Adaptable Concurrency Control for Atomic Data Types

M. S. ATKINS and M. Y. COADY
Simon Fraser University

In many distributed systems concurrent access is required to a shared object, where abstract object servers may incorporate type-specific properties to define consistency requirements. Each operation and its outcome is treated as an event, and conflicts may occur between different event types. Hence concurrency control and synchronization are required at the granularity of conflicting event types. With such a fine granularity of locking, the occurrence of conflicts is likely to be lower than with whole-object locking, so optimistic techniques become more attractive.

This work describes the design, implementation, and performance of servers for a shared atomic object, a semiqueue, where each server employs either pessimistic or optimistic locking techniques on each conflicting event type. We compare the performance of a purely optimistic server, a purely pessimistic server, and a hybrid server which treats certain event types optimistically and others pessimistically, to demonstrate the most appropriate environment for using pessimistic, optimistic, or hybrid control. We show that the advantages of low overhead on optimistic locking at low conflict levels is offset at higher conflict levels by the wasted work done by aborted transactions.

To achieve optimum performance over the whole range of conflict levels, an adaptable server is required, whereby the treatment of conflicting event types can be changed dynamically between optimistic and pessimistic, according to various criteria depending on the expected frequency of conflict.

We describe our implementations of adaptable servers which may allocate concurrency control strategy on the basis of state information, the history of conflicts encountered, or by using preset transaction priorities.

We show that the adaptable servers perform almost as well as the best of the purely optimistic, pessimistic, or hybrid servers under the whole range of conflict levels, showing the versatility and efficiency of the dynamic servers.

Finally we outline a general design methodology for implementing adaptable concurrency control in servers for atomic objects, illustrated using an atomic shared B-tree.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; D 4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *mutual exclusion*; *synchronization*; D.4.8 [**Operating Systems**]: Performance—*measurements*; *simulation*; H.2.4 [**Database Management**]: Systems—*concurrency*; *transaction processing*

General Terms: Design, Performance

Additional Key Words and Phrases: Concurrent access to shared type-specific abstract data types, hybrid locking, optimistic locking, pessimistic locking, transactions serializability

Authors' address: School of Computing Science, Simon Fraser University, Burnaby, B. C. V5A 1S6, Canada; email: stella@cs.sfu.ca

1. INTRODUCTION

Concurrency control for a classical transaction model, where transactions perform READ and WRITE operations on records in a database, is based on *serializability*. Serializability ensures that operations from simultaneously active transactions are interleaved in such a way that each transaction is provided with a consistent view of the system state, as if it were executing alone. Classical database transactions are serialized by analysis of the READ/WRITE and WRITE/WRITE conflicts between concurrent transactions. Recently, techniques which allow more concurrency through additional interleavings than are permitted using these conflicts have been proposed, which employ type-specific properties of objects to recognize when certain operations such as concurrent WRITE operations need not conflict [13, 14, 16–18, 33–35].

For example, a *semiqueue* [28, 33] is a species of queue that does not guarantee to dequeue items in the order they were enqueued. Hence, all elements are equally eligible for dequeuing. Type-specific concurrency constraints for a semiqueue are such that concurrent transactions executing *Enqueue* operations do not conflict, and neither do *Dequeue* operations that attempt to remove different elements. By incorporating this kind of semantic information into a precise definition of type-specific concurrency control, the potential for concurrent access can subsequently be enhanced. Global correctness is achieved because the semiqueue maintains serializability which satisfies a nondeterministic specification [33].

Fundamental approaches used by all concurrency control schemes can be broadly categorized as either *pessimistic* or *optimistic*. In a pessimistic approach, conflicts between concurrent transactions are identified during a transaction's execution and resolved by imposing a delay on some transactions. In an optimistic approach, conflicts are identified at the end of a transaction's execution and resolved by aborting and restarting some transactions at a later time. Standard pessimistic concurrency control mechanisms for the classical transaction model, surveyed in Bernstein and Goodman [3], have been based on two-phase locking [12], multiversion timestamps [24], and hybrids of these approaches. Optimistic methods are based on validation [19].

Optimistic and pessimistic approaches to concurrency control are appropriate under opposing sets of conditions, characterized by the probability of conflict between concurrently executing transactions. An optimistic approach is efficient only if the amount of wasted work (i.e., work performed by an aborted transaction) is relatively insignificant. Hence, it is only cost-effective if the level of conflict is sufficiently low. Conversely, a pessimistic scheme is less efficient when the level of conflict is low, due to locking overhead causing transactions to delay, and increasingly cost effective when the conflict level is sufficiently high. As a result, cost-effective synchronization for transactions should not just be equipped with one technique or the other, but both. Such a dual mechanism or *hybrid* approach could then allow for the selective application of the most efficient method of concurrency control.

Therefore, to provide cost-effective transaction management, several researchers have proposed hybrid optimistic/pessimistic schemes for the concurrency control of transactions in databases, together with nontrivial proofs of correctness of compatibility [6, 7, 21, 22, 25, 26, 29]. Herlihy has extended these hybrid concurrency control schemes to work on abstract data types using type-specific concurrency control mechanisms to enhance the number of permissible concurrent interleavings of transactions [16, 18]. Herlihy's scheme is designed to allow transactions to execute either optimistically or pessimistically at a particular shared data object; hence, for example, allowing a transaction accessing an airline to use optimistic techniques within the airline's database while the same transaction can access the bank using pessimistic techniques [18].

All hybrid techniques must simultaneously support more than one concurrency control method, and because of the increase in overhead for a hybrid implementation it is not clear whether or not a practical advantage can be obtained from using hybrid servers.

We have implemented a hybrid concurrency control method based on [16, 18] for transactions accessing a shared abstract object—a shared semiqueue —through a server, and we have obtained experimental evidence that pessimistic and optimistic mechanisms that exploit data type semantics are efficient under different circumstances. We chose to implement a semiqueue server to test our ideas, as the theory and correctness proofs for hybrid semiqueue servers are already published [16]. This hybrid method allows optimistic or pessimistic control to be applied at a per conflict-type granularity. We compare the performance of a purely optimistic server, a purely pessimistic server, and a hybrid server that treats certain conflict types optimistically and others pessimistically, to demonstrate the most appropriate environment for using pessimistic, optimistic, or hybrid control. In order to achieve reasonable performance, we needed some nonobvious performance enhancements such as an "early abort" scheme for backward-oriented optimistic servers (detailed in Section 2.6.2), and type-specific deadlock detection and type-specific "wakeup" calls for the purely pessimistic server (detailed in Section 2.6.2). We found, as expected, that the advantages of low overhead on an optimistic approach at low conflict levels is offset at higher conflict levels by the wasted work done by aborted transactions. Conversely, at low conflict levels, the pessimistic locking overhead mars the performance of the pessimistically treated conflict types.

We chose to implement one type of object—a shared semiqueue server—to test the design principles, and we show that similar techniques can be applied to another type of shared data object, a B-tree. We speculate that these techniques can be generalized to other shared data structures with similar characteristics, with the amount of performance enhancements possible (such as type-specific deadlock detection) determining the conflict percentage at which pessimistic techniques become more efficient than optimistic.

A server that can dynamically change between optimistic and pessimistic treatment of events according to the percentage of conflict should have a

performance advantage over a nonadaptable (static) server, provided the overhead of changing from one mode to the other is not too high. We have designed and implemented a novel adaptable server for controlling access to a shared data object, whereby the treatment of conflict types can be changed dynamically between optimistic and pessimistic, according to three heuristics based on the state of the object, its conflict history, and some semantic information related to its transactions, so that optimum performance can be achieved over almost the whole range of conflict levels. The adaptable semiqueue server can thus allow one transaction executing a successful *Dequeue* operation with an optimistic approach to execute concurrently with another transaction executing a successful *Dequeue* operation using pessimistic locking. However, as Reidl notes in [25], that, although adaptability is a powerful tool, its use requires careful guidance because of the overheads necessary to combine and support more than one data structure and algorithm. We show that the adaptable server performs as well as the best of the purely optimistic, pessimistic, or hybrid servers over the whole range of conflict levels, showing the versatility and efficiency of the dynamic server.

We then extend our designs to other types of shared abstract objects such as a shared B-tree index and show that the same relative performance advantages can be expected for these adaptable servers. Hence an adaptable dynamic server is to be preferred over the static locking mechanisms traditionally employed in the control of concurrent transactions.

1.1 Related Work

Performance studies indicate that optimistic approaches work well at low conflict levels and low resource utilizations, whereas pessimistic approaches work well at higher conflict levels [2, 9]. Based on this, several workers have suggested "hybrid" approaches for mixed optimistic/pessimistic concurrency control in databases and proved their compatibility [7, 21, 22, 26, 29], although to our knowledge only a prototype of Robinson's has yet been implemented.

The RAID distributed database system [5] can use optimistic or pessimistic algorithms for concurrency control. Reidl and Bhargava present RAID performance data [5, 25] for an adaptable hybrid concurrency control scheme based on simple READ/WRITE or WRITE/WRITE conflicts. Unlike our scheme, theirs does not automatically adapt between modes, although they have designed a prototype expert system for controlling an adaptable distributed system [4]. Their results indicate that an adaptable hybrid system can be cost effective because the concurrency control is only 10–15% of the transaction execution time, so that the increased overhead in the concurrency control mechanism is not significant compared with the gains in using the most appropriate concurrency control algorithm to reduce the number of aborted transactions. Our scheme also exploits semantic information in our adaptable servers, leading to a greater amount of concurrency; hence the performance advantages of adaptability may not be the same as those for classical transactions in RAID with no type-specific information to increase concurrency.

At the same time, other researchers were exploring how to exploit semantic information on user-defined abstract data types in order to increase the amount of concurrency a synchronization mechanism can provide [13, 14, 16–18, 28, 33–35]. All except Herlihy's are purely pessimistic schemes. Argus [23], TABS [30], and Camelot [31] are existing systems that provide pessimistic-based support for extended transactions on user-defined abstract types. A specification framework for atomic data types and precise limits of concurrency levels for pessimistic schemes are defined in [34], but no performance data are available. In fact, the only performance data for exploiting semantic information to increase concurrency appears in [11], where Cordon and Garcia-Molina show that under high levels of conflict, conventional two-phase locking schemes are outperformed under certain circumstances by pessimistic schemes that exploit semantic information. This result was attributed to the significant increase in the amount of concurrency the semantic-based control could support.

Herlihy in [18] gives several references to performance studies of optimistic/pessimistic schemes, which are analyzed in [2]. The results indicate that traditional optimistic techniques are probably most useful if they can be applied to individual objects rather than monolithically to the whole system. This is exactly the environment described in this paper.

1.2 Overview of Paper

Section 2 describes type-specific optimistic, pessimistic, and hybrid control and performance enhancements used in our implementation through the example of a semiqueue server. Section 3 describes three different approaches to adaptable dynamic allocation. Section 4 presents performance tests that demonstrate the practicality and efficiency of this highly flexible means of concurrency control. Section 5 presents the design methodology to be used for extending the technique to general shared abstract data types through an example of a B-tree index. Section 6 presents conclusions.

2. TYPE-SPECIFIC CONCURRENCY CONTROL USING EVENTS

2.1 Objects, Events, and Servers

An *object* is a container for data, and each object has a *type* that defines a set of primitive operations and a set of possible values that provide the only means for transactions to create and manipulate the data in the object. An *event* consists of an operation invocation and a response based on semantic success (Ok) or failure (Failed) of the invocation, independent of concurrency concerns [18]. A *server* completely encapsulates its data object and the operations that manipulate it. Consequently, a server is responsible for controlling the concurrent access of transactions operating on its object.

For correct global execution of a set of transaction accessing more than one object, the servers must employ a *local atomicity* property [34]. Further, as concurrency control and recovery interact in subtle ways [26, 35], each server must use the *same* local atomicity property for correct global execution. Thus,

one server using socalled *dynamic atomic* correctness criterion may not produce a serializable global schedule with another server utilizing *static atomic* concurrency control, where *dynamic atomic* refers to a protocol such that each server determines a serialization order on transactions dynamically, based on the order in which transactions invoke operations and obtain locks on elements. *Static atomic* refers to a protocol that orders transactions statically, based on timestamps chosen when transactions start. Both protocols enforce unnecessary waiting for certain transactions. We, like Herlihy [18], use a *hybrid atomic* local atomicity property, whereby timestamps for update transactions are chosen dynamically as they commit [34]. This allows for more concurrency than dynamic atomicity because the object servers have more information, namely, the timestamps assigned to update transactions as they commit.

2.2 Recovery Methods

A conflict-based approach is used for concurrency control (CC), where each server has a set of predefined conflicts between pairs of events. Recovery from conflicts is achieved by defining each abstract data object as being a composite of two components: a permanent state and a set of intentions lists. An object's permanent state records the effects of transactions that have terminated successfully. There is an intentions list, recording tentative changes, for each active transaction that has accessed the object through execution of an event. When a transaction commits, changes in its intentions list are applied to the permanent state of the object. If a transaction aborts, its intentions list is simply discarded. This recovery method is called a *deferred update* (DU) strategy, where intentions lists or private workspaces are used to buffer modifications to the permanent state of the object until a transaction commits. As Robinson points out in [26], DU recovery allows separation of the CC and recovery mechanisms. The servers are responsible for the CC, and the only interaction between the two occurs when the server is informed by the recovery subsystem that a transaction's intentions list must be applied to the object. DU is used in all the research work on hybrid concurrency control because all algorithms work correctly with it, and in some commercial schemes such as INGRES [27] and POSTRES [32]. However, it is important to note that for a purely pessimistic approach, an *update in place* (UIP) recovery method (as opposed to a strategy that relies on DUs) can provide more efficient access to some abstract data types. (We discuss this further in Section 2.6.5.)

2.3 Pessimistic Locks and Optimistic Flags

Transactions access data managed by a server through execution of an event. The server managers CC on elements of its data through use of event-specific pessimistic locks and optimistic flags.[1] Pessimistic transactions mark the element(s) of data accessed by an event with a pessimistic R or W lock. This

[1] Herlihy [18] refers to "optimistic locks," but we prefer the term "optimistic flags," as there is no delay associated with an optimistic flag.

P-lock can be used to delay another permissible transaction executing an event accessing the same element(s). Optimistic transactions mark the elements(s) of data accessed by an event with an optimistic R or W flag which is used to identify R/W sets at validation (commit) time. The P-locks do not use a transaction id (TiD). However, the Tid is used with the optimistic flags (O-flags), so that the "early abort" optimization described in Section 2.6.2 can be implemented.

2.4 Conflict-Based Concurrency Control

Conflict-based concurrency control is based on predefined conflicts between pairs of events. Optimistic conflicts are detected after a transaction has performed all of its events upon its intentions list, but before any changes have been applied to the permanent state of the object. In an optimistic scheme, successful validation depends on the absence of conflicting events between the validating transaction and other concurrent transactions (see also Section 2.6). In a pessimistic scheme, the conflicts are used to introduce delays. In a hybrid atomic local atomicity scheme for CC, pessimistic conflicts must be symmetrical because they are detected at the time when an event is executed and the commit order of transactions is unknown. A pessimistic lock (P-lock) for each event must be obtained before that event can be added to a transaction's intentions list. Identification of the conflict types between the events of concurrently active transactions is accomplished by deriving a set of *proscribed serial dependencies*. Weihl notes that determination of these serial dependencies is based on the choice of local atomicity property [34]; we use the hybrid atomicity protocol, and serial dependencies are derived directly from the abstract data type's specification.

2.5. Example: A Semiqueue Server

As an example, we consider the abstract data type, *semiqueue*, which allows for more concurrency than a strictly FIFO queue because it does not guarantee to dequeue items in the order they were enqueued [33]. As a result, all elements in the permanent state of the semiqueue are all equally eligible for dequeuing. Let the operations associated with this object be *Enq*, *Deq*, and *Inspect*, which add, remove, and inspect (a generic $O(n)$ read-only operation where n is the number of items in the semiqueue) elements, respectively. Inspect also returns a tally of the number of items in the queue. An attempt to remove an item from an empty queue results in a "Failed" response. Recall that using deferred updates, the *Enq* and *Deq* operations are made in intentions lists, and the effects are not seen by any other transactions until a transaction attempts to validate and commit.

2.5.1 *Optimistic Conflicts.* Type-specific conflicts are formally expressed as dependencies D1 through Dn; the optimistic conflicts at a semiqueue are shown in Table I, where events in the rows have been executed by a transaction Tj attempting to validate. Events in the columns are events that have been executed by any other concurrently active transaction(s). The events may have been executed in any order at the semiqueue server, but the

Table I. Optimistic Conflicts at a Semiqueue

	Enq(i)/OK()	Deq()/OK(i)	Deq()/Failed()	Inspect()/OK(*#items*)
Enq(i)/OK()			D1	D2
Deq()/OK(i)		D3		D4
Deq()/Failed()				
Inspect()/OK(*#items*)				

Table II. Pessimistic Conflicts at a Semiqueue

	Enq(i)/OK()	Deq()/OK(i)	Deq()/Failed()	Inspect()/OK(*#items*)
Enq(i)/OK()			D1	D2
Deq()/OK(i)		D3		D4
Deq()/Failed()	D5			
Inspect()/OK(*#items*)	D6	D7		

optimistic conflicts are only detected when a transaction attempts to validate. For example, in the case of forward-oriented CC (see Section 2.6.2) dependency, D1 reflects the fact that if Tj had enqueued an item, a conflict with a concurrently active transaction that performed an unsuccessful dequeue event (a *Deq( )/Failed( )* event) will result. Dependency D2 shows that enqueuing an item also conflicts with an inspection and tally. Dependency D3 prevents two transactions from removing the same item. Dependency D4 shows that a successful dequeue event invalidates a subsequent inspection. Note that the optimistic conflicts are not symmetric; if Tj had executed a *Deq( )/Failed( )* event, and another concurrent active transaction had executed an enqueue event, then Tj is allowed to validate—there is no conflict here. Upon uncovering any of these conflicts, one of the offending transactions is aborted (usually Tj trying to validate) and restarted as new.

2.5.2 *Pessimistic Conflicts.* Pessimistic transactions also have conflicts D1 through D4, and in addition, they must also include their symmetric counterparts D5, D6, and D7, as shown in Table II. The extra conflicts D5 and D7 are introduced because the transactions are delayed at execution time, and it is not known in which order the transactions will commit. Thus if Tj executes a *Deq( )/Failed( )* event, it must delay (because of D5) if any other active transaction Tk is holding and *Enqueue P-lock*, even if Tj would actually have "liked" to commit before Tk.

2.6 Implementation Issues

2.6.1 *Introduction.* We implemented a semiqueue server using the high-level language SR [1] and tested it using different artificial workloads of transactions. An overview of the implementation is given in Appendix A; details are given in [10]. SR contains may language features for concurrent programming which we found very helpful in implementing our various CC algorithms. We found our initial implementations were not very efficient [10]; we needed to design performance enhancements for an effective implementation of a hybrid concurrency control mechanism. Although the performance enhancements described here are specific to the semiqueue example, several of the same techniques can be applied to other data structures such as a B-tree (see Section 5.2) and a directory [10].

2.6.2 *Type-Specific Performance Enhancement for Optimistic Conflicts: BOCC and Early Abort.* Within an optimistic scheme, concurrency control techniques are enforced during a transaction's validation phase. Validation schemes can be classified into two categories [15, 18]: backward- and forward-oriented concurrency control (BOCC and FOCC, respectively). Basically, given transaction Tj that is trying to validate, BOCC checks for conflicts between Tj and transactions that were concurrently executing with Tj that have previously validated. If there are any conflicts of this kind, Tj has been invalidated and must be aborted. FOCC, however, checks for conflicts between Tj and transactions that were concurrently executing with Tj and that are still active. If there are any conflicts of this kind, then several courses of action can be taken, the simplest of which is to abort transaction Tj. We found that the performance of BOCC was similar to FOCC [10]. However, a type-specific performance enhancement is possible for BOCC which can be supplemented with our type-specific early-abort mechanism.

According to BOCC, given transaction Tj that is attempting to validate, if there are any conflicts between the events executed in Tj and transactions that were executing concurrently with Tj but have previously committed, Tj will be aborted. In BOCC, conflicts are identified by allocating timestamps according to a logical clock [20]. Timestamps are issued upon each active transaction's first execution of an event that could potentially be invalidated. In addition, timestamps are maintained for the most recent commitment of potentially invalidating events. Validation is successful according to BOCC iff all of the validating transaction's timestamps, representing its first execution of events that may be invalidated, are greater than the timestamps corresponding to the most recent commitment of invalidating events.

In the case of the conflict type that arises between concurrent transactions attempting to dequeue the same element from the semiqueue (D3), the BOCC technique relies on an early-abort strategy. An optimistic approach permits any number of active transactions to include the exact same element of the shared semiqueue in their intentions lists. In BOCC, the first transaction to successfully validate will actually remove this conflict-causing element from the permanent state of the semiqueue. As a result, all active transactions that also performed a dequeue of that element (in their respective intentions

lists) become unconditionally invalidated. Hence, BOCC's early-abort strategy, wherein active transactions are aborted before reaching their validation phase, kills transactions if their intentions lists include the removal of an element that is being dequeued from the permanent state of the semiqueue by a committing transaction. Treatment of this particular conflict type lends itself to this strategy by virtue of the fact that all transactions attempting to dequeue a given element are identified within optimistic flag information associated with that element. Optimistic flags are not blocking (as is the case for pessimistic locks). Hence, it is most cost effective to exploit this optimistic flag information at the time when the element is being removed, as opposed to waiting until each transaction involved in this type of conflict attempts validation.

This approach can be used for any similar conflict arising in any abstract data type—for example, on deleting a node from a B-tree using optimistic concurrency control (see Section 5.2).

2.6.3 *Delaying Mechanisms for the Pessimistic Server.* In situations where a transaction's request for a P-lock is denied (because another transaction already holds a P-lock for a conflicting event), the event must be discarded and the transaction must retry the invocation. In order to accommodate these delayed/blocked operations, special wake-up calls have been included in the protocol associated with the release of P-locks, as described in [33]. For example, with an *Enq* operation, the server checks to ensure that no other active transaction holds a P-lock on either a *Deq*( )/*Failed*( ) or an *Inspect*( )/*Ok*(*items*) event (D1 and D2 in Table II). If there is a conflict, the *Enq* invocation will block, waiting to receive a wake-up message. Wake-up messages are sent out to blocked operation invocations every time a transaction that has performed an event that conflicts with the blocked operation has released an appropriate P-lock. This way, when a transaction that executed a *Deq*( )/*Failed*( ) event releases its *Deq*( )/*Failed*( ) P-lock, the blocked *Enq* operation will be reactivated and retried. If this P-lock happened to be the blocked *Enq*'s sole impediment, the operation proceeds, otherwise it will be blocked again and must wait for the next wake-up message.

When a transaction has successfully obtained all of its necessary P-locks and has finished executing, it inflicts the changes recorded in its intentions list to the global state of the semiqueue. Once these changes have been accomplished, the committing transaction's P-locks can be released and wake-up calls issued to any blocked invocations that could benefit by these releases. The alternative of using timeouts will cause unnecessary delays to waiting transactions.

2.6.4 *Type-Specific Deadlock Detection.* Given two transactions, T1 and T2, and the following combination of events:

T1: Deq( )/Ok(k)
T2: Deq( )/Ok(1)
T1: Inspect—BLOCKED, waiting for wake-up from T2
T2: Inspect—BLOCKED, waiting for wake-up from T1

it becomes evident that some kind of type-specific deadlock prevention or resolution mechanism must be included in the pessimistic server. Given that Ti is about to block on Tj, a simple type-specific prevention mechanism enforces the following rule:

Deadlock Rule. For Ti to block on Tj, Tj can not already be blocked on a lock that Ti possesses.

Consequently, in the example above, T2 will not wait for T1 since T1 is already blocked and waiting for the release of T2's *Deq( )/Ok(l)* P-lock. In this situation, T2 would be aborted and have to be restarted.

Is this mechanism robust enough to handle deadlock between more than two transactions? The answer to this depends on the kinds of circular waits that can arise among transactions. In the case of the semiqueue object server, we show all circular waits of length n must contain a cycle of length two, hence this mechanism is sufficient. The proof for this type-specific deadlock is in Appendix B.

The implementation of this deadlock prevention algorithm, which only checks for circular waits between two concurrent transactions, is very efficient, and unlike timeouts, it means that transactions don't have to wait unnecessarily. The implication of this efficient deadlock-prevention mechanism is that the pessimistic CC scheme is favoured more than would be usually the case; we discuss this further in Section 5.

Unfortunately, it transpires that this simple kind of type-specific deadlock prevention cannot be extended to all other data types; a counter-example for the B-tree data type is given in Section 5.2.

2.6.5 *Use of UIP Recovery Strategy.* An update-in-place, rather than a deferred update recovery strategy can possibly provide more efficient access to some abstract data types, when used with purely pessimistic locking methods. Weihl proposes [33] that transactions accessing a semiqueue could use an intentions list for *Enq* operations (the DU technique), but perform *Deq* operations directly on the permanent state of the object (i.e., use UIP). Consequently, since the permanent state of the object contains only those items which have been Enqueued by committed transactions and have not yet been Dequeued, a *Deq* operation does not have to search the semiqueue for an "unlocked" item. This is an improvement over the pessimistic DU approaches for semiqueue implementations presented in [23, 30, 31, 33] (and all the hybrid schemes referenced here), where items in the permanent state of the semiqueue are individually locked by active transactions intending to Dequeue them. As a result, in all these systems *Deq* operations must search the semiqueue for an eligible (unlocked) item, and hence the cost of a *Deq* operation is proportional to the number of items in the semiqueue.

However, the UIP recovery strategy is not compatible with many concurrency control methods, and using UIP would also increase the cost of aborts. Further, as all the optimistic and hybrid techniques referenced here use DU recovery and as it is not clear how well the UIP recovery can be incorporated

Table III. Hybrid Pessimistic (P) and Optimistic (O) Conflicts at a Semiqueue

	Enq(i)/OK()	Deq()/OK(i)	Deq()/Failed()	Inspect()/OK(*#items*)
Enq(i)/OK()			D1(O)	D2(P)
Deq()/OK(i)		D3(O)		D4(P)
Deq()/Failed()				
Inspect/OK(*#items*)	D6(P)	D7(P)		

in a hybrid server without increasing the conflicts and the number of aborts, we decided not to make this performance enhancement to our hybrid scheme.

2.7 Hybrid Control

The hybrid synchronization mechanism introduced in [16] supports the selective application of type-specific optimistic and pessimistic control on a per conflict-type basis. Hence, for any abstract object, each type-specific conflict can be treated independently with an optimistic or pessimistic strategy.

Deciding which conflicts to treat optimistically and which to treat pessimistically is a nontrivial task, as the efficiency of the hybrid depends on the workload. For example, we actually implemented and tested several different hybrid semiqueue servers [10]. The most interesting performance characteristics among these was exhibited by the server defined by the following six proscribed serial dependency relations, illustrated in Table III. This hybrid server treats all conflicts associated with Inspect( )/Ok(*items*) events pessimistically, and so is appropriate for a situation where *Inspect* events are expected to occur frequently.

As described in the context of the optimistic server, optimistic conflict types are resolved by transaction abort during the validation phase (which can be either FOCC and BOCC). Pessimistic conflict types, however, are identified during a transaction's execution (hence the symmetry of these conflict-types) and resolved by imposing a delay. Thus it is possible for a transaction Ti to hold pessimistic locks and optimistic flags at the hybrid server. For example, Ti could do an *Inspect* (pessimistic) event and a *Deq* (optimistic) event.

3. DYNAMIC ADAPTABLE SERVERS

3.1 Feasibility

Each of the pessimistic, optimistic, and hybrid servers have performance advantages over the others, depending on the percentage of conflict between active transactions. For example, our performance results show that an optimistic server behaves very badly relative to a pessimistic server when there is more than approximately 25% conflict between *Deq( )/Ok(i)* events (see Section 4.2.2). As already mentioned in the Introduction, a server that

could dynamically change between an optimistic or pessimistic treatment of events according to the percentage of conflict should have a performance advantage over the static servers described in Section 2, provided the overhead of changing from one mode to the other is not too high.

First though, we consider the feasibility of such a dynamic server. A dynamic server must be able to treat each event either optimistically or pessimistically, and must be able to switch between these modes without invalidating serializability constraints on any active transactions. For efficiency reasons, during the lifetime of each transaction's interactions with a particular object server, all the type-specific conflicts must always be treated the same way. For example, if transaction Ti executes, say six *Deq( )/Ok(i)* events at the same server for object A, they must be either all optimistically or all pessimistically treated; it is inefficient to allow Ti's *Deqs* to switch from optimistic to pessimistic treatment at server A. If this were not the case and a transaction Ti could switch from optimistic *Deqs* to pessimistic *Deqs*, then, at validation time, Ti could be aborted because of an optimistic type conflict, which would be very inefficient because of Ti's wasted time obtaining P-locks.[2] However, a transaction can be optimistically treated at one server and pessimistically at another; this could be desirable for transactions touching several different databases, although efficiency arguments against this still apply. Further, the transaction's event types need not all be the same at the same object server; for example, a hybrid server for semiqueue A permits Ti to execute all *Deq( )/Ok(i)* events optimistically (i.e., without delay) and all *Inspect( )Ok(items)* events pessimistically, as shown in Table III.

The difference between the static server and the dynamic adaptable server is that the latter allows some transactions to have purely optimistic interactions, while simultaneously allowing other transactions to have purely pessimistic, or hybrid, interactions. Proof of correctness for this approach is given in [16].

We allocate transaction Tj a parameter called its *class* which is valid during Tj's interaction with a particular server. The adaptable server uses the class to determine whether conflicts involving this transaction are all treated optimistically, pessimistically, or as a hybrid. We use three values for the class: *o*, *p*, and *h*, corresponding to optimistic, pessimistic, and hybrid. For the example of a semiqueue server, for an o-class transaction, Tj, proscribed serial dependency relations D1 through D4 are treated optimistically as shown in Table I. If Tj executes a *Deq( )/Failed( )* event and any other concurrently active transaction, Tk, executed *Enq(i)/Ok( )*, then the dependency relation D5 shown in Table II is not asserted on Tj if Tj is validating while Tk is still active (i.e., Tj is not aborted or delayed, and Tk is not invalidated). Note that if Tj is a p-class transaction executing a *Deq( )/Failed( )* event and if Tk is o-class or h-class and executed *Enq(i)/Ok( )*, no *Enq(i)/Ok( )* P-locks are held by Tk so Tj is not delayed.

[2] Note that Reidl does provide this feature by allowing the user to manually switch the class of a transaction in the middle of its execution.

However, if Tk is a p-class transaction, then D5 is asserted and Tj is delayed on the *Deq( )/Failed( )* event.

3.2 Interclass Conflict Resolution

The interclass conflicts arising between, say a p-class Tj and an o-class Tk are resolved at validation/commit time (not during execution). The dynamic server permits transactions of different classes to execute concurrently. The implementation of the dynamic server combines features of all three static servers (i.e., O-flags and P-locks), and in addition, contains interclass conflict-resolution strategies.

In most abstract data types, the cost for executing an event for an optimistic transaction will be less than those for a hybrid which in turn will be less than for a pessimistic transaction. This is because of the amount of overhead associated with conflict detection for each event within each scheme. As optimistic events do not include any such overhead (since conflicts are not detected until the validation phase of an optimistic transaction), these events are the least costly. Pessimistic events, however, always include conflict-detection (locking) overhead, and hence are the most costly. For example, in the case of the hybrid semiqueue server presented in Section 2.7 with conflicts shown in Table III, where a successful *Deq* event involves both optimistic conflict resolution (D3) and pessimistic conflict resolution (D4), a hybrid *Deq* event is more costly than purely optimistic, but less costly than purely pessimistic *Deq* events. Other types of hybrid servers, however, could be defined differently for *Deq* events. They could have as little overhead as the optimistic scheme, dictating an order of: optimistic = hybrid < pessimistic, or as much overhead as the pessimistic scheme: optimistic < hybrid = pessimistic.

Conflicts between concurrent transactions of different classes are resolved by introducing a *suicidal* (S) strategy into the validation procedure for optimistic transactions and the optimistic component of hybrid transactions, and *commit-and-kill* (C & K) strategy into the commitment procedure for the pessimistic component of hybrid and all pessimistic transactions, as shown in Table IV. Note that h-class transactions first try to validate optimistically, and if successful, commit in a single atomic step, called (val & com).

Row 1 of the interclass conflicts shown in Table IV means that an optimistic transaction will commit suicide during its validation if it conflicts with an active hybrid or pessimistic transaction. Row 2 means that a hybrid transaction attempting to validate and commit will succeed if it conflicts with an active optimistic transaction, but the conflicting o-class transaction will be killed. However, if the h-class transaction attempting to validate and commit conflicts with an active p-class transaction, the h-class transaction will have to commit suicide. Row 3 means that a pessimistic transaction trying to commit will always do so, but if there is an active conflicting o-class or h-class transaction, the latter must be killed. In summary, interclass conflicts are resolved by aborting the transaction of the lesser class, which minimizes the amount of wasted work.

Table IV. Interclass Conflicts between Transactions

	To(act)	Th(act)	Tp(act)
To(val)		S	S
Th(val&com)	C&K		S
Tp(com)	C&K	C&K	

Our concern is that the extra overhead in the adaptable server in resolving these interclass conflicts is too high for the advantages gained in providing such flexibility. As Reidl notes in [25], an adaptable server is necessarily less efficient than a static server because individual concurrency controllers perform better when they have the freedom to manipulate the data structure as they wish. However, for the particular case of locking and optimistic, it works quite well, since they have similar constraints on concurrency and hence similar data structures. We are encouraged by Reidl's findings that the overheads on adaptability in a classical concurrency control scheme are not significant, and we therefore proceeded with the designs and implementations for three kinds of adaptable servers, detailed below.

3.3 Transaction Class Allocation

Lausen defined two ways to generalize a hybrid optimistic/pessimistic concurrency control scheme, based on state information in the shared object (such as the load factor for a B-tree) or based on the number of restarts [22]. We investigated these two approaches, extended the second to use general conflict information, and added a third, that of preassigning a transaction class to a transaction on the basis of priority. We then implemented different adaptable semiqueue servers which could allocate a class (optimistic, pessimistic, or hybrid) to a new transaction attempting to execute an event at the server on the basis of the following:

(1) *State information*: size of the queue determines class. With a small queue size, p-class is chosen, as the *Deq* conflict probability is high. For queue size above a threshold, o-class is chosen, as the *Deq* conflict probability is lower.

(2) *Conflict-based assignment*: new transactions are assigned a class according to the percentage of conflict types in the current environment of the semiqueue (similar to Lausen's suggestions based on the number of restarts).

(3) *Preassignment*: class is preassigned on the basis of priority.

The server allocates the transactions class in (1) and (2) above, while in (3) a transaction's class is assigned prior to its interaction with the server.

3.3.1 *A State-Based Semiqueue Server.* In this dynamic server, a transaction's class is determined by the size of the semiqueue at the start of the transaction. When the number of eligible elements in the permanent state of the queue is less than some user-defined threshold value, new transactions rely on a pessimistic conflict-resolution strategy. Above this threshold, however, new transactions employ an optimistic strategy. This assignment is based on the assumption that, in general, smaller queues must endure higher levels of *Deq* conflict (D3, which are the most costly) than larger ones. Hybrid control is not included in this type of server, since the difference between optimistic and hybrid control involves their respective treatments of *Inspect* conflicts. It is assumed that in most cases the frequency of these conflict types will be largely independent of queue size. Using the same threshold for both state transitions could lead to flip-flopping between states, but this is not a problem because there is negligible overhead involved in a state change.

3.3.2 *A Conflict-Based Semiqueue Server.* In this adaptable server, the class of a new transaction is determined by the most recent assessment of the level of conflict type in the environment of the semiqueue. Periodically, the server does an assessment of the percentage of conflict types defined by dependencies D3 and D4, which are the most costly conflicts. Based on these results, the server automatically determines the most appropriate class assignment for new transactions. For our server the threshold percentages of conflict are derived from performance tests detailed by Coady [10], described in Section 4.2.2, and shown in Figure 1.

Although this automatic method of dynamic control is potentially the most accurate, and consequently the most effective for dynamic servers, it is also the most costly as it has the most overhead. The accuracy (and cost) of this method is directly related to the frequency of conflict assessment. The heuristic we use to determine the frequency of conflict assessment is the number of concurrent transactions accessing the semiqueue. Assessment starts only when the number of concurrent transactions reaches a predefined threshold value (one hundred in our experiments). If the current class is optimistic, assessment begins at the start of the one-hundreth concurrently active transaction, and ends when all of those one hundred transactions have completed validation. The number of transactions aborted due to each of the conflict-types D3 and D4 is then determined, and the assignment of class to new transactions may be changed from optimistic of hybrid or pessimistic accordingly. In the case where the server is assigning a pessimistic class to new transactions, assessment involves a count of the number of transactions currently blocked by either a D3 or D4 conflict type. Due to the symmetrical nature of pessimistic conflict relations, only one type or the other will exist. When the server is allocating a hybrid class to new transactions, assessment is done via both the optimistic and pessimistic methods described above, with pessimistic assignment taking priority.

3.3.3 *A Preassigned-Class Server.* Another type of dynamic server can exploit semantic information by relying on an assignment of class prior to a transaction's first invocation of the server.

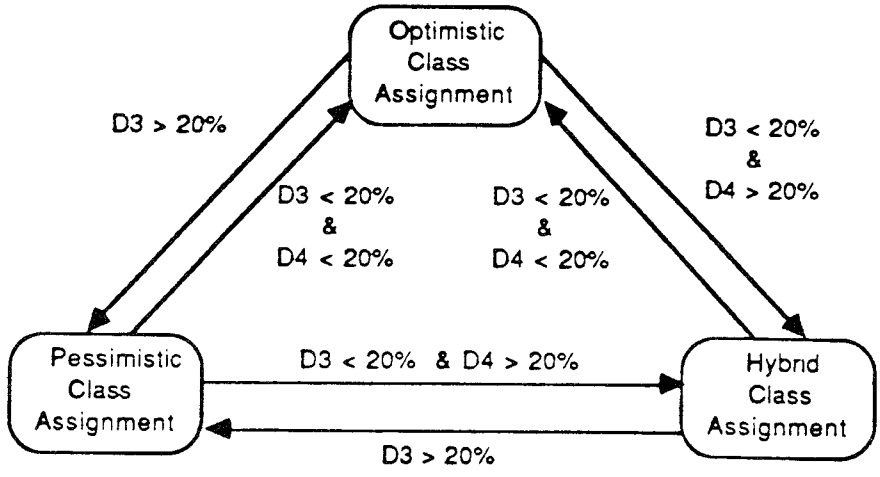

Fig. 1. Conflict-based semiqueue server.

For example, transactions that are predominately *Enq* items ($O(1)$) can be substantially less costly than transactions that are predominately *Deq*, which is $O(n)$, where n is the number of items in the semiqueue, as discussed in Section 2.6.5. Consequently, preassigning *Enq* transactions with optimistic and *Deq* transactions with pessimistic control can ensure the successful execution of the costly *Deq* transactions, while potentially providing the most cost effective means of control for the relatively inexpensive *Enq* transactions. Likewise, in situations where *Inspect* operations ($O(n)$) predominate in certain transactions, hybrid control can be assigned in order to ensure that their execution is not wasted due to a conflict with a less costly *Enq* transaction. This preassignment can be combined with either of the state-based or automatic servers for transactions that do not have their class preassigned. Furthermore, any particular transaction could be preset to run optimistically at one object server and pessimistically at another.

Within some applications the role of a transaction may also contribute to its class. For example, it is possible that an application could employ some transactions that include interactive parts. In the interest of user friendliness, these types of transactions would be most effectively handled as a preassigned pessimistic class, regardless of their anticipated costs.

4. PERFORMANCE AND EVALUATION

4.1 Introduction

We are interested here in determining whether or not the higher complexity of overhead associated with an adaptable application-dependent concurrency control scheme pays off in terms of increased transaction throughput compared with a static, purely optimistic, purely pessimistic, or hybrid scheme.

Our system captures on a small scale the fundamental concurrency considerations addressed by distributed transaction systems such as TABS [30] and Argus [23], where a distributed program consists of client transactions executing operation invocations on object servers. These systems adhere to the philosophy that "coarse-grained" concurrency is much more important than "fine-grained" transaction concurrency—a realistic assumption for many transaction systems where time spent processing a transaction at an object server is small compared with the total transaction time. In support of this, several workers have shown that the execution time of concurrency control algorithms is a small (less than 10%) percentage of the total execution time of a transaction [2, 9, 11, 25]. Reidl shows that, to the limits of his concurrency control experiments, there were no discernible performance differences between his adaptable shared-object implementations and his specialized static implementations [25] because the execution time of concurrency control algorithms is small (a few percent) in comparison to the execution time required to process a transaction, and hence the differences in execution times between specialized and generic algorithms are not significant.[3] Disk accesses reduce still further the percentage of transaction time spent in execution of concurrency control algorithms. For example, simulation parameters used by Carey in [9] assume 1 ms for the concurrency control algorithm and 10 ms for the cpu time to process a data page. At start-up and at commit times disk accesses taking 35 ms are required, and in Carey's experiments every page access also incurs a 35 ms overhead. We decided to use an in-memory shared object with no disk accesses, because we are only considering here the overhead of introducing adaptability into a scheme for concurrency control of access to a shared object. Taking disk accesses into account would only reduce further the already small percentage of transaction time used by the server in executing the concurrency control algorithm, and would serve only to reinforce any conclusions made to an in-memory implementation.

Carey states that the parameters chosen for his simulations are not intended to duplicate those of real applications; the intention is to investigate how various multiversion concurrency control algorithms compare with one another under various conditions. We follow the same philosophy; our test conditions are not intended to be realistic, but are intended to provide a fair comparison between the different concurrency control servers. However, the relatively small concurrency control decision costs lead us to hope that experimental evidence will show that our adaptable type-specific concurrency control scheme for abstract data types has an acceptable performance.

In generating experimental evidence, we first had to decide on appropriate performance metrics for meaningful comparisons of the different severs. Several workers have shown that the *level of conflict* is the most dominant

[3] Note, however, that Reidl points out that his initial implementation of the generic data structure did cause a 20% performance penalty on the adaptable server, which he was able to remove through a careful redesign of a data structure.

performance factor in the comparison of different concurrency control schemes [2, 9, 11, 25]. Other variables such as the length and duration of transactions, the fraction of updates, and the hot-spot access fraction affect the performance of the system mainly according to how they affect the probability of conflicts encountered by the transactions. In keeping with these studies, the work presented here relies on the percentage of conflict as its performance variable. Unlike most of the other performance studies referenced, we do not present simulation results. Instead, we have actually completed implementations of the servers described here. The servers (and transactions) are implemented in the distributed high-level language SR [1] running on a SUN workstation.

As already mentioned in Section 1, we used the semiqueue as our example of a typical shared abstract data type. We initially implemented three different servers, using a purely pessimistic, a purely optimistic, and a hybrid concurrency control scheme as described in Section 2. Details of these implementations are given in [10], and an overview in Appendix A. The transactions (also coded in SR) invoke operations such as *Enq* and *Deq* on the servers of the shared objects. Some transactions will be delayed at a pessimistic server until locks are released, and some transactions may also be aborted if deadlock is detected. At an optimistic server, some transactions may be aborted if conflict is detected at validation time. At a hybrid server, transactions could be delayed or aborted at validation time. We then implemented adaptable servers that could treat new transactions either pessimistically or optimistically, using state-based or conflict-based information as described in Section 3.

To test the servers we subjected them to different workloads which were designed to establish a threshold percentage of conflict above which pessimistic schemes outperform optimistic ones. Hence we examined the relative behaviors of the optimistic, pessimistic, and hybrid concurrency control strategies as they were subjected to increasing levels of conflict.

The method we used to implement the transaction workload was to maintain peak load conditions in the system by arranging that all the transactions originate at time zero. Each new transaction in the system begins by requesting a unique transaction identification number from a transaction server, and then uses this number to identify its subsequent operation requests. Each transaction is represented as a sequence of operations to invoke; and the transactions proceed by executing one operation at a time.

In order to evaluate the respective performances of each of the three concurrency control approaches fairly, all tests were designed to achieve the same amount of work under the control of each locking scheme (i.e., within a given level of concurrency, the test for each scheme must successfully execute the same number and type of events). We therefore held the level of multiprogramming and the type of transactions constant over each experiment. This is similar to Reidl's *closed experiment* approach, wherein after one transaction completes, another is immediately started. This compares with an *open experiment* whereby transaction arrivals are separated by exponential random variables representing, for instance, arrivals of a customer at a teller.

Reidl prefers the closed technique, which provides a constant level of multiprogramming, because the results of closed experiments are consistently easier to understand and interpret than those of open experiments [25]. We also found that the experimental results were easier to understand using a fixed level of multiprogramming (100 in most of our experiments).

We conducted many experiments (of which two are detailed below) for conflicts of type D3 and D4, which involve *Deq* events. We chose these because successful *Deq* events dominate performance costs (as noted in Section 2.6.5, the cost of *Deq* is $O(n)$ while *Enq* is $O(1)$, where n is the number of items in the semiqueue). Other tests using different transaction workloads showed similar relative performance [10], bearing out Agarwal and Carey's observation that the level of conflict is a suitable measure for comparing different concurrency control techniques [2]. To eliminate the effects of variable network delays, we ran all the tests on a single SUN-3 workstation.

Both experiments have several components:

(1) performance of type-specific static pessimistic, optimistic, and hybrid servers;
(2) performance of dynamic servers in single operating mode (to measure overhead compared with the above static servers); and
(3) performance of adaptable dynamic servers changing over a range of conflict levels.

4.2 Costs of D3 Conflicts

4.2.1 *Test Environment.* This test deals with the conflict defined by D3 in Tables I, II, and III, whereby two transactions attempt to remove the same element. We had to devise a method to force this event to occur for a controlled variable percentage conflict. The basic idea behind our test method is that 100 concurrently active transactions attempt to dequeue an element from the shared semiqueue which has been initialized with x ($x < 100$) elements. Hence only x of the transactions will successfully commit; the other ($100 - x$) transactions will be delayed (pessimistic server) or aborted (optimistic server). We define the value ($100 - x$) as the percentage of conflict in our test results. However, in order to compare the performance of the pessimistic and optimistic servers, we had to ensure that the same total amount of work was done by each server, so that all the delayed and aborted transactions were eventually executed to completion. Hence we extended the basic test plan so that there are 100 concurrently active transactions, but one of them acts solely as a conflict-causing transaction, while the other 99 must successfully *Deq* (that is, perform a *Deq*( )/*Ok*(i) event) 30 elements each. The shared semiqueue always starts with 2970 (= 30*99) elements. To establish a controlled percentage p of conflict, the conflict-causing transaction (Trans[1]) acts as a "dummy" transaction and puts ($30*p$) elements in its *Deq* intentions list, so it holds *Deq*( )/*OK*(i) locks (pessimistic) or flags (optimistic) on ($30*p$) of these elements. During the test, once the desired

level of $p\%$ conflict has been achieved, Trans[1] is forcefully aborted (for testing purposes), making its previously locked elements once again eligible for dequeuing by other transactions. We tested the server's performance under various conflicts.

Execution ordering of pessimistic test for D3 conflicts:

(1) Trans[1] performs a *Deq( )/Ok(i)* event on *30*p* elements (where *p* is the percentage of conflicts being tested)—this is not included in the timing results of the tests.
(2) Trans[2] to Trans[101-p] perform (on their intentions lists) *Deq( )/Ok(i)* for 30 elements each.
(3) The remaining transactions (Trans[102-p] through Trans[100]) each attempt to *Deq* one element, but all are blocked.
(4) Trans[1] is aborted manually (for test purposes only), and blocked transactions complete their initially blocked *Deq* events, followed by 29 more *Deq* events each.
(5) All transactions (except Trans[1]) commit.

Due to the optimistic nature of this type of conflict in the hybrid server, the hybrid test for this conflict is identical to that of the optimistic server's. Optimistic and hybrid class transactions accomplish the same amount of work (although they also include some wasted work) as the pessimistic transactions in the following way:

(1) Trans[1] performs a *Deq( )/Ok(i)* event on 30*p elements (where p is the percentage of conflict being tested)—this is not included in the timing results of the tests.
(2) Trans[2] to Trans[100] perform (on their intentions lists) *Deq( )/Ok(i)* for 30 elements each.
(3) Trans[102-p] through Trans[100] attempt to validate, but all abort.
(4) Trans[1] is manually aborted (for test purposes only).
(5) The aborted transactions are redone.
(6) All transactions (except Trans[1]) successfully validate and commit.

4.2.2 *Specific Pessimistic and Optimistic Servers.* Performance data for execution of these 100 transactions on the static specific servers is shown in Figure 2. Within these tests, the pessimistic server's performance remains essentially unaffected by an increasing percentage of conflict, whereas the optimistic server's performance deteriorates linearly. The optimistic server performs better than the pessimistic at conflict levels of less than 26%.

4.2.3 *Adaptable State-Based Server.* New transactions are assigned a p-class or o-class based on the number of available elements (i.e., elements not already in a *Deq* intentions list) relative to a predefined threshold value, in that if the number of entries is less than the threshold, then new transactions are assigned a p-class, else they are assigned an o-class.

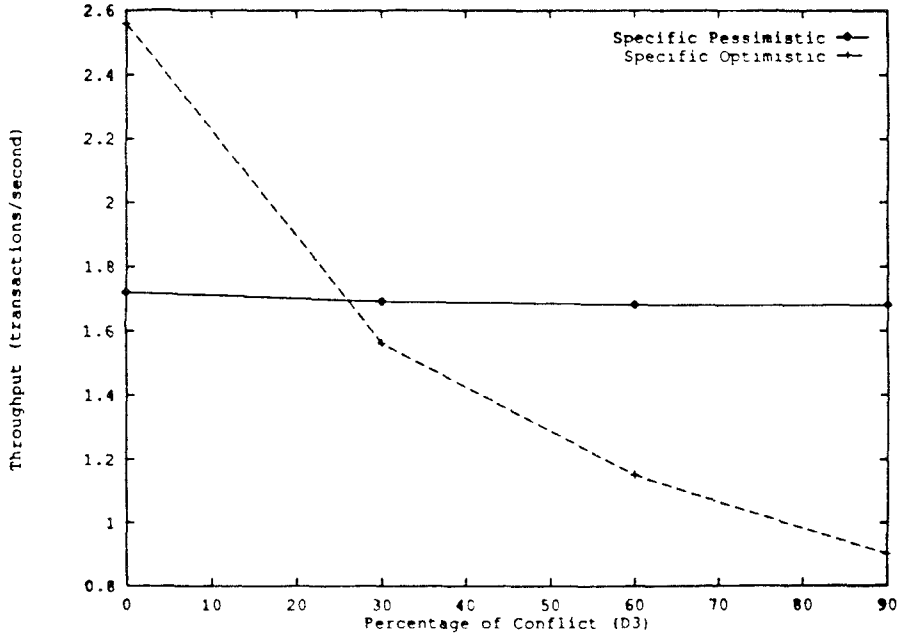

Fig. 2. Throughput for specific pessimistic and optimistic servers testing D3 conflicts.

We measured the overhead of this adaptable server by executing the same 100 transactions while forcing the server to act purely in optimistic mode (threshold = 0) and in purely pessimistic mode (threshold > 2970). This data is shown as the dashed lines in Figure 3. Comparing Figures 2 and 3, we see that the static servers are approximately 10% more efficient than the adaptable servers working in purely optimistic or pessimistic modes. As this is a compute-intensive task, for any realistic transaction system the overhead for adaptability will be less than 10% which is within an acceptable margin.

Note that the optimistic-pessimistic conflict threshold of around 20% for the adaptable servers is lower than the specific servers' conflict threshold of 26%; thus to demonstrate optimal performance over the whole range of conflicts, the dynamic state-based server needs to change from optimistic to pessimistic at approximately 20% conflict. This corresponds to a threshold value of 2370 within this test; hence, where the conflict-causing transaction removes 600 elements or more, the adaptable server operating in dynamic mode assigns the remaining 99 transactions a pessimistic class, otherwise they are assigned optimistic.

The performance of the state-based server set so that the state changes at threshold = 2370 is shown as the solid line in Figure 3. The dynamic performance has negligible overhead over the static performance when the threshold for the state-based server is accurately set to reflect an appropriate level of transaction conflict. However, the threshold at which automatic switching between optimistic and pessimistic should occur depends on the transaction

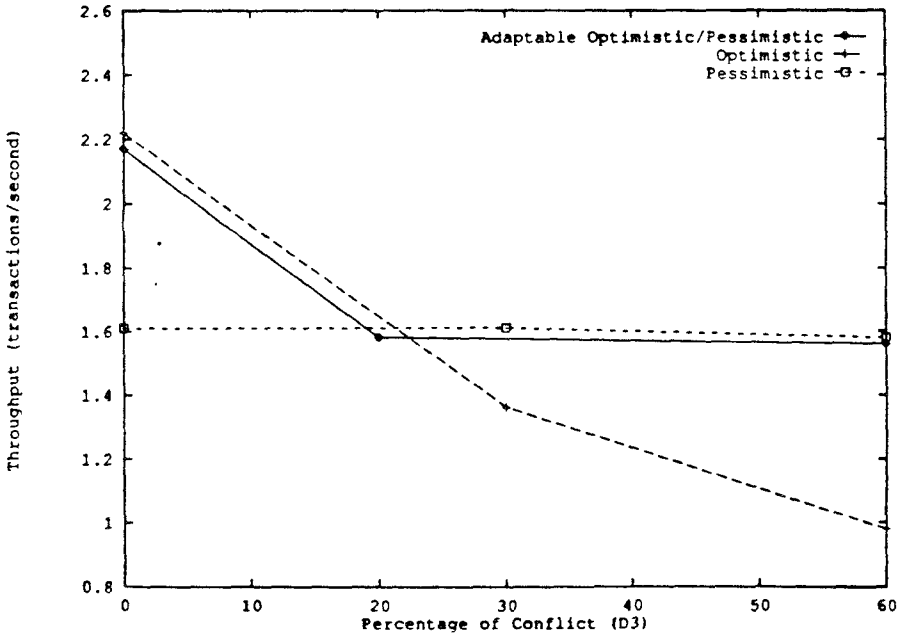

Fig. 3. Throughput for adaptable state-based server testing D3 conflicts.

access patterns. In applications without a stable transaction access pattern, a simple state-based threshold for determining the class of a new transaction cannot be used; instead the conflict crossover point (measured as 20% from the extreme threshold study) must be used at runtime. We call this the *adaptable conflict-based server*.

4.2.4 *Adaptable Conflict-Based Server*. Performance data for the overhead of the adaptable conflict-based server described in Section 3.3.2 is obtained by setting the conflict threshold to 0% and to 100% so that no changes from optimistic to pessimistic occur. The performance results are shown as the lower three lines in Figure 4. Comparing the optimistic and pessimistic data with the specific servers in Figure 2, we see there is around 10–15% overhead for this concurrency control decision, as for the state-based adaptable server.

Due to the optimistic nature of the hybrid server's treatment of D3, hybrid performance is similar to optimistic performance, but is slightly worse because this hybrid server's *Deq*( )/*Ok*(*i*) event includes the pessimistic overhead involved in its treatment of dependency D4. That is, every successful *Deq* event performed by the hybrid server must also check all active transactions for conflict-type D4. This additional pessimistic overhead in optimistically-treated conflict types adds to the per-event cost of a hybrid *Deq* event.

This figure (upper two lines in Figure 4) also shows the advantage of early abort in BOCC (where an active transaction is killed when an element held in

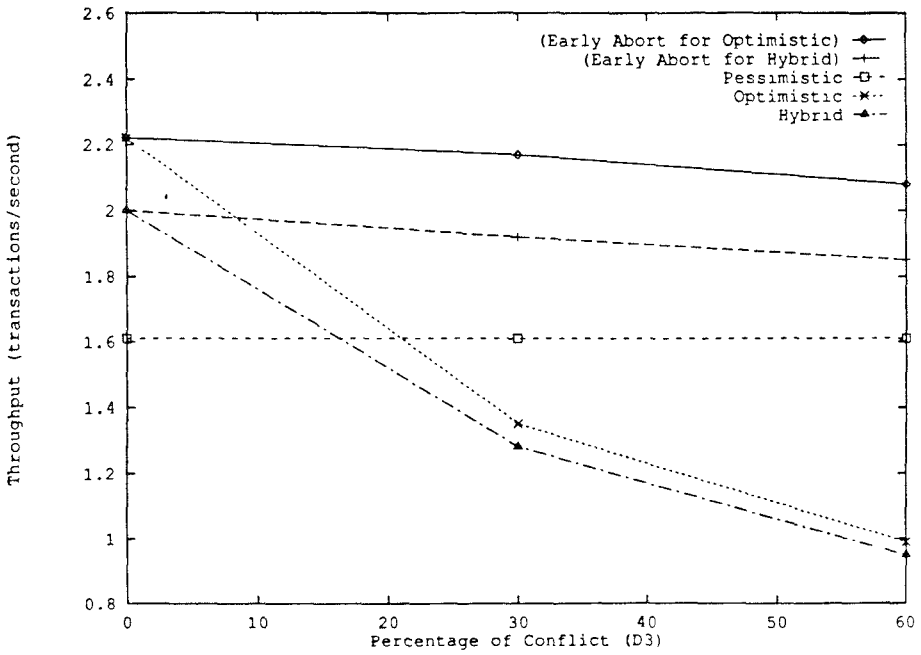

Fig. 4. Throughput for specific conflict-based servers testing D3 conflicts.

its intentions list is removed from the permanent state of the semiqueue by a validated transaction), as wasted work in the optimistic server is kept to a minimum. The tests demonstrating the potential performance benefits of early abort in BOCC represent situations where the active transactions that were killed held only one item in their *Deq* intentions list. Under these conditions, the amount of wasted work is limited to one *Deq*( )/*Ok*(*i*) event per aborted transaction. In the worst case here, however, where all 30 *Deq* events are wasted in each aborted transaction, the performance of unoptimized optimistic, and hybrid modes is very poor, as demonstrated in Figure 4. Although the enhanced server can perform much better than the unoptimized one, the improvement is very workload dependent, and in these tests, using artificial workloads, we decided to continue to use the unenhanced versions, reflecting the worst-case performance gains.

Figure 5 shows the dynamic performance results for the adaptable conflict-based server (together with a repeat of the specific optimistic and pessimistic servers' throughput from Figure 2), where o-class allocation is performed until the level of conflict for D3 is above 20%, at which point p-class is allocated to new transactions. The adaptable server has a low overhead of around 10–15%, despite the complexity of the conflict calculation; this is attributed to the relatively small percentage of time that the server spends in the concurrency control decision. If we had used a heuristic that had taken snapshots more frequently the overhead would have been higher.

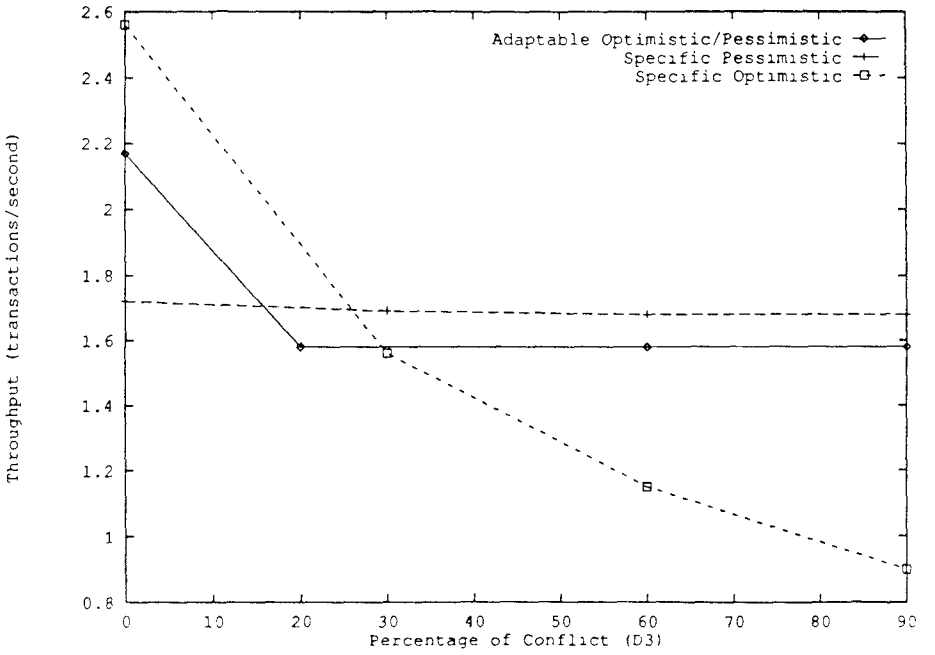

Fig. 5. Throughput for adaptable conflict-based server in dynamic mode testing D3 conflicts.

Note that the state-based server can provide as cost effective a means of control as the automatic server (the state-based server, however, does not include a hybrid option).

4.3 Tests for D4 Conflicts

4.3.1 *Test Environment.* Another conflict type tested deals with the proscribed dependency D4, whereby a *Deq* event conflicts with another active transaction executing an *Inspect* event. As for the previous tests, 100 transactions each dequeue 30 elements from the shared semiqueue. Conflict is introduced when one of the 100 active transactions executes an *Inspect* event.

4.3.2 *Static Pessimistic, Optimistic, and Hybrid Servers.* Performance data for the static, purely optimistic, purely pessimistic, and hybrid servers for this conflict type are shown in Figure 6. This figure shows that the pessimistic server for D4 conflicts behaves similarly for D3 conflicts as shown in Figure 2. However, the optimistic server for D4 conflicts gains over the same server for D3 conflicts because in the tests for conflicts defined by D3, before a transaction can set its Deq O-flag on an element that is already flagged by another transaction, it must first search the entire semiqueue for an unflagged element. Since these fruitless searches do not occur under the test conditions for conflicts defined by D4, the *Deq*( )/*OK*(*i*) events are less expensive, and consequently the costs of redoing them are less. This is

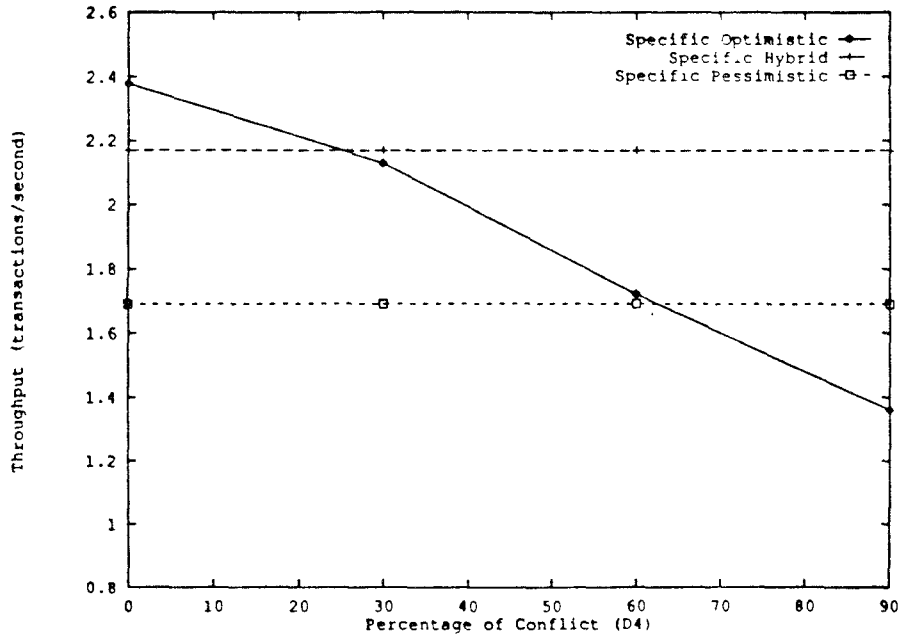

Fig. 6. Throughput for specific optimistic, pessimistic, and hybrid servers testing D4 conflict.

demonstrated by comparing the performance of the optimistic servers at 60% in Figure 2 and Figure 6. Under these conditions, the percentage of conflict representing the threshold between optimistic and pessimistic performance for the D4 conflict is approximately 60%.

As the level of conflict increases, hybrid performance exhibits the constant behavior characteristic of a pessimistically treated conflict type, but it outperforms the pessimistic technique. This is because within the hybrid strategy, a *Deq* event incorporates less pessimistic overhead as a result of its optimistic treatment of the other two conflict types involving *Deq* operations (D2 and D3). (But recall that in the previous example, the hybrid server exhibited behaviour similar to the optimistic server.)

4.3.3 *Adaptable State-Based Server.* The state-based server would not change state on this kind of conflict, so its performance reflects either purely optimistic or purely pessimistic results, according to the threshold.

4.3.4 *Adaptable Conflict-Based Server.* Figure 7 shows the performance of the adaptable server with conflicts set so that no changes occur—either all are pessimistic or all are optimistic. Comparing this with the data in the previous figure, we see that the adaptable server has $< 10\%$ performance penalty over the specific servers. This figure also shows that the percentage of conflict representing the threshold between optimistic and pessimistic performance is approximately 45% for the D4 conflict-type and the threshold between optimistic and hybrid performance is approximately 15%.

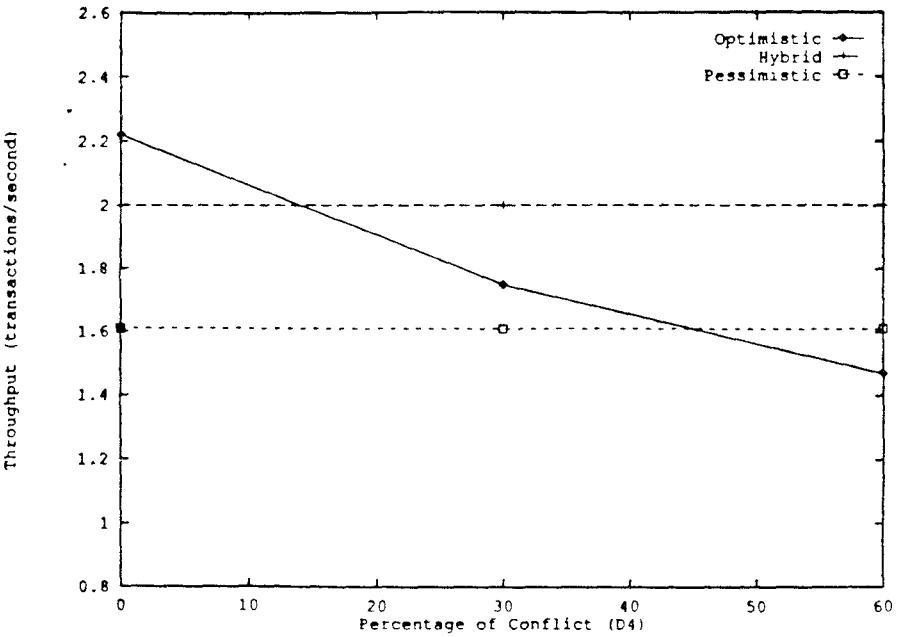

Fig. 7. Throughput for conflict-based server in static mode testing D4 conflicts.

Results for the conflict-based server operating in dynamic mode are shown in Figure 8.

The solid line in Figure 8 illustrates that the performance of the dynamic server set to change from optimistic to hybrid when the D4 conflicts are at 20% (as specified in Figure 1 in Section 3.3.2). The dashed line shows that the performance of the dynamic server set to change from pessimistic to hybrid at D4 > 20% when there are more than 20% of D3 conflicts. This performance is close to optimal for the entire range of D3 and D4 conflicts.

4.4 Analysis and Evaluation

The tests presented here are highly event intensive, so subtle difference in performance costs incurred by the respective conflict resolution strategies are essentially negligible. For example, the difference between FOCC and BOCC validation is small, aside from the potential performance enhancement associated with BOCC's early abort strategy. Also, the additional cost of our type-specific Wake-up calls as the percentage of conflict increases in a pessimistic scheme are negligible.

As anticipated, the optimistic strategy is most cost effective at low levels of conflict, while the pessimistic strategy is most efficient at high levels. And in environments where the level of conflict reliably varies on a conflict-type basis, the hybrid strategy can provide the most cost-effective control. Our dynamic servers combine all the best features of optimistic, pessimistic, and hybrid control, so dynamic servers are most effective when the conflict levels

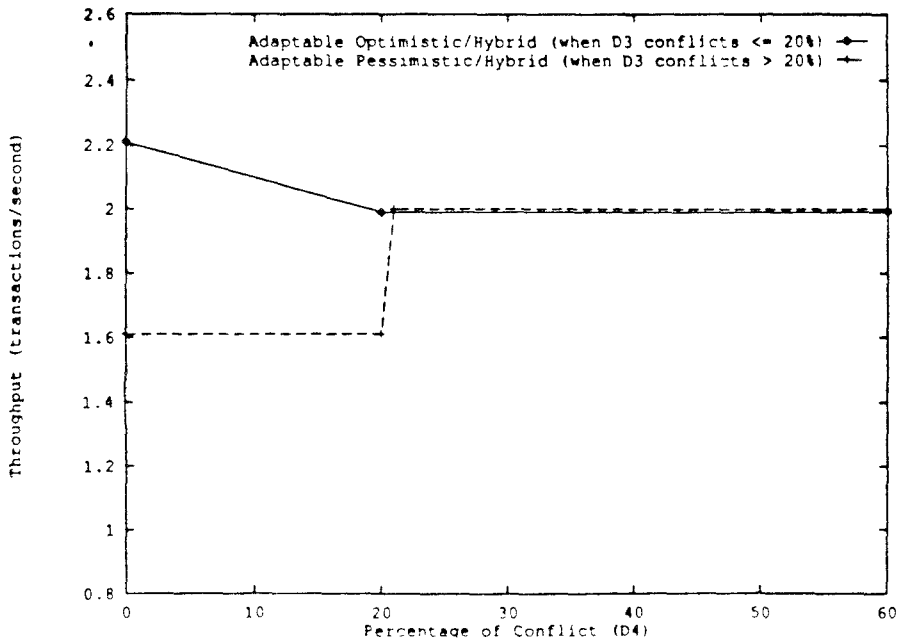

Fig. 8. Throughput for adaptable conflict-based server testing D3 and D4 conflicts.

vary, as the most appropriate strategy is selected according to the state threshold or the current conflict level.

Using the threshold levels established by these tests, we verify that the adaptable conflict-based server is most effective if, for both conflict-types D3 and D4 the level of conflict is below 20%, an optimistic strategy is used. In an environment where D3 is above 20%, the pessimistic strategy can provide the best control (based on worst-case early abort results). Furthermore, when the level of D3 is below 20% but D4 is above 20%, the hybrid strategy can provide the most effective means of control.[4]

5. DESIGN METHODOLOGY

5.1 General Approach

Our strategy for implementing an efficient adaptable server for shared abstract objects by finding conflict levels at which the server should switch modes (or the state threshold at which the state-based server should switch) is readily extended to any shared abstract object. The activities which should be performed are described below and illustrated with another example, that of a shared B-tree. Another example, that of a shared directory object, is given in [10].

[4] Note that these conflict thresholds would be even higher, if the pessimistic deadlock prevention was not optimized.

(1) Identify all the events (operations + outcomes).
(2) Identify which pairs of events cause conflicts in an optimistic environment by deriving proscribed serial dependency relations. Identify additional pairs of events for conflicts in a pessimistic environment (must be symmetrical).
(3) Design acceptable representation of intentions lists, according to the demands of the application.
(4) Consider optimistic and pessimistic performance enhancements such as type-specific wake-up calls and deadlock detection (pessimistic servers) and BOCC with early abort (optimistic servers). Enhancements made to optimistic servers will push the conflict threshold higher; those to pessimistic servers lower.
(5) Implement a static optimistic server, a static pessimistic server, and any appropriate hybrid servers.
(6) Measure the performance of these static servers for each conflict type to locate the threshold conflict levels.
(7) Implement a dynamic adaptable server by combining the static implementations with a dynamic strategy (state-based, automatic, or preset) and augmenting the static validation and commitment procedures with the suicidal and commit-and-kill strategies, respectively.
(8) As in (7), measure the performance of the dynamic servers in a purely optimistic, a purely pessimistic, and appropriate hybrid modes, to locate the conflict thresholds. Compare these results with the purely static servers to confirm that the dynamic overhead is acceptable (say $< 10\%$).
(9) Allow the state-based or automatic dynamic servers to switch modes at the located thresholds for optimum performance and provide for preset transaction classes if required.

5.2 Example: Shared B-Tree Atomic Data Type

(1) Identify all the events (operations + outcomes).

We consider a B-tree index as a shared abstract object within a distributed transaction system. The possible events associated with this object could be:

```
Insert(n)/Ok( )
Delete(n)/Ok( )
Delete(n)/Failed( )
Search(n)/Ok( )
Search(n)/Failed( )
```

(2) We illustrate the conflicts between pairs of events in Table V.

Conflicts D1 through D4 are for optimistic and pessimistic transactions, whereas D5, D6, and D7 are for pessimistic transactions only. Row 1 shows that Inserting an entry conflicts with an unsuccessful delete of the same entry (D1) and an unsuccessful search for the entry (D2). Row 2 shows that two transactions cannot delete the same entry (D3) and that an attempt to delete an entry conflicts with a successful search for the

Table V. Optimistic (O) and Pessimistic (P) Conflicts at a B-Tree

	Insert(n)/OK()	Delete(n)/OK()	Delete(n)/Failed()	Search(n)/OK()	Search(n)/Failed()
Insert(n)/OK()			D1(O&P)		D2(O&P)
Delete(n)/OK()		D3(O&P)		D4(O&P)	
Delete(n)/Failed()	D5(P)				
Search(n)/OK()		D7(P)			
Search(n)/Failed()	D6(P)				

entry (D4). Row 3 shows that a pessimistic transaction cannot have failed to delete an entry which has been added by another transaction (D5); row 4 shows that a successful search for an item must wait if another pessimistic transaction has executed a successful dequeue of the same item (D7) and the last row shows that an unsuccessful search for an item conflicts with another transaction which has added the same item.

(3) The intentions list data structure has several alternatives. For example, in some applications transactions accessing the B-tree may actually perform a relatively high number of each of the event-types before committing. In such a situation, intentions lists may best be handled by building local B-trees as intentions lists. That is, all *Insert(n)/Ok( )* events performed by a transaction Ti would be inflicted upon a B-tree structure that is local to Ti. Attempts by Ti to delete or search would then start by checking the local structure before accessing the global object. An advantage to this approach would be that the same procedures used to access the global object could be applied at the local level. Of course, a disadvantage would be that the costs involved in maintaining these local B-trees may be prohibitive in applications where transactions perform relatively few events. In such situations, intentions lists may best be handled simply by building local linked lists for newly inserted nodes.

(4) *Optimistic and Pessimistic Performance Enhancements. Optimistic Enhancements*: *BOCC + Early Abort*. As with the semiqueue, in the case of the conflict type between concurrent transactions attempting to delete the same node from the B-tree (D3), the BOCC technique can rely on an optimistic early abort strategy. This way, the first transaction to successfully validate will actually remove the conflict-causing node from the permanent state of the B-tree, and all active transactions that include that node in their respective intentions lists will be aborted. This approach is made possible by virtue of that fact that all transactions attempting to delete a given node are identified within optimistic flag information associated with that node, a useful generalization of this optimistic enhancement.

Pessimistic Enhancements: *Wakeup Calls*. Pessimistic performance enhancements include wake-up calls associated with the release of P-locks and type-specific deadlock strategies. In the case of the B-tree events, the release of an *Insert*(*n*)/*Ok*( ) P-lock should send wake-up messages to all blocked *Delete*(*n*)/*Failed*( ) and *Search*(*n*)/*Failed*( ) events (D1, D2) and vice versa (D5, D6), and the release of a *Delete*(*n*)/*Ok*( ) P-lock should send wake-up messages to all blocked *Delete*(*n*)/*Ok*( ) and *Search*(*n*)/*Ok*( ) events (D3, D4) and vice versa (D7).

Pessimistic Enhancements: *Type-Specific Deadlock Detection*. Unlike the case of the semiqueue server where all circular waits must contain a cycle of length two, circular waits among transactions accessing the B-tree server may involve more than two transactions. For example:

T1:Insert(n1)/Ok( )
T2:Delete(n2)/Ok( )
T3:Search(n3)/Failed( )
T1:Insert(n3)—BLOCKED, waiting for wake-up from T3 (Conflict D2)
T2:Search(n1)—BLOCKED, waiting for wake-up from T1 (Conflict D6)
T3:Search(n2)—BLOCKED, waiting for wake-up from T2 (Conflict D7)

Consequently, the algorithm presented in Section 2.6.4 for type-specific deadlock detection in the semiqueue wherein all deadlocks must involve a cycle of two transactions is not adequate for the B-tree server. Instead a less efficient mechanism must be employed, causing a relative increase in overhead associated with the pessimistic component of the B-tree server.

Rules (6)–(9) must then be followed: implement static servers to determine conflict thresholds, dynamic servers with interclass conflicts to determine the overhead, and, if acceptable, implement an adaptable server.

Inspection of the specification and implementation design for the B-tree reveals that its implementation should perform similarly to the semiqueue, with conflict thresholds from optimistic to pessimistic of around 20% or a little higher (because the B-tree deadlock prevention code is not as efficient as the semiqueue's) for the most expensive operations (Delete and Insert).

6. CONCLUSIONS AND FUTURE WORK

6.1 Summary

These implementations and performance studies show the feasibility and practicality of integrating both optimistic and pessimistic concurrency control in servers for shared objects, where semantic information is used to identify conflicting events between concurrent transactions.

Herlihy proposed static servers incorporating a hybrid concurrency control scheme in [16]; we have extended this work to include the facility to dynamically adjust the mode in which a server treats conflicting events. This is especially useful in the situations where the conflict levels in the server's environment vary. We have described three practical methods for implement-

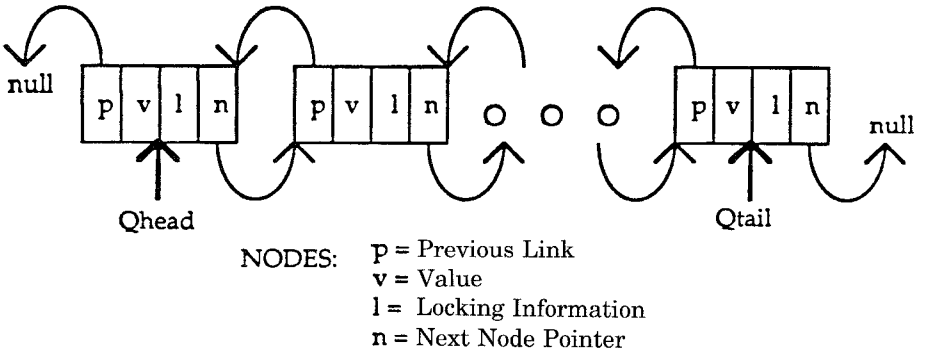

Fig. 9. The shared semiqueue.

ing these so-called dynamic servers; semantic information on the likely use of the server can be used to identify which technique is most cost-effective.

One method is based on the server's state information, allowing the server (the state-based server) to choose optimistic or pessimistic strategies for concurrency control. Another (the conflict-based server) uses the conflict history at the server, allowing it to choose optimistic, hybrid, or pessimistic strategies for concurrency control according to the percentage of conflicts encountered. The third method uses preset classes for each transaction, optimistic, hybrid, or pessimistic, which the server must use in resolving this transaction's conflicts. The preset method can be used in conjunction with either of the other two methods.

Furthermore, we have shown how semantic information can be used to enhance the performance of pessimistic type-specific deadlock prevention and unblocking strategies and how "early-abort" strategies can improve the performance of optimistic execution.

6.2 Future Work

Our future plan is to integrate adaptable servers for different types of shared objects into a distributed database and then to test the system with real transactions.

APPENDIX A: OVERVIEW OF SEMIQUEUE IMPLEMENTATION

Our implementation is in Version 1 of the SR programming language running under the Sun UNIX 4.2 operating system (release 3.4). Three different servers were implemented to enforce concurrency control on a shared semiqueue (called specific servers). The specific semiqueue servers (optimistic, pessimistic, and hybrid) use a doubly-linked list for the shared semiqueue's elements. Each node in this list contains an integer value (since this is a semiqueue of integers) and the appropriate optimistic and pessimistic locking information. *Qhead* and *Qtail* mark the beginning and end of the semiqueue, respectively (Figure 9).

Every active transaction is identifiable by a unique transaction number, *Tnum*. For every *Tnum* that accesses an object, the object's server maintains

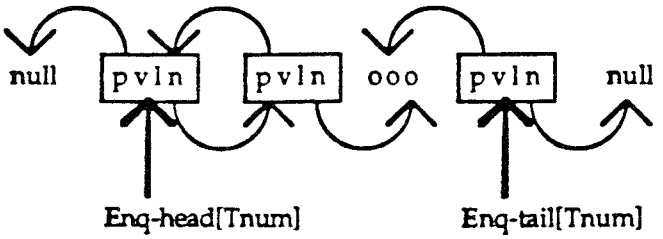

Fig. 10. Enq intentions list.

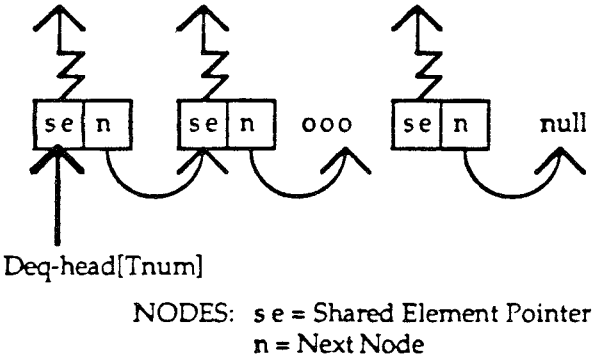

Fig. 11. Deq intentions list.

an intentions list with two components: an enqueue (*Enq*) list and a dequeue (*Deq*) list. The enqueue component consists of a doubly-linked list of elements to be enqueued, and the dequeue component consists of a singly-linked list of pointers to the shared elements of the semiqueue (Figure 10 and Figure 11). When a transaction successfully commits, the *Enq* list is attached to the end of the shared semiqueue, *Qtail* is updated, and the appropriate elements associated with the *Deq* component of the transaction's intentions list are removed from the semiqueue in an atomic action. In the event of transaction abort, however, these lists are discarded. The nodes of the enqueue component of each transaction's intentions list are identical to those of the shared semiqueue (Figure 9). This facilitates the method of appending the entire Enq intentions list at commit time.

APPENDIX B: TYPE-SPECIFIC DEADLOCK DETECTION

Given Ti that is about to block on Tj, a simple type-specific prevention mechanism enforces the following rule:

Deadlock Rule. For Ti to block on Tj, Tj cannot already be blocked on a lock that Ti possesses.

PROOF. Let T be the set of active transactions that have completed at least one event and are not blocked. All completed events performed by the transactions in T must compatible. That is, there cannot be any conflicts

among these completed events. Combinations of compatible event types can be selected from one of the two sets R or W, where

R = {*Inspect*( )/*Ok*(*#items*), *Deq*( )/*Failed*( )}
W = {*Enq*(*i*)/*Ok*( ), *Deq*( )/*Ok*(*j*)}

In the case where all the events completed by transactions in T are from the set R, transactions in T may take on one of three forms:

(1) *Inspect*( )/*Ok*(*items*) event type only: In this case, in order for one transaction in T, Ti, to incur conflict, Ti must attempt an event type from the set W. Ti will then be blocked on all other transactions in T, since they all hold Inspect P-locks and both event types from W conflict with these P-locks. Any subsequent transaction Tj that attempts an event from the set W must also block on all other transactions in T, including Ti. Thus, all circular waits must include a cycle of length two.

(2) Deq( )/Failed( ) event type only: In this case, in order for one transaction Ti to incur conflict, Ti must attempt an Enq(i)/Ok( ) event type from the set W, as the Deq( )/Ok(i) event-type is not possible on an empty semiqueue. Again, Ti will then be blocked on all other transactions in T and any subsequent transaction Tj that also becomes blocked will necessarily be blocked on Ti, forcing all circular waits to include a cycle of length two.

(3) A mixture of Deq( )/Failed( ) and Inspect( )/Ok(*items*) event-types. As with the previous case, in order for one transaction Ti to incur conflict, Ti must attempt an Enq(i)/Ok( ) event type from the set W, since again the Deq(i)/Ok( ) event type is not possible on an empty semiqueue. As above, this would force Ti to be blocked on all transactions in T and force all subsequent conflicts to include a cycle of length two.

In the case where all the events completed by transactions in T are from the set W, transactions in T may take on one of three forms:

(1) Enq(i)/Ok( ) events only: In this case, in order for one transaction Ti to incur conflict, Ti must attempt an event type from the set R. Ti will then be blocked on all other transactions in T, and any subsequent transaction Tj that attempts an event type from the set R will also block on all other transactions in T, including Ti, forcing all circular waits to include a cycle of length two.

(2) Deq( )/Ok(i) events only: In this case, in order for one transaction Ti to incur conflict, Ti must attempt either an Inspect( )/Ok(*items*) or a Deq( )/Ok(i) event type. The Deq( )/Ok(i) event causes a conflict due to the fact that although there are items still in the permanent state of the semiqueue, they have all been Deq( )/Ok(i) P-locked and are not available for dequeuing. As a result, Ti will again be blocked on all other transactions in T and any subsequent transaction Tj that blocks will necessarily be blocked on Ti as well, forcing a circular wait of length two.

(3) A mixture of Enq(i)/Ok( ) and Deq( )/Ok(i) event types: Again, in this case, a transaction Ti may incur conflict either by attempting an Inspect( )/Ok(*items*) or a Deq( )/Ok(i) event type. As with all previous cases, if Ti blocks due to a conflict with an Inspect( )/Ok(*items*) event type, Ti will block on all transactions in T and any subsequent transaction Tj that also blocks as the result of an Inspect( )/Ok(*items*) conflict will necessarily be blocked on Ti as well, forming a circular wait of length two. However, this final case is different from the rest in that a Deq( )/Ok(i) conflict will only form among transactions that have completed Deq( )/Ok(i) event types. In this case, once all of the items in the semiqueue have been P-locked for dequeuing, all subsequent dequeue attempts from any transaction, Tj, are blocked only on the transactions in T that have completed Deq( )/Ok(i) event-types. Tj is not blocked however, on any transactions in T that have only performed Enq(i)/Ok( ) event-types. Thus, a situation may arise where Ti may have completed at least one Enq(i)/Ok( ) event-type followed by an attempted Inspect ( )/Ok(*items*) event-type, becoming blocked on all other transactions in T. But T may include a transaction, Tj, that becomes blocked on Deq(i)/Ok ( ) conflict. Here, unlike all previous cases, although Ti is blocked on Tj, Tj is not blocked on Ti. Circular waits, however, must include a transaction, Tk, that becomes blocked on Ti. By virtue of the fact that the only conflict-type that will block on Ti will be an attempted Inspect ( )/Ok(*items*) event-type, and Ti is blocked on all transactions in T, Ti will necessarily be blocked on any transaction, Tk, that attempts an Inspect( )/Ok(*items*) event-type. Consequently, all circular waits will necessarily include a cycle of length two. □

REFERENCES

1. ANDREWS, G. R., OLSSON, R. A., COFFIN, R. A., ELSHOFF, M. I. NILSEN, I J. P., AND PURDIN, T. An overview of the SR language and implementation. *ACM Trans Program. Lang. Syst. 10*, 1 (Jan. 1988), 51–86.
2. AGRAWAL, R , CAREY, M., AND LIVNY, M Concurrency control performance modelling. Alternatives and implications. *ACM Trans. Database Syst. 12* 4 (Dec. 1987), 609–654.
3. BERNSTEIN, P., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.
4. BHARGAVA, B., RIEDL, J., AND WEBER, D. An expert system to control an adaptable distributed database system. Tech. Report CSD-TR-693, Purdue Univ., May 1987.
5. BHARGAVA, B., AND RIEDL, J. The Raid distributed database system. *IEEE Trans. Softw. Eng. 16*, 6 (June 1989), 726–736.
6. BHARGAVA, B., AND RIEDL, J. A model for adaptable systems for transaction processing *IEEE Trans. Knowl. Data Eng. 1*, 4 (Dec. 1989), 433–449.
7. BORAL, H., AND GOLD, I. Towards a self-adapting centralized concurrency control algorithm, in *Proceedings of the ACM SIGMOD Conference* (1984), pp. 18–32.
8. CAREY, M., AND LIVNY, M. Models for studying concurrency control performance· Alternatives and implications. in *Proceedings of the ACM SIGMOD Conference* (1985), pp. 108–121.
9. CAREY, M., AND MUHANNA, W. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst. 4*, 4 (Nov. 1986), 338–378.
10. COADY, Y. An investigation of type-specific optimistic, pessimistic, and hybrid concurrency control. M.Sc. thesis, School of Computing Science, Simon Fraser Univ., Dec. 1988.
11. CORDON, R., AND GARCIA-MOLINA, H. The performance of a concurrency control mechanism

that exploits semantic knowledge. in *Proceedings of the Fifth International Conference on Distributed Computing Science* (1985), pp. 350–358.
12. ESWAREN, K. P. GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11, (Nov. 1976), 624–633.
13. GARCIA-MOLINA, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst. 8*, 2 (June 1983), 186–213.
14. GAWLICK, D. Processing "Hot spots" in high performance systems. in *Proceedings IEEE COMPCON Conference* (Feb. 1985), pp. 249–251.
15. HARDER, T. Observations on optimistic concurrency control schemes. *Inf. Syst. 9*, (June 1984), 111–120.
16. HERLIHY, M. Optimistic concurrency control for abstract data types. in *Proceedings of the Principles of Distributed Computing Conference* (Aug. 11–13, 1986). ACM, New York, pp. 206–217.
17. HERLIHY, M. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. Database Syst. 15*, 1 (Mar. 1990), 96–124.
18. HERLIHY, M., AND WEIHL, W. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM-SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)* (Austin, Tex., Mar. 21–23, 1988), pp. 201–210.
19. KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6* 2, (June 1981), 213–226.
20. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
21. LAUSEN, G. Concurrency control in database systems: A step towards the integration of optimistic methods and locking. In *Proceedings of the ACM Conference* (Dallas, Tex., Oct. 25–27, 1982). ACM, New York, pp. 64–68.
22. LAUSEN , G. Integrated concurrency control in shared B-trees. *Comput. 33*, (1984), 13–26.
23. LISKOV, B., AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust distributed programs. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 381–404.
24. REED, D. R. Naming and synchronization in a decentralized computer system. PhD. thesis, Laboratory for Computer Science, MIT, MIT/LCS/TR-205, 1978.
25. RIEDL, J. Adaptable distributed transaction systems. Ph.D. thesis, Computer Science Dept. Purdue Univ., May 1990.
26. ROBINSON, J. T. Design of concurrency controls for transaction processing systems. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon University CMU-CS-82-114, April, 1982.
27. ROWE, L., AND STONEBRAKER, M. The commercial INGRES epilogue. In *The INGRES papers: Anatomy of a Relational Database System*. Addison-Wesley, Reading Mass., 1986.
28. SCHWARTZ, P. M., AND SPECTOR, A. Z. Synchronizing shared abstract types. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 223–250.
29. SHETH, A., AND LIU M. Integrating locking and optimistic concurrency control in distributed database systems. in *Proceedings of the 6th International Conference on Distributed Computing Systems* (May 1986), pp. 89–99.
30. SPECTOR, A. Z. Distributed transactions for reliable systems. In *Proceedings of the Principles of Distributed Computing Conference* (1985).
31. SPECTOR, A. Z. Distributed processing and the camelot system. Carnegie-Melon Univ., TR-87-100, Jan. 1987.
32. STONEBRAKER, M., AND ROWE, L. The design of POSTGRES. In *Proceedings of the ACM SIGMOD Conference* (Washington, D.C., May 28–30, 1986), pp. 340–355.
33. WEIHL, W. Specification and Implementation of Atomic Data Types. Ph.D. thesis, MIT Laboratory for Computer Science, MIT/LCS/TR-314, Apr. 1984.
34. WEIHL, W. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst. 11* 2 (Apr. 1989), 249–282.
35. WEIHL, W. The impact of recovery on concurrency control. In *Proceedings of the ACM Conference on the Principles of Databases (PODS)* (Philadelphia, Pa., Mar. 29–31, 1989), pp. 259–269.

Received March 1990; revised August 1990; accepted March 1991