur within the same critical section. Since each global and local semaphore is accessed only by a single processor, deadlocks can't occur across processor boundaries. The only possibility, we have not considered yet, is a deadlock within a processor. We showed in Section 3.2 that the ASPCP used on each processor excludes deadlock on that node. □

Bounded Priority Inversion.

Theorem 4.2 Under the DASPCP, the blocking experienced by a task T is finite.

Proof: We partition blocking into three types and show that each type is finite.

1. *A Task's Execution on its Local Node.* Task T can be blocked for the duration of at most (n^G+1) local critical sections of lower priority jobs bound to the same processor as T. Here n^G is the number of

GCS's executed by T at remote processors during its period. To see this, realize that task T suspends itself n^G times during one period as its execution is transferred to the GCS at the remote node. Task T may be blocked every time it attempts to resume its execution after returning from the GCS. Under the ASPCP, the blocking time of each resumption is limited by a longest critical section of one low priority task. Thus, T's execution on its local node has bounded priority inversion.

2. *Task T's GCSs.* For every outermost GCS that task T enters at a remote processor, T can be blocked for the duration of one longest GCS of a lower priority priority job executing its GCS at the same node. This follows from the fact that under ASPCP on that node, the GCSs as tasks on that node are limited to blocking by at most one lower priority GCS. Thus, each GCS of T has bounded blocking time.

3. *Blocking by Remote Tasks.* Task T can be preempted by any task T_i residing at a remote node and accessing GCSs on T's host node, as well as by higher priority tasks executing their GCS at the same remote node used by T's GCSs. The execution times of GCSs are finite; the number of tasks is finite; the periods are finite; therefore there may not be an infinite repetition of a task T_i during one period of T. Thus, the blocking due to remote tasks is finite.

Since all types of blocking are finite, the overall blocking for task T is finite. □

Increased Potential Concurrency. The example of Section 4.1 shows how the DASPCP lowers ceilings, which reduces blocking, thereby increasing concurrency.

Theorem 4.3 The DASPCP never decreases concurrency compared to the basic DPCP technique.

Proof: The priority ceiling used by the DPCP is the maximum of the conflict priority ceilings of the DASPCP. Since lower priority ceilings can only mean less blocking time, concurrency can only increase when DASPCP is used instead of DPCP. □

Serializable Execution.

Theorem 4.4 The DASPCP enforces a serializable schedule of method operations for each object.

Proof: The proof is similar to that of Theorem 3.4 - since the DASPCP also defines priority ceilings based on conflict, and conflict is still defined by affected set semantics.

4.3 DASPCP in a RT CORBA System

As we stated in Section 1, the DASPCP can provide concurrency control in a static RT CORBA system. A static RT CORBA system provides for seamless interaction among clients and servers in a distributed, hard real-time environment in which timing constraints must be guaranteed to be met. It allows for the passing of priority information across the distributed system to enable the enforcement of timing constraints. In this section we briefly describe how a RT CORBA middleware model maps to the more general distributed model used by the DASPCP. We also provide an example to illustrate this mapping. We then describe an implementation in which we have extended the *PERTS* (Prototyping Environment for Real-Time Systems) [5] real-time analysis tool to accept RT CORBA constructs as input, and to analyze the system under DASPCP for concurrency control.

Mapping RT CORBA. A RT CORBA system consists of periodic clients, servers on which clients make method invocations, and an Object Request Broker (ORB) through which client/server communication is performed. A client may have multiple deadlines specified within a single execution of its period.

Consider the example of Section 4.1, and assume that each object, O_{track1} and O_{track2} are servers in a RT CORBA system. The clients are as follows:

```
Node 2                          Node 1
Client 1:  period = 6           Client 2:  period = 4
  .                               .
  .  local code                   .  local code
  .                               .
  Otrack1->read_altitude        Otrack1->write_speed
  Otrack2->read_depth           Otrack1->write_alt
  .                               .
  .                               .  local code
  deadline = 2                    .
  .                             end
  .  local code
  .
  Otrack1->read_speed
  Otrack2->read_speed
  .
  .
  .
end
```

$Client_1$ has a period of 6, and it has an intermediate deadline of 2 after the first two method invocations and some associated local code. $Client_2$ has a period of 4, and makes two method invocations to the same object.

In order to analyze and execute such a system under DASPCP, the RT CORBA constructs must be mapped to the lower-level entities of the model used by DASPCP. A RT CORBA client with period p, final deadline d, and m intermediate deadlines d_1 to d_m maps to $m+1$ dependent tasks, t_1 to t_{m+1}, each with period p. Tasks t_1 to t_m have deadlines d_1 to d_m, and task t_{m+1} has deadline d. The tasks depend on each other in that for $i < j$, task t_i must complete before task t_j starts, for all $i, j = 1...m+1$. A method invocation by a RT CORBA client on a server S maps to a critical section which represents the lock granularity for the DASPCP.

Now we examine the example above, and map it to the level of the DASPCP model. Note that this mapping results in a system that is identical to the example of Section 4.1. $Client_1$ maps to two separate tasks because of the intermediate deadline. These tasks are equivalent to tasks T_4 and T_1 in the previous example. $Client_2$ maps to a single task, equivalent to task T_3 in the previous example. The execution of the RT CORBA system above is equivalent to the execution described in the example of Section 4.1.

Extended PERTS Implementation. PERTS is a real-time analysis tool originally developed at the University of Illinois, Urbana-Champaign, and commercialized by Tri-Pacific Software [5]. The tool analyzes systems of real-time tasks, and produces schedulability analysis, based on given criteria. For concurrency control in hard real-time systems, the original PERTS implementation was able to perform analysis based on DPCP. We have modified PERTS to enable it to also analyze a system using DASPCP schedualbility criteria. We have also modified the system to allow for input in the form of RT CORBA constructs.

The original PERTS design allowed users to enter a real-time system in terms of tasks with timing constraints, and the resources that they require. The Graphical User Interface (GUI) for PERTS consists of three parts: the *Task Graph Editor* which allows a user to specify the tasks in the system, how they are related to each other, and which resources each task uses; the *Resource Graph Editor* which allows the user to specify which resources exist on which nodes in the system, and the *Schedulability Analyzer* which allows the user to specify the type of analysis to be performed on the system, and to perform the analysis. One of the responsibilities of the Schedulability Analyzer is to compute the priority ceilings of every resource in the system.

We have modified the PERTS tool to allow a user to enter a RT CORBA system on which the DASPCP can execute. This modification includes changes to all three parts of the tool. Our Task Graph Editor provides a mechanism for specifying intermediate deadlines on client execution. Because the DASPCP represents methods as resources, our Resource Graph Editor allows the user to specify conflicts between methods. Figure 10 displays a resource graph in which there is a conflict (represented by the dotted line, double arrow) between resources *read_speed* and *write_speed*. These two resources are methods of the same server object that both access the *speed* attribute. Based on affected set semantics, they conflict.

The Schedulability Analyzer has been modified in several ways. The first step that is performed by our Schedulability Analyzer is to translate the RT CORBA constructs into PERTS constructs (as described above). This translator generates additions to the task dependencies in the system's task graph based on the dependencies among all tasks produced by the same client. The new Schedulability Analyzer also computes the conflict priority ceilings for each resource for the DASPCP. Recall that for the DASPCP, the conflict priority ceiling of a method is the sum of the PG and the priority of the highest priority task that will access a conflicting method. The Schedulability Analyzer uses the conflict information in the resource graph to compute these ceilings. Once the conflict priority ceilings are computed, the schedulability analysis for DASPCP is identical to that for DPCP (see [8] for more on schedulability analysis of DPCP).

5 Conclusion

This paper has presented the Affected Set Priority Ceiling Protocols (ASPCP) for concurrency control in real-time object-oriented systems. It showed that these protocols: enforce logical consistency by serializable access to objects; prevent deadlock; bound priority inversion; and provide more potential concurrency in object-

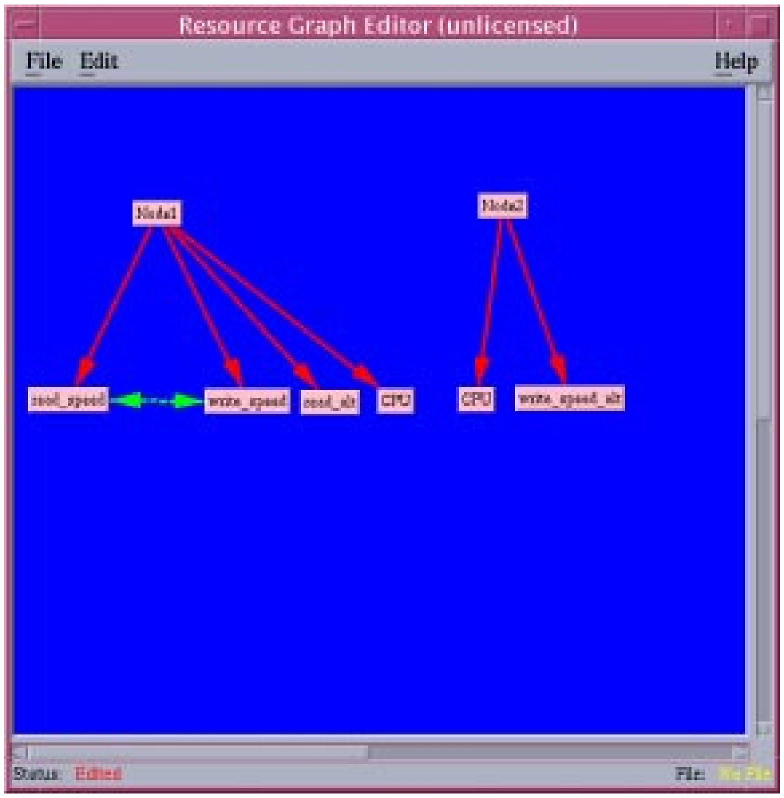


Figure 10: Example Resource Graph for PERTS with DASPCP

oriented systems compared to other priority ceiling techniques. The paper also presented implementations that demonstrate the usefulness of each of the protocols in real-time object-oriented systems.

The ASPCP is suitable for single node systems such as real-time object-oriented databases. We have built the ASPCP into a shared main memory real-time object-oriented database prototoype [9]. The results of performance tests executed on this prototype indicate that on average, the ASPCP misses fewer deadlines than other single-node PCPs, and the the ASPCP reduces priority inversion better.

The DASPCP is suitable for controlling access to distributed objects. The modifications that we have implemented in the PERTS analysis tool will enable users to analyze a RT CORBA system using the DASPCP. The tool also produces task priorities for the system if it is found to be schedulable. We are currently implementing the DASPCP in a static RT CORBA scheduling service with a global Deadline-Monotonic priority assignment. This scheduling service will use the results of the PERTS analysis to automatically assign priorities to the tasks in the system.

The drawbacks of the ASPCP protocols, like all PCP protocols, center on the strong requirement that all task/transactions and their behavior be known *a priori*. For some real-time applications, this assumption is reasonable, for others it is not. Also, PCP protocols potentially allow less concurrency than straight read/write locking and many other traditional concurrency control techniques. However, this reduced concurrency of the PCP protocols is due to the denials of locks that are allow for the deadlock prevention and priority inversion bounds that they provide.

In real-time object-oriented systems where tasks are known *a priori*, a simple and efficient implementation of either ASPCP or DASPCP will yield logical consistency of objects while supporting real-time analysis through deadlock prevention and priority inversion bounding.

References

[1] V. F. Wolfe, L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston, "Real-time CORBA," in *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, June 1997.

[2] T. R. P. S. I. G. of the OMG, "CORBA/RT white paper." ftp site: ftp://ftp.osaf.org/whitepaper/Tempa4.doc, Dec 1996.

[3] L. C. DiPippo and V. F. Wolfe, "Object-based semantic real-time concurrency control with bounded imprecision," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, pp. 135–147, Jan-Feb 1997.

[4] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, Boston, MA, 1991.

[5] M. Squadrito, "Extending the priority ceiling protocol using read/write affected set semantics," Master's thesis, Dept. of Computer Science, The University of Rhode Island, April 1996.

[6] M. Squadrito, B. Thurasingham, L. C. DiPippo, and V. F. Wolfe, "Towards priority ceilings in semantic object-based concurrency control," in *1996 International Workshop on Real-Time Database Systems and Applications*, March 1996.

[7] B. Badrinath and K. Ramamritham, "Synchronizing transactions on objects," *IEEE Transactions on Computers*, vol. 37, pp. 541–547, May 1988.

[8] J. W. S. L. et. al, "PERTS: A prototyping environment for real-time systems," Tech. Rep. UIUCDCS-R-93-1802, The University of Illinois, Urbana, May 1993. Commercial version information available at www.tripac.com.

[9] A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham, "ASSET: A system for supporting extended transactions," in *Proceedings of ACM SIGMOD Conference*, May 1994.

[10] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental evaluation of real-time transaction processing," in *IEEE Real-Time Systems Symposium*, Dec. 1989.

[11] R. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," in *14th VLDB Conference*, Aug. 1988.