Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types

MAURICE HERLIHY
Carnegie Mellon University

An optimistic concurrency control technique is one that allows transactions to execute without synchronization, relying on commit-time validation to ensure serializability. Several new optimistic concurrency control techniques for objects in decentralized distributed systems are described here, their correctness and optimality properties are proved, and the circumstances under which each is likely to be useful are characterized.

Unlike many methods that classify operations only as Reads or Writes, these techniques systematically exploit type-specific properties of objects to validate more interleavings. Necessary and sufficient validation conditions can be derived directly from an object's data type specification. These techniques are also modular: they can be applied selectively on a per-object (or even per-operation) basis in conjunction with standard pessimistic techniques such as two-phase locking, permitting optimistic methods to be introduced exactly where they will be most effective.

These techniques can be used to reduce the algorithmic complexity of achieving high levels of concurrency, since certain scheduling decisions that are NP-complete for pessimistic schedulers can be validated after the fact in time, independent of the level of concurrency. These techniques can also enhance the availability of replicated data, circumventing certain tradeoffs between concurrency and availability imposed by comparable pessimistic techniques.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, data types and structures*; D.4.3 [**Operating Systems**]: File Systems Management—*distributed file systems*; D.4.5. [**Operating Systems**]: Reliability—*fault-tolerance;* H.2.4 [**Database Management**]: Systems—*distributed systems*; *transaction processing*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Abstract data types, optimistic concurrency control, replication

This research was sponsored by the Department of Defense Advanced Research Projects Agency, ARPA Order 4976, monitored by the Air Force Avionics Laboratory under contract F33615-84-K-1520.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

A preliminary version of this paper appeared in the *Proceedings of the 5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Aug. 1986), pp. 206–217.

Author's current address: Digital Equipment Corporation, Cambridge Research Lab., 1 Kendall Square, Building 700, Cambridge, MA 02139.

1. INTRODUCTION

Optimistic concurrency control is based on the premise that it is sometimes easier to apologize than to ask permission. Transactions execute without synchronization, but before a transaction is allowed to commit, it is *validated* to ensure that it preserves atomicity. If validation succeeds, the transaction commits; otherwise the transaction is aborted and restarted. This paper proposes new optimistic concurrency control techniques for objects in distributed systems, proves their correctness and optimality properties, and characterizes the circumstances under which each is likely to be useful.

In conventional optimistic techniques, operations are classified simply as Reads or Writes, and transactions are validated by analyzing Read/Write conflicts between concurrent transactions. These techniques are intended primarily for applications where most concurrent accesses to data are Reads; they are poorly suited for general-purpose applications such as banking or reservations where concurrent Write operations occur frequently at "hot spots" such as counters, account balances, or queues. A novel aspect of the techniques proposed here is that they validate more interleavings by systematically exploiting type-specific properties of objects to recognize when concurrent Write operations need not conflict. An object's validation conditions can be derived directly from its data type specification, and the derivation technique is applicable to objects of arbitrary type. These techniques are optimal in the sense that no method using the same information can validate more interleavings.

Any optimistic scheme, however clever, is cost effective only if validation succeeds sufficiently often. Numerous studies have shown that the success rate of validation depends critically on the nature and frequency of transaction conflict. In administratively decentralized distributed systems, it is reasonable to expect that different subsystems will have different patterns of conflict and that those patterns may change over time. For example, consider a distributed system that supports transactions that span databases maintained by different companies, say, reserving an airline seat and transferring funds to pay for the ticket. Such transactions are possible only if the bank and the airline have agreed to use compatible concurrency control mechanisms. Under these circumstances, however, an optimistic technique is unlikely to be chosen as the system-wide standard. Even if the airline's transaction mix favors optimistic techniques, the bank's may not, and neither institution has any control over the other's transaction mix. These observations suggest that optimistic techniques are more likely to be useful if they can be applied locally, perhaps allowing the airline to use optimistic techniques within its own database while the bank uses pessimistic techniques. (See Lausen [29] and Boral and Gold [7] for similar arguments). Even for pessimistic techniques, however, the compatibility of distinct mechanisms is a nontrivial question. For example, two-phase locking [13] and multiversion timestamping [34] cannot be used together in a single system, because they may serialize transactions in incompatible orders. A novel aspect of the techniques proposed here is that they are compatible with a large class of standard pessimistic techniques, including two-phase locking; thus they can be applied selectively on a per-object (or even per-operation) basis exactly where they are most cost effective.

The techniques proposed in this paper permit a substantial reduction in the algorithmic complexity of achieving high levels of concurrency. In general, if a pessimistic scheduler takes full advantage of available information, then each scheduling decision may require time NP-complete in the number of concurrent transactions. By contrast, an optimistic scheduler can validate each such decision in time independent of the level of concurrency, suggesting that optimistic techniques are a promising approach to implementing highly concurrent atomic objects.

These techniques can also be used to enhance the availability of quorum-consensus replication [20]. Under pessimistic techniques, restrictions on availability and concurrency are not independent; weakening restrictions on concurrency may require strengthening restrictions on the availability [21]. The optimistic techniques proposed here circumvent this tradeoff: enhancing validation to accept more interleavings has no effect on availability, suggesting that optimistic techniques are also promising for highly concurrent replicated objects.

This paper is organized as follows. Section 2 surveys some related work, and Section 3 describes our model of computation. Section 4 describes several forms of *conflict-based* validation, a simple validation technique based on predefined conflicts. Section 5 describes *state-based* validation, a more complex scheme that validates additional interleavings by exploiting knowledge about the object's state. Section 6 examines how optimistic techniques affect the availability of replicated objects, and Section 7 closes with a discussion.

2. RELATED WORK

Perhaps the earliest concurrency control scheme to use validation is that of Thomas [41]. Kung and Robinson [27] have proposed a centralized optimistic method based on Read/Write conflicts. Ceri and Owicki [9] have extended Kung and Robinson's method to permit validation in distributed systems. Reimer [35] presented an optimistic scheme that uses validation to solve the "phantom record" problem. Lausen [29] has proposed a centralized optimistic scheme integrating two-phase locking with Kung and Robinson's scheme and has also shown that several general formulations of the validation problem are NP complete [30]. Boral and Gold [7] presented a scheme that permits a system to adapt to changing loads by shifting dynamically between pessimistic and optimistic techniques. Agrawal et al. [2] have proposed a multiversion technique in which update transactions are validated against one another, but Read-only transactions need not be validated. Härder [19] makes a useful distinction between *backward* validation, in which each transaction checks that its own results have not been invalidated by concurrent transactions, and *forward* validation, in which each transaction checks that its own effects will not invalidate any concurrent transaction's results. The distributed validation protocol used in this paper generalizes Kung and Robinson's centralized transaction numbering scheme, and it is simpler and requires fewer messages than that of Ceri and Owicki [9].

IMS Fast Path [16] uses an optimistic technique for shared counters. Like the more general techniques proposed in this paper, this technique mixes pessimistic and optimistic techniques and exploits type-specific properties of counters

to make validation more effective. This approach is discussed further in Section 5.3.

Numerous studies have compared the performance of pessimistic and optimistic techniques [1, 4, 8, 14, 31, 40]. These studies have yielded a variety of conclusions. Agrawal et al. [3] have analyzed a queuing model that encompasses many of these earlier studies, showing how the effectiveness of optimistic techniques depends in complex ways on a number of simulation parameters, including the database size, the distribution of transaction lengths, the resources available, and others. We interpret these results as suggesting that traditional optimistic techniques are not well suited to large, administratively decentralized distributed systems, where such parameters may be difficult to predict and subject to local variance. Instead, we suggest that optimistic techniques are most likely to be useful if they can be applied to individual objects (or subsystems) rather than monolithically to the entire system.

Pessimistic concurrency control techniques that exploit type-specific properties of objects include those proposed by Korth [26], Bernstein, et al. [6], Schwarz and Spector [37]. Weihl [42–44], Shasha and Goodman [39], and Herlihy [22, 24, 23].

Weihl [43, 44] has developed analytic techniques for characterizing when atomicity mechanisms are compatible. The techniques proposed here satisfy *hybrid atomicity* and are compatible with a wide variety of pessimistic techniques, including two-phase locking [13, 26, 32], as well as schemes that combine locking with timestamps [10, 11, 21, 24].

3. MODEL

3.1. Objects and Histories

The basic containers for data are called *objects*. Each object has a *type*, which defines a set of possible *values* and a set of primitive *operations* that provide the only means to create and manipulate objects of that type. For example, a bank account might be represented by an object of type Account whose value is a nonnegative dollar amount, initially zero. The Account data type provides credit and debit operations. Credit increments the account balance:

```
Credit = Operation(sum: Dollar).
```

Debit attempts to decrement the balance:

```
Debit = Operation(sum: Dollar) Signals (Over).
```

If the amount to be debited exceeds the balance, the invocation signals an exception, leaving the balance unchanged. For brevity, a debit that returns normally is referred to as a debit; otherwise it is an overdraft.

In the absence of failures and concurrency, a computation is modeled as a history, which is a finite sequence of operations. Histories are denoted by lowercase letters (g, h). An operation is written as $x\ op(args^*)/term(res^*)$, where x is an object name, op is an operation name, $args^*$ denotes a sequence of argument values, $term$ is a termination condition, and res^* is a sequence of results. The operation name and argument values constitute the *invocation*, and the

termination condition and result values constitute the *response*. We use "Ok" for normal responses. The object name is often omitted when it is clear from the context. For example,

a Credit($5)/Ok( )
a Credit($6)/Ok( )
a Debit($10)/Ok( )
a Debit($2)/Over( )

is a history for an Account a.

Each object has a serial specification, which defines a set of legal histories for that object. For example, the specification for an Account object consists of histories in which the balance covers any debit and fails to cover any overdrafts. An object subhistory, $h \mid x$(h at x), of a history h is the subsequence of operations in h whose object names are x. A history h involving multiple objects is *legal* if each object subhistory $h \mid x$ lies within the serial specification for x.

3.2 Transactions and Schedules

Distributed systems are subject to two kinds of faults: sites may crash, and communication links may be interrupted. A widely accepted approach to ensuring consistency in the presence of crashes and network partitions is to make the activities that manage the data *atomic*. Atomicity encompasses two properties: serializability and recoverability. *Serializability* [33] means that the execution of one activity never appears to overlap (or contain) the execution of another, while *recoverability* means that the overall effect of an activity is all or nothing: it either succeeds completely, or it has no effect. Atomic activities are called transactions. A transaction's effects become permanent when it *commits*, its effects are discarded if it *aborts*, and a transaction that has not committed or aborted is *active*. Well-known atomic commitment protocols (e.g., [12, 18]) can be used to ensure the recoverability of distributed transactions.

In the presence of failure and concurrency, an object's state is given by a *schedule*, which is a sequence of steps of the form: $\langle x\ op(args^*)/term(res^*)P \rangle$, $\langle x$ commit $P \rangle$, or $\langle x$ abort $P \rangle$, where x is an object name, *op*, *args**, *term*, and *res** are as before, and P is a transaction name. Schedules are denoted by uppercase letters (G, H). A *subschedule* of H is a subsequence of steps of H. If H is a schedule and x an object name, $H \mid x$ is the subschedule of H consisting of steps whose object names are x. If P is a transaction and S a set of transactions, $H \mid P$ and $H \mid S$ are defined analogously. The object name is often omitted when it is clear from the context.

For example, the following is a schedule for an Account a:

a Credit($5)/Ok( ) P
a Credit($6)/Ok( ) Q
a Commit P
a Debit($10)/Ok( ) Q
a Commit Q

Here, P and Q are transaction identifiers. The ordering of operations in a schedule reflects the order in which the object returned responses, not necessarily the order in which it received invocations. When a transaction commits or aborts,

news of the event propagates asynchronously through the system. A schedule's commit and abort steps represent the arrival of such news at an object (or a component of that object if it is replicated). A schedule is well formed if no transaction executes an operation after it commits and no transaction both commits and aborts. All schedules are assumed to be well formed.

(Serial) histories and (concurrent) schedules are related by the notion of *atomicity*. Let $>$ denote a total order on committed and active transactions, and let H be a schedule. The *serialization* of H in the order $>$ is the history h constructed by reordering the operations in H so that if $Q > P$, then the subsequence of operations associated with P precedes the subsequence of operations associated with Q. H is *serializable in the order* $>$ if h is legal. H is serializable if it is serializable in some order. H is *atomic* if the subschedule associated with committed transactions is serializable.

H is not necessarily atomic just because each $H \mid x$ is atomic. A property P is a *local atomicity property* [44] if H is atomic provided that each $H \mid x$ is atomic and satisfies P. The techniques proposed in this paper are based on a local atomicity property called *hybrid atomicity* [44]. As each transaction commits, it is issued a logical timestamp [28]. Each object must ensure that transactions are serializable in commit timestamp order. Our techniques are thus compatible with pessimistic methods such as two-phase locking [13, 21, 26, 32] and others [10, 11] in which transactions are serializable in the same order. Because hybrid atomicity is a local atomicity property, we henceforth focus our attention on the behavior of individual objects.

3.3 Two Validation Protocols

Internally, an object is implemented by two components: a *permanent state* that records the effects of committed transactions, and a set of *intentions* that record each active transaction's tentative changes. When a transaction commits, its intentions are applied to the permanent state. For example, a bank account's permanent state is the current balance, and its intentions records each active transaction's net credit or debit.

This section describes two protocols for validating distributed transactions. Both protocols ensure that transactions are validated in commit timestamp order and that committing transactions' intentions are applied to an object's permanent state in the same order. Both protocols are described as modifications to the standard two-phase commitment protocol [18]; it should be clear how to integrate them with multiphase nonblocking protocols [12].

The first validation protocol assigns commit timestamps at the start of the commitment protocol's first phase. To validate Q:

(1) The coordinator generates a commit timestamp $\mathrm{ts}(Q)$ from its logical clock, which is sent with the *prepare* message to each site. Each site keeps track of *tmax*, the latest logical timestamp assigned to any transaction validated at that site. If $\mathrm{ts}(Q) < tmax$, the site replies *failure*. Otherwise, the site validates the transaction locally, using techniques described below. If validation fails, it replies *failure*; otherwise it records the transaction's intentions on nonvolatile storage and replies *success*.

(2) The coordinator commits the transaction only if all sites report success. If the transaction commits, each site updates *tmax* and applies the transaction's intentions.

This protocol ensures that transactions are validated in timestamp order by aborting transactions that try to commit out of order. This protocol could be modified to permit a site to request that the protocol be restarted with a later timestamp, as in protocols proposed by Jacobson [25] and by Sinha et al. [39].

In the following alternative protocol, transactions' commit timestamps are chosen in the second phase.

(1) The coordinator sends a *prepare* message to each site. Each site validates the transaction locally, using techniques described below. If validation fails, it replies *failure*. Otherwise it records the transaction's intentions on nonvolatile storage and replies *success*.
(2) If all sites report success, the coordinator commits the transaction and chooses a commit timestamp, which it distributes with the *commit* message. If the transaction commits, each site applies the transaction's intentions.

The second protocol has the advantage that it does not abort transactions with out-of-order validation requests. Concurrent attempts at validation can cause incipient deadlocks, which are broken by timeouts. The major disadvantage of this protocol occurs in systems that mix pessimistic and optimistic techniques (see Section 4.5). The first protocol permits a transaction to release (pessimistic) read locks at the end of the prepare phase, while the second protocol requires that read locks be held for the duration of the protocol.

Both protocols permit only one transaction at a time to validate at any particular object. This restriction can be relaxed by the following optimization. Two transactions are *noninterfering* if neither transaction's intentions includes an operation that conflicts with an operation in the other's, where our notion of conflict is defined in the next section. Noninterfering transactions may prepare concurrently, but care must be taken to apply their intentions in commit timestamp order. Our correctness arguments below do not address this optimization.

The local validation techniques discussed in the remainder of this section work with either protocol. We assume only that each object validates and commits transactions in commit timestamp order; if $\langle \text{Commit } P \rangle$ precedes $\langle \text{Commit } Q \rangle$ in $H \mid x$, then $\text{ts}(P)$ precedes $\text{ts}(Q)$.

4. CONFLICT-BASED VALIDATION

This section introduces *conflict-based validation*, an optimistic concurrency control mechanism for which validation is based on predefined conflicts between pairs of operations. This approach is the optimistic analog of locking mechanisms, which use similar predefined conflicts to introduce delays.

We consider two distinct local validation techniques [19]: *backward* validation ensures that the transaction's results have not been invalidated by the effects of a recently committed transaction, while *forward* validation ensures that the transaction's effects will not invalidate the results of any active transaction.

4.1 Definitions and Lemmas

We begin with a definition of serial dependency, the property that underlies our notion of conflict. Let h be a history, g a subhistory (i.e., subsequence) of h, and $\mathbf{C}$ a binary relation between operations.

Definition 1. g is a $\mathbf{C}$-*closed* subhistory of h if whenever g contains an operation q of h it also contains every earlier operation p such that $(q, p) \in \mathbf{C}$.

Definition 2. A subhistory g of h is a $\mathbf{C}$-*view* of h for q if g is $\mathbf{C}$-closed, and if it includes every p in h such that $(q, p) \in \mathbf{C}$.

Informally, $\mathbf{C}$ is a *serial dependency* relation if, whenever an operation is legal for a $\mathbf{C}$-view, it is legal for the complete history. More precisely, let a dot denote concatenation:

Definition 3. $\mathbf{C}$ is a *serial dependency relation* if for all operations q, and all legal histories g and h such that g is a $\mathbf{C}$-view of h for q, $g \cdot q$ is legal $\Rightarrow h \cdot q$ is legal.

A serial dependency relation $\mathbf{C}$ is *minimal* if no $\mathbf{C}' \subset \mathbf{C}$ is also a serial dependency relation. The notion of serial dependency arises in a variety of contexts, including algorithms for managing replicated data [20, 21], and algorithms for pessimistic concurrency control, both locking [24] and multiversion timestamping [22].

We now outline some ways to construct serial dependency relations from the serial specifications for objects and give some examples of serial dependency relations for particular types of objects. One way of defining a dependency relation for an object is to say that an operation depends on any earlier operations that might invalidate it. More precisely,

Definition 4. Operation p *invalidates* operation q if there exist histories h_1 and h_2 such that $h_1 \cdot p \cdot h_2$ and $h_1 \cdot h_2 \cdot q$ are legal, but $h_1 \cdot p \cdot h_2 \cdot q$ is not. Define the relation *ivalidated-by* to contain all pairs (p, q) such that q invalidates p.

The *invalidated-by* relation is a serial dependency relation [24], although it need not be minimal.

"Failure to commute" is also a serial dependency relation [24], where we use Weihl's notion of commutativity [42, 44]:

Definition 5. Two histories h and h' are *equivalent* if they cannot be distinguished by any future computation: $h \cdot g$ is legal if and only if $h \cdot g$ is legal for all histories g. Two operations p and q *commute* if for all histories h, whenever $h \cdot p$ and $h \cdot q$ are both legal, then $h \cdot p \cdot q$ and $h \cdot q \cdot p$ legal and equivalent. Define the relation *failure-to-commute* to contain all pairs (p, q) such that q and p do *not* commute.

Like *invalidated-by*, *failure-to-commute* is not necessarily a minimal serial dependency relation.

For the Account data type, *invalidated-by* yields the serial dependency relation shown in Table I, and the *failure-to-communte* relation yields the symmetric closure of this relation. Here, an entry indicates that the row operation depends

Table I. Serial Dependency Relation for Account

	Credit(n)/Ok	Debit(n)/Ok	Debit(n)/Over
Credit(m)/Ok			
Debit(m)/Ok		true	
Debit(m)/Over	true		

on the column operation when the indicated condition holds. This particular serial dependency relation is minimal. It takes into account operation names and termination conditions, but not argument or result values. Debits do not depend on prior credits, because the debit cannot be invalidated by increasing the balance. Overdrafts do depend on prior credits, however, because the exception can be invalidated by increasing the balance.

Now consider a FIFO queue in which Deq blocks when the queue is empty. For this data type, *invalidated-by* and *failure-to-commute* yield distinct serial dependency relations, shown in Tables II and III. In the first relation, an Enq operation cannot be invalidated by any other operations, but a Deq operation can be invalidated either by an Enq of a different value or by a Deq of the same value. In the second relation, Enq operations with distinct arguments fail to commute, as do Deq operations with the same argument.

We make extensive use of the following lemmas and definitions when reasoning about serial dependency relations. The following lemma states that any sequence of operations can be inserted into the middle of a history provided no later operation depends on any inserted operation.

LEMMA 6. *If* **C** *is a serial dependency relation, f, g, and h histories such that $f \cdot g$ and $f \cdot h$ are legal, and there is no q in h and p in g such that $(q, p) \in \mathbf{C}$, then $f \cdot g \cdot h$ is legal.*

PROOF. The proof is by induction on the length of h. If h is empty, the result is immediate. Otherwise, let $h = h' \cdot q$. *By assumption,* $f \cdot h$ is a **C**-view of $f \cdot g \cdot h'$ for q. Moreover, $f \cdot g \cdot h'$ is legal by the induction hypothesis, and $f \cdot h' \cdot q$ is legal by assumption, thus $f \cdot g \cdot h' \cdot q = f \cdot g \cdot h$ is legal by Definition 3. □

Let g and h be legal histories and q an operation.

Definition 7. *g is a false* **C***-view* of h for q if g is a **C**-view of h for q, $g \cdot q$ is legal, but $h \cdot q$ is not.

By Definition 3, **C** is a dependency relation if and only if it has no false views.

The following lemma states that if **C** is *not* a serial dependency relation, it has a false view missing exactly one operation of h.

LEMMA 8. *If* **C** *is not a serial dependency relation, then there exist legal histories g and h and an operation q such that g is a false* **C***-view of h for q missing exactly one operation.*

Table II. First Serial Dependency Relation for Queue

	Enq(v')/Ok	Deq/Ok(v')
Enq(v)/Ok		
Deq/Ok(v)	$v \neq v'$	$v = v'$

Table III. Second Serial Dependency Relation for Queue

	Enq(v')/Ok	Deq/Ok(v')
Enq(v)/Ok	$v \neq v'$	
Deq/Ok(v)		$v = v'$

PROOF. Since **C** is not a serial dependency relation, there exists a false **C**-view g of h for some operation q. Suppose g is missing k operations of h. Consider the sequence of histories $\{h_i \mid i = 0, \ldots, k\}$, where $h_0 = g$, $h_k = h$, and h_{i+1} is constructed by inserting in h_i its earliest "missing" operation, that is, the earliest operation in h but not in h_i.

Suppose there exists an i such that h_i is legal but h_{i+1} is not. Let $g_0 \cdot p \cdot g_1 \cdot r$ be the shortest illegal prefix of h_{i+1}, where p is the operation inserted in h_i to produce h_{i+1}, and r is an operation. The schedule $g_0 \cdot g_1 \cdot r$ is legal as a prefix of h_i, $g_0 \cdot p \cdot g_1$ is legal by assumption, but $g_0 \cdot p \cdot g_1 \cdot r$ is illegal. Because h_i is a **C**-closed subschedule of h_{i+1}, $g_0 \cdot g_1$ is a false **C**-view of $g_0 \cdot p \cdot g_1$ for r, proving the lemma.

Otherwise, suppose all the h_i are legal. Because $h_0 \cdot q = g \cdot q$ is legal and $h_k \cdot q = H \cdot q$ is not, there must exist an i such that $h_i \cdot q$ is legal but $h_{i+1} \cdot q$ is not. This h_i is a false **C**-view of h_{i+1} for q missing one operation, proving the lemma. □

4.2 Forward Validation

Forward validation ensures that a committing transaction cannot invalidate any active transactions. When a transaction executes an operation at an object, the object grants an *optimistic lock* for that operation. That object will validate a transaction Q if and only if there is no other active transaction that holds an optimistic lock for an operation that conflicts with an operation in the intentions list for Q. A transaction's optimistic locks are released when it commits or aborts.

Formally, each object is modeled by an automaton that accepts certain schedules. The automaton's state is defined using the following primitive domains: TRANS is the set of transaction identifiers and OP is the set of operations. The derived domain HISTORYis the set of sequences of operations. X → Y denotes the set of partial maps from X to Y. A *forward validation atuomaton* has the following state components:

Perm: HISTORY
Intent: TRANS → HISTORY
O-Lock: OP → 2^{TRANS}

Perm is a history that represents the object's committed state, initially empty.[1] Intent(Q) is the history of operations executed by transaction Q, initially none. A transaction's *view* is constructed by appending its intentions to the object's committed state; that is, the history View(Q) is defined to be Perm · Intent(Q). O-Lock(q) is the set of active transactions that hold an optimistic lock for operation q, initially none.

Each transition has a precondition and a postcondition. For brevity, we assume that all input schedules are well formed. (Well formedness could be checked explicitly by adding more state components and preconditions.) In postconditions, primed component names denote new values, and unprimed names denote old values. For transaction Q to execute operation q,

Pre: View(Q) · q is legal.
Post: Intent'(Q) = Intent(Q) · q
O-Lock'(q) = O-Lock(q) ∪ {Q}.

A transaction may execute an operation only if it appears to be legal. Once the execution is complete, the operation is appended to the transaction's intentions and the transaction is given an optimistic lock for the operation.

Validation is governed by an *optimistic conflict relation* **OC** ⊆ OP × OP. Let Q be the validating transaction.

Pre: For all q in Intent(Q), $(p, q) \in$ **OC** ⇒ O-Lock(p) − {Q} = ∅.
Post: Perm' = Perm · Intent(Q)
For all q in Intent(Q), O-Lock'(q) = O-Lock(q) − {Q}.

A transaction may commit only if no other transaction holds an optimistic lock for a conflicting operation. Afterwards, the transaction's intentions list is applied to the permanent state, and the optimistic locks are released. When a transaction aborts, its optimistic locks are released.

Forward validation ensures that no active transaction can be invalidated by the commit of another transaction. Moreover, no active transaction ever sees an inconsistent state:

LEMMA 9. *For any forward validation automaton whose optimistic conflict relation is a serial dependency relation, View(Q) is legal for all active Q.*

PROOF. The argument proceeds by induction on the number of transactions that have committed, showing that View(Q) remains legal when another transaction P commits. By the induction hypothesis, View(P) = Perm' = Perm · Intent(P) is legal, as is View(Q) = Perm · Intent(Q). The validation condition for P implies that there is no q in Intent(Q) and p in Intent(P) such that $(q, p) \in$ **OC**. Because **OC** is a serial dependency relation, View(Q) = Perm · Intent(Q) and View'(Q) = Perm · Intent(P) · Intent(Q) satisfy the conditions of Lemma 6, hence View'(Q) is legal. □

The correctness theorem for forward validation is a direct consequence of Lemma 9:

[1] In practice, an object's committed state would be represented by a more compact and efficient data structure, such as an account balance.

THEOREM 10. *A forward validation automaton whose optimistic conflict relation is a serial dependency relation will accept only hybrid atomic schedules.*

PROOF. Recall that a schedule is hybrid atomic if the result of serializing committed transactions in commit timestamp order is a legal history. By construction, the automaton's Perm component is exactly this serialization, and Lemma 9 implies that each commit carries Perm from one legal history to another. □

Serial dependency is a necessary as well as sufficient condition on the automaton's optimistic conflict relation:

THEOREM 11. *Any forward validation automaton whose optimistic conflict relation is not a serial dependency relation will accept some schedule that is not hybrid atomic.*

PROOF. Since **OC** is not a serial dependency relation, there exist by Lemma 8 an operation q and legal histories h and g such that g is a false **OC**-view of h for q missing exactly one operation p. Let $g = g_1 \cdot g_2$ and $h = g_1 \cdot p \cdot g_2$. Suppose transaction R executes g_1 and commits, leaving Perm $= g_1$. P executes p, and Q executes $g_2 \cdot q$. Now P attempts to commit. Since $g_1 \cdot g_2$ is an **OC**-view of h for q, no optimistic lock held by Q conflicts with p; hence P will be validated. When Q commits, it is trivially validated, leaving as the final value of Perm the illegal history $g_1 \cdot p \cdot g_2 \cdot q = h \cdot q$. □

4.3 Backward Validation

Backward validation ensures that the committing transaction has not been invalidated by the recent commit of another transaction. Each object keeps track of Last(q), the most recent commit timestamp for a transaction that executed the operation q. For each active transaction Q, each object also keeps track of First(Q, q), the logical time when Q first executed q. Here, too, validation is governed by an optimistic conflict relation **OC**. An object will validate Q if and only if Last(p) < First(Q, q) for all q in Intent(Q) and all p such that $(q, p) \in$ **OC**. This condition ensures that Q has not been invalidated by a transaction that committed since Q executed q.

Let TIMESTAMP be a totally ordered set of timestamps with minimal and maximal elements $-\infty$ and ∞. In a *backward validation automaton*, the O-Lock component is replaced by

Clock: TIMESTAMP
First: TRANS × OP → TIMESTAMP
Last: op → TIMESTAMP.

The Clock component models a system of logical clocks, initially set to an arbitrary value. Last(q) and First(Q, q) are initialized to $-\infty$ and ∞. The precondition for Q to execute q is unchanged. The postcondition is slightly different: the clock is advanced and First(Q, q) is updated if necessary.

Pre: View(Q) · q is legal.
Post: Clock′ > Clock
First′(Q, q) = min(First(Q, q), Clock).

For Q to commit,

Pre: For all q in Intent(Q), $(q, p) \in \mathbf{OC} \Rightarrow$ First(Q, q) > Last(p).
Post: Clock′ > Clock
Perm′ = Perm · Intent(Q)
For all q in Intent(Q), Last′(q) = Clock.

The precondition states that a transaction may commit only if no recently committed transaction has executed a conflicting operation. Afterwards, the Last timestamp is updated for each operation executed by that transaction.

An active transaction is defined to be *valid* if the precondition for its commit is satisfied. Backward validation ensures that all valid transactions view consistent states, although invalid transactions may not.

LEMMA 12. *If the optimistic conflict relation is a serial dependency relation, then View(Q) is legal for any valid Q.*

PROOF. As before, we argue by induction on the number of committed transactions. The base case is trivial, so it is enough to show that if the commit of P does not invalidate Q, then View(Q) remains legal. If Q remains valid, there is no q in Intent(Q) and p in Intent(P) such that $(q, p) \in \mathbf{OC}$. By the induction hypothesis, View(P) = Perm · Intent(P) and View(Q) = Perm · Intent(Q) are both legal; therefore View′(Q) = Perm · Intent(P) · Intent(Q) is legal by Lemma 6. □

The basic correctness theorem for backward validation is a direct consequence of Lemma 12:

THEOREM 13. *Any backward validation automaton whose optimistic conflict relation is a serial dependency relation will accept only hybrid atomic schedules.*

PROOF. Perm is the serialization in commit timestamp order of the schedule accepted by the automaton, and Lemma 12 implies that each commit carries Perm from one legal history to another. Since the accepted schedule is serializable in commit timestamp order, it is hybrid atomic. □

Serial dependency is also a necessary condition for backward validation:

THEOREM 14. *Any backward validation automaton whose optimistic conflict relation is not a serial dependency relation will accept a schedule that is not hybrid atomic.*

PROOF. By the same scenario constructed for Theorem 11. □

4.4 Discussion

The Account data type illustrates how a type-specific definition of conflict improves validation. Under conventional schemes employing Read/Write conflicts, both Credit and Debit would be classified as a combination of Read and Write operations; hence any transaction to access the account would either invalidate or be invalidated by any concurrent transaction. Here, there are fewer conflicts: a credit can invalidate an overdraft, and a debit can invalidate another debit, but no other operations conflict.

It is difficult to judge whether forward or backward validation is preferable for conflict-based validation. The run-time costs of both techniques are comparable.

Table IV. Invocation/Invocation Conflict Relation for Account

	Credit	Debit
Credit		
Debit	true	true

An advantage of forward validation is that if all objects employ forward validation, then no transaction can observe a nonserializable state as a result of synchronization conflicts.[2] Also, asymmetric conflicts can sometimes be resolved by postponing rather than by denying validation. For example, if a transaction that credited an Account discovers that an active transaction has attempted an overdraft, the crediting transaction might choose to postpone validation until the other has had a chance to commit. The principal drawback of forward validation is its extreme optimism: while backward validation restarts active transactions in favor of committed transactions, forward validation restarts active transactions in favor of other active transactions, which themselves may never commit.

Because validation occurs after the invocations' results are known, validation can readily exploit information about operations' results. In many pessimistic schemes, a lock is acquired before invoking an operation; thus conflicts must be defined between invocations, not between complete operations. More recently, however, pessimistic schemes have been proposed in which a transaction requests a lock after tentatively executing the operation [24, 44]. These schemes incorporate a kind of "operation-level" optimism; if the lock cannot be granted, that operation (but not the whole transaction) must be rolled back. The advantages of exploiting result information can be illustrated by comparing the operation/operation conflict relation for Account in Table I and the invocation/invocation conflict relation in Table IV. Invocation locks for credit and debit must conflict, but conflict-based validation will permit a credit to occur concurrently with a debit (but not an overdraft), a useful distinction if most debits are expected to be successful.

Optimistic schemes can also exploit knowledge about the order in which transactions commit. For example, under backward validation, a transaction that executed an overdraft will be allowed to commit before (but not after) a concurrent transaction that executed a conflicting credit, while pessimistic locking would have introduced a delay.

4.5 Mixing Pessimistic and Optimistic Methods

Because hybrid atomicity is a local atomicity property, the optimistic techniques proposed so far are compatible with hybrid atomic pessimistic schemes (such as two-phase locking), implying that objects employing optimistic and pessimistic techniques can be used together in a single system. This section shows that optimistic and pessimistic techniques can also be combined within a single object. For example, consider an Account whose balance is expected to cover all debits, but for which concurrent debits are frequent. Optimistic techniques are well

[2] In a distributed system, an *orphan* transaction [33] whose optimistic locks have been released prematurely by a site crash may observe nonserializable states before aborting.

suited for resolving the infrequent conflicts between credits and debits, but poorly suited for the more frequent conflicts between debits. A mixed scheme could exploit the strengths of each method by using pessimistic techniques to prevent "high-risk" conflicts, reserving optimistic methods to detect "low-risk" conflicts.

Mixed conflict-based validation is implemented as follows. After a transaction executes an operation, but before it updates its intentions list, it requests a *pessimistic lock* for that operation. Pessimistic locks are related by a *pessimistic conflict* relation **PC**. If any other transaction holds a conflicting pessimistic lock, where conflict is defined by **PC**, then the lock is refused, the operation is discarded, and the caller waits and retries the operation when the lock is released. When the lock is granted, the intentions list is updated, and the response is returned to the caller. A transaction's pessimistic locks are released when it commits or aborts. When the transaction commits, validation proceeds as before.

Unlike optimistic conflict relations, pessimistic relations must be symmetric, since the order in which transactions eventually commit is unknown when pessimistic conflicts are detected. Define an object's *conflict relation* to be the union of its optimistic and pessimistic conflict relations. The fundamental constraint governing an object's optimistic and pessimistic conflict relations is the following: the conflict relation must be a serial dependency relation. An empty pessimistic relation yields the conflict-based validation scheme of Section 4, and an empty optimistic relation yields a type-specific two-phase locking scheme similar to that of [24]. Numerous possibilities lie between these two extremes; the appropriate balance between pessimism and optimism depends on the expected frequency of each conflict.

The mixed protocol is modeled by adding the following state component to both the forward and backward validation automata:

$$\text{P-Lock: } \text{OP} \rightarrow 2^{\text{TRANS}}.$$

P-Lock(q) is the set of transactions that hold pessimistic locks for q. Initially, all such sets are empty. The precondition for Q to execute q has an additional clause:

$$\text{If } (q, p) \in \mathbf{PC} \text{ or } (p, q) \in \mathbf{PC}, \text{ then P-Lock}(p) - \{Q\} = \varnothing.$$

Note that lock conflicts are determined by the symmetric closure of **PC**. Afterwards, the transaction is granted a pessimistic lock for the operation.

$$\text{P-Lock}'(q) = \text{P-Lock}(q) \cup \{P\}.$$

A transaction's pessimistic locks are released when it commits or aborts.

Pessimistic conflicts prevent concurrent transactions from executing conflicting operations:

LEMMA 15. *For a mixed (forward or backward) automaton, if Q and P are concurrent active transactions, and q is an operation in Intent(Q), then there is no p in Intent(P) such that $(q, p) \in \mathbf{PC}$.*

PROOF. The precondition for Q to execute q ensures that the property holds initially, and it prevents any other transaction from violating the property while Q is active. □

LEMMA 16. *For any mixed forward validation automaton whose conflict relation is a serial dependency relation, View(Q) is legal for all active Q.*

PROOF. As before, it is enough to show that View(Q) remains legal after the commit of a distinct transaction P. By the induction hypothesis, View(P) = Perm · Intent(P) and View(Q) = Perm · Intent(Q) are legal. There is no q in Intent(Q) and p in Intent(P) such that $(q, p) \in \mathbf{OC}$ (Lemma 9) or $(q, p) \in \mathbf{PC}$ (Lemma 15). Because $\mathbf{PC} \cup \mathbf{OC}$ is a serial dependency relation, View'(Q) = Perm · Intent(P) · Intent(Q) is legal by Lemma 6. □

LEMMA 17. *For any mixed backward validation automaton whose conflict relation is a serial dependency relation, View(Q) is legal for all valid Q.*

PROOF. If P commits without invalidating Q, there is no q in Intent(Q) and p in Intent(P) such that $(q, p) \in \mathbf{OC}$ (Lemma 12) or $(q, p) \in \mathbf{PC}$ (Lemma 15); therefore View'(Q) is legal by Lemma 6. □

The proofs of the remaining correctness and optimality results are omitted for brevity, since they are almost identical to their analogs in the previous section.

THEOREM 18. *All schedules accepted by a mixed forward or backward validation automaton will be hybrid atomic if and only if the automaton's conflict relation is a serial dependency relation.*

5. STATE-BASED VALIDATION

Although conflict-based validation accepts more interleavings than other optimistic schemes, it will nevertheless restart certain transactions unnecessarily. For example, one debiting transaction need not be invalidated by another if the balance covers both debits. The optimality proofs given above imply that no scheme, optimistic or pessimistic, can permit concurrent debits simply on the basis of conflicts between pairs of operations. Instead, the accuracy of validation can be enhanced only by taking objects' states into account. Such *state-based* validation may be more expensive than conflict-based validation, since it may (at worst) amount to reexecuting part of the transaction. Nevertheless, state-based validation may be cost effective in special cases where predefined conflicts are too restrictive and where validation conditions can be evaluated efficiently.

5.1 Model

A *state-based validation* automaton has the following state components:

Perm: HISTORY
Intent: TRANS → HISTORY.

The pre- and postconditions for Q to execute q are the following:

Pre: true.
Post: Intent'(Q) = Intent(Q) · q.

The precondition states that a transaction is free to execute any operation; illegal operations will be detected at validation. In practice, however, it would be prudent to permit Q to execute q only if View(Q) · q is legal.

Using backward validation, Q commits as follows:

Pre: View(Q) is legal.
Post: Perm′ = View(Q).

While using forward validation,

Pre: For all active P distinct from Q, Perm · Intent(Q) · Intent(P) is legal.
Post: Perm′ = View(Q).

These formulations reveal an important practical asymmetry between forward and backward state-based validation: forward validation is linear in the number of concurrent transactions, while backward validation is constant. Consequently, we do not further consider state-based forward validation.

5.2 An Example

The cost of conflict-based validation is largely type independent, but the cost of state-based validation depends on type-specific properties: how compactly intentions can be represented and how efficiently they can be validated. The following idealized implementation of an Account provides an "existence proof" that state-based validation can be efficient for certain data types. Informally, the key idea is the following. Consider a transaction whose only operation is to debit k dollars. That transaction has "observed" that the account balance is at least k, and it can be validated as long as that observation remains valid. Similarly, a transaction that unsuccessfully tries to debit k dollars has observed that the account balance is less than k. For longer transactions that execute a sequence of credits, successful debits, and unsuccessful debits, these conditions can be encoded into two quantities: an observed upper bound on the account balance and an observed lower bound. These bounds are adjusted with each operation, and the transaction can be validated as long as the final committed balance lies between them.

More precisely, an Account is modeled as an automaton with the following components:

Bal: INT
Low: TRANS → INTEGER
High: TRANS → INTEGER
Change: TRANS → INTEGER.

Bal is the account's permanent state, represented here as a balance. Low(Q) is the observed lower bound on the current balance (initially zero), High(Q) is the observed upper bound (initially ∞), and Change(Q) is the transaction's net change to the balance (initially zero).

Account operations have the following pre- and postconditions. For Q to execute Debit(k)/Ok( ),

Pre: Bal + Change(Q) $\geq k$.
Post: Change′(Q) = Change(Q) − k
Low′(Q) = max(Low(Q), k − Change(Q)).

The precondition states that the debit returns successfully only if the balance appears to cover the debit. The postcondition states that Q records its cumulative

change and adjusts its observed lower bound. For Debit(k)/Overdraft( ),

Pre: Bal + Change(Q) < k
Post: High'(Q) = min(High(Q), k − Change(Q)),

and for Credit(k)/Ok( ),

Pre: true
Post: Change'(Q) = Change(Q) + k.

Q will be validated if and only if the current committed balance lies between the observed upper and lower bounds:

$$\text{Low}(Q) \leq \text{Bal} < \text{High}(Q).$$

After Q commits, its changes are applied to the balance:

$$\text{Bal}' = \text{Bal} + \text{Change}(Q).$$

In this particular example, the run-time cost of validation is comparable to the cost of conflict-based validation.

Reuter [36] has proposed a pessimistic concurrency control algorithm for counters that is based on similar principles: transactions may execute concurrently as long as they do not invalidate one another's observed upper and lower bounds. Reuter's algorithm supports a slightly different mix of operations than ours, and it imposes additional restrictions: transactions are not allowed to execute certain operations more than once.

5.3 Discussion

This section compares the computational complexity of optimistic and pessimistic state-based concurrency control.

A schedule is *on-line hybrid atomic* if any active transaction can commit at any time without violating hybrid atomicity. A pessimistic hybrid atomic scheduler, by definition, accepts only on-line hybrid atomic schedules. A *hybrid serialization* of a schedule is a history constructed by committing some set of active transactions and serializing the result in commit order. A schedule is on-line hybrid atomic if and only if all its hybrid serializations are legal.

THEOREM 19. *In general, recognizing that a schedule is not on-line hybrid atomic is NP-complete in the number of active transactions.*

PROOF. Recognizing that H is not on-line hybrid atomic is equivalent to recognizing that it has an illegal hybrid serialization. This problem is in NP because it suffices to "guess" a history and to verify in polynomial time that it is an illegal hybrid serialization of H.[3] To see that the problem is NP-complete for certain data types, let us augment the Account data type with the following operation:

isEqual(k: Dollar) returns(bool)

[3] We assume here that for any data type of practical interest, one can check the legality of a single hybrid serialization in time polynomial in the number of active transactions. Without this assumption, the problem become NP-hard rather than NP-complete.

This operation returns *true* if and only if the current balance is k. Let $A = \{i_1, \ldots, i_n\}$ be an arbitrary set of positive integers, and let H be the following (on-line hybrid atomic) schedule:

$$\begin{array}{c} \text{Credit}(i_1)/\text{Ok}(\,)Q_1 \\ \vdots \\ \text{Credit}(i_n)/\text{Ok}(\,)Q_n \end{array}$$

Deciding whether $H \cdot \langle \text{isEqual}(k)/\text{Ok}(\text{false})Q \rangle$ is *not* on-line hybrid atomic is equivalent to deciding whether there exists $A' \subseteq A$ such that the sum of the values in A' is exactly k. This *subset sum* problem is known to be NP-complete [15]. □

Deciding on-line hybrid atomicity is not intractable for every data type; one can sometimes exploit the type's algebraic structure to reduce the number of histories whose legality must be checked. One might also be able to derive state-based techniques for efficiently recognizing certain subsets of the on-line hybrid atomic schedules. Nevertheless, Theorem 19 implies that pessimistic state-based techniques cannot fully exploit all the concurrency permitted by hybrid atomicity.[4]

Optimistic state-based techniques have very different properties. A scheduling decision that is computationally intractable for a pessimistic scheduler can be validated after the fact in constant time. To validate Q, it suffices to check the legality of the single history Perm · Intent(Q), a cost independent of the number of concurrent transactions. These observations support Gawlick's [16] empirical claim that optimistic state-based techniques are well-suited for certain *hot spots*, objects that are accessed frequently by concurrent transactions. A common example of such an object is a nonnegative "quantity on hand" counter that can be incremented, decremented, and read. Conflict-based techniques, whether optimistic or pessimistic, permit too little concurrency because decrementing operations, like debits, must conflict. State-based techniques do permit adequate concurrency, but pessimistic techniques may be too expensive, since each transaction's cost increases with the level of concurrency, which is assumed to be high. If real conflicts are rare, optimistic state-based techniques seem to be the most promising, since each transaction's cost remains constant.

6. REPLICATED OBJECTS

A *replicated object* is a typed data object whose state is stored redundantly at multiple locations to enhance availability. A *replication method* is an algorithm for managing the object's distributed components so that its functional behavior is equivalent to that of a single-site object, a property known as *one-copy serializability* [5]. This section describes several ways to integrate optimistic techniques with quorum consensus replication [17, 20]. We first show that conflict-based schemes extend naturally to replicated data, imposing the same restrictions on availability as their pessimistic counterparts. Optimistic state-based schemes, however, impose *weaker* restrictions on availability than their

[4] Unless, of course, P = NP.

pessimistic counterparts, implying that one can achieve a wider range of availability tradeoffs if one is willing to accept the risk that validation will fail.

6.1 Model

Replicated objects are implemented by two kinds of sites: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends carry out operations. A transaction applies an operation to a replicated object by sending an invocation to one of the object's front-ends. The front-end reads data from some collection of repositories, carries out a local computation, sends updates to some collection of repositories, and returns the response to the calling transaction. The transaction must locate an available front-end for the object, and the front-end must in turn locate enough available repositories to carry out the operation. Front-ends can be replicated to an arbitrary extent, perhaps placing one at each site, implying that the availability of the replicated object is dominated by the availability of its repositories.

A *quorum* for an operation is any set of repositories whose cooperation suffices to execute that operation. It is convenient to divide a quorum into two parts: a front-end executing an operation reads from an *initial quorum* and writes to a *final quorum*. (Either the initial or final quorum may be empty.) A *quorum assignment* associates each operation with a set of valid initial and final quorums. An object's quorum assignment determines the availability of its operations; thus the constraints governing quorum assignment are the basic constraints governing the availability realizable by a replication method.

Quorum assignments are constrained by a *quorum intersection relation*: certain initial and final quorums are required to have nonempty intersections. For example, any quorum assignment for a replicated file [17] must ensure that each initial Read quorum intersects each final Write quorum; otherwise it would be impossible to guarantee that each value read is the value most recently written. If two operations are related by the quorum intersection relation, then their levels of availability can be traded off: if one operation's quorums are made smaller (rendering it more available), then the other's quorums must be made correspondingly larger (rendering it less available).

6.2 Conflict-Based Validation

At each repository, the operations of committed transactions are recorded in a *log*, which is a sequence of *entries*, where an entry is the timestamped record of an operation. For example, Table V is a schematic representation of an Account replicated among three repositories. For readability, a "missing" entry is marked by a blank. The account has been credited three times, although no repository has an entry for all three.

A *log* is a map from a finite set of timestamps to operations. Two logs L and M are *coherent* if they agree for every timestamp where they are both defined. The *merge* operation $\cup$ is defined on pairs of coherent logs by:

$$(L \cup M)(t) = \text{if } L(t) \text{ is defined then } L(t) \text{ else } M(t).$$

Every log corresponds to a history in the obvious way. For brevity, we sometimes refer to a log L in place of its history; for example, "L is legal" instead of "the

Table V. A Replicated Account

R1	R2	R3
1:00 Credit ($5)/Ok( )	1:00 Credit ($5)/Ok( )	
	1:15 Credit ($10)/Ok( )	1:15 Credit ($10)/Ok( )
1:30 Credit ($15)/Ok( )		1:30 Credit ($15)/Ok( )

history represented by L is legal." The log whose single entry is q with timestamp t is denoted by $[t \rightarrow q]$.

A *replicated forward validation automaton* has the following state components. Let REPOS be the set of repositories, and QUORUM = 2^{REPOS} the set of possible quorums.

Perm: REPOS → LOG
Intent: REPOS → (TRANS → LOG)
O-Lock: REPOS → (OP → 2^{TRANS})
Clock: TIMESTAMP.

Let R be a repository, Q a transaction, and p an operation. Intent(R, Q) is the log of entries for Q recorded at R, initially none. O-Lock(R, q) is the set of transactions holding optimistic locks for q at R, initially none. As before, *Clock* models a system of logical clocks. A transaction's *view* at R, written View(R, Q), is Perm(R) · Intent(R, Q). If S is a set of repositories, and Q a transaction, let Intent(S, Q) denote $\cup_{R \in S}$ Intent(R, Q), and similarly for View(S, Q) and Perm(S).

The automaton's transition relation is defined using the following sets:

(1) an *optimistic conflict* relation **OC**;
(2) Initial: OP → 2^{QUORUM} assigns initial quorums to operations;
(3) Final: OP → 2^{QUORUM} assigns final quorums to operations.

Initial and Final define a quorum intersection relation **Q** as follows: $(q, p) \in \mathbf{Q}$ if every initial quorum for q intersects every final quorum for p.

We have defined initial quorums and optimistic locks in terms of complete operations, not just invocations. How can a front-end executing a Debit predict whether it should acquire locks for a debit or an overdraft? An optimistic strategy is to guess that the debit attempt will succeed, requesting locks for a debit. If the balance is insufficient, the front-end releases its locks and restarts the operation (but not the entire transaction), guessing this time that the debit attempt will fail. A pessimistic strategy is to acquire locks for both operations, perhaps releasing the superfluous lock when the response becomes known. (Similar considerations arise if an operation's choice of initial quorums depend on its anticipated result.)

For Q to execute operation q, there must exist an initial quorum IQ in Initial(q) and a final quorum FQ in Final(q) such that

Pre: View(IQ, Q) · q is legal.
Post: Clock′ > Clock
For all R in IQ, O-Lock′(R, q) = O-Lock(R, q) ∪ {Q}
For all R in FQ,
Perm′(R) = Perm(R) ∪ Perm(IQ)
Intent′(R, Q) = Intent(R, Q) ∪ Intent(IQ, Q) ∪ [Clock → q].

A transaction may execute an operation only if it appears to be legal from the view assembled from an initial quorum. Once the execution is complete, the clock is advanced, the operation is appended to the transaction's view, the view is merged with the logs at a final quorum of repositories, and the transaction acquires an optimistic lock for the operation at every repository in the initial quorum.

Validation proceeds as follows. Let IQ (FQ) be the set of repositories that participated in an initial (final) quorum for transaction Q.

Pre: For all R in FQ and q in Intent(R, Q), $(p, q) \in \mathbf{OC} \Rightarrow$ O-Lock(R, p) $-$ $\{Q\}$ $= \emptyset$.
Post: Clock′ $>$ Clock
For all R in FQ, Perm′(R) = Perm(R) · Intent(R, Q)
For all R in IQ, O-Lock′(R) = O-Lock(R) $-$ $\{Q\}$.

Each repository in any of the transaction's final quorums checks that no other transaction holds a conflicting optimistic lock. If validation succeeds, the clock is advanced, the transaction's intentions are appended to the permanent state, and all locks are released. Note that validation is performed locally at each repository.

The following is a necessary and sufficient correctness condition: the intersection of the optimistic conflict relation **OC** with the quorum intersection relation **Q** must be a serial dependency relation. (In practice, OC and Q would probably be the same relation.) This requirement ensures that any conflict between two operations will be detected at some repository in the intersection of their quorums, causing validation to fail. This restriction on availability is identical to that imposed by *consensus locking* [21], a conflict-based pessimistic replication method.

Let RFV be a replicated forward validation automaton with $\mathbf{C} = \mathbf{OC} \cap \mathbf{Q}$. If **C** is a serial dependency relation, then every schedule accepted by RFV is also accepted by a forward validation automaton FV with optimistic conflict relation **C**. Since FV accepts only hybrid atomic schedules (Theorem 10), the same is true of RFV.

We use the following technical lemmas.

LEMMA 20. *If* **C** *is an arbitrary relation between operations, the result of merging* **C***-closed sublogs of a particular log is itself a* **C***-closed sublog.*

LEMMA 21. *Suppose RFV and FV have accepted the same schedule. If Q is an active transaction and S a set of repositories, then View(S, Q) (in RFV) is a* **C***-closed subhistory of View(Q) (in FV).*

PROOF. By induction on the length of the accepted schedule. The result is immediate when both automata have accepted the empty schedule. We assume the property holds, and show that it is preserved whenever both automata accept an additional step.

Suppose both automata accept $\langle q\ Q \rangle$, where RFV chooses initial and final quorums IQ and FQ. If $S \cap FQ = \emptyset$, then View′(S, Q) = View(S, Q), which remains closed in View′(Q) = View(Q) · q. Otherwise, View′(S, Q) = View(S, Q) $\cup$ View(IQ, q) $\cup$ [Clock $\rightarrow q$]. View(S, Q) and View(IQ, Q) are each **C**-closed by

the induction hypothesis, and View(IQ, Q) ∪ [Clock → q] is closed by construction, thus *View'*(S, Q) is **C**-closed by Lemma 20.

Suppose both automata accept ⟨p P⟩, for P distinct from Q, where RFV chooses initial and final quorums IQ and FQ. If $S \cap FQ = \emptyset$, then View'(S, Q) = View(S, Q), which remains closed in View'(Q) = View(Q). Otherwise, View'(S, Q) = View(S, Q) ∪ Perm(IQ). View(S, Q) and View(IQ, P) are each **C**-closed by the induction hypothesis, and Perm(IQ) is **C**-closed as a prefix of View(IQ, P), thus View'(S, Q) is **C**-closed by Lemma 20.

Suppose both automata accept ⟨commit P⟩, for P distinct from Q. View'(S, Q) = Perm(S) · Intent(S, P) · Intent(S, Q) and View'(Q) = Perm · Intent(P) · Intent(Q) · View'(S, Q) fails to be **C**-closed only if Intent(S, Q) includes a q and Intent(P) a p such that $(q, p) \in \mathbf{C}$ and p is not in Intent(S, P), violating the validation condition for P in RFV. □

Because **C** is a subset of the quorum intersection relation **Q**, the following statements can be made:

COROLLARY 22. *If Q is an active transaction and IQ is in Initial(q), then View(IQ, Q) is a* **C***-view of View(Q) for q.*

THEOREM 23. *If* **C** *is a serial dependency relation, then every schedule accepted by RFV is accepted by FV.*

PROOF. By induction on the length of the schedule. The result is immediate for schedules of length zero, so we assume the result for schedules of length n and show that any additional step accepted by RFV is accepted by FV.

Suppose RFV accepts ⟨q Q⟩ with initial and final quorums IQ and FQ. View(IQ, Q) is a **C**-view of View(Q) for q (Lemma 21), View(IQ, Q) · q is legal (precondition for accepting ⟨q Q⟩), and **C** is a serial dependency relation (by assumption), thus View(Q) · q is legal, and FV accepts ⟨q Q⟩.

Suppose RFV accepts ⟨Commit Q⟩. FV fails to accept ⟨Commit Q⟩ only if Q has executed an operation q and some active P has executed p such that $(p, q) \in \mathbf{C}$. If this condition holds, P in RFV has an optimistic lock for p at an initial quorum for p, but this initial quorum for p intersects the final quorum for q, which violates the validation condition for Q in RFV.

Since a forward validation automaton can be treated as a replicated forward validation automaton with a single repository, the optimality proof of Theorem 11 applies directly.

For backward validation, each repository keeps track of First(Q, q), the logical time when Q first executed q at that repository, and Last(q), the commit timestamp of the most recent transaction to execute q at that repository. A transaction Q is validated if and only if, for each operation q of Q and each operation p of P such that $(q, p) \in \mathbf{OC} \cap \mathbf{Q}$, Last($p$) < First($Q$, q) at each repository in the transaction's initial quorums. The automaton is correct if and only if **OC** ∩ **Q** is a serial dependency relation. The correctness argument is similar to that of Theorem 23 and is omitted. □

6.3 State-Based Validation

While conflict-based backward validation can be done at repositories, state-based backward validation must be done at front-ends, because the state information

Table VI. State-Based Validation: An Example

R1	R2	R3
1:00 Debit ($10)/Over( ) R	1:00 Debit ($10)/Over( ) R	1:00 Debit ($10)/Over( ) R
		1:15 Credit ($5)/Ok( ) P
1:30 Credit ($5)/Ok( ) Q		

at any individual repository may be incomplete. For example, consider an Account replicated among three repositories, where Credit has a quorum of one and Debit a quorum of three (see Table VI). Operations executed by active transactions are tagged with transaction names.

Clearly, these transactions cannot be validated in the order P, Q, and R, because the combination of the two $5 credits would invalidate the $10 overdraft. No individual repository, however, is guaranteed to observe all credits; thus no validation scheme performed at the repositories can permit concurrent credits and debits.

Instead, state-based validation must take place at front-ends. Each transaction's intentions are accumulated at a front-end. To validate the transaction, the front-end constructs a view by merging the logs from an initial quorum for the transaction and appending the intentions. If the updated view is legal, the transaction is validated, and the view is sent to a final quorum for the transaction, where it is merged with the resident logs. The volume of message traffic need for validation can be reduced if each front-end caches earlier views, requesting only the most recent updates.

A *replicated state-based* automaton has the following state components:

Perm: REPOS → LOG
Intent: TRANS → LOG
Clock: TIMESTAMP.

As in the previous section, the state transition relation is defined in terms of initial, Final, and their induced quorum intersection relation **Q**.

For transaction Q to execute operation q, there must exist an initial quorum IQ in Initial(q) and a final quorum FQ in Final(q) such that

Pre: true
Post: Clock′ > Clock
Intent′(Q) = Intent(Q) ∪ [Clock → q].

Let IQ and FQ be initial and final quorums for the transaction. For Q to commit,

Pre: View(IQ, Q) is legal.
Post: Clock′ > Clock
For all R in FQ, Perm′(R) = Perm(R) ∪ View(IQ, Q).

Let RSV be a replicated state-based validation automaton with quorum intersection relation **C**. If **C** is a serial dependency relation, we show that every schedule accepted by RSV is also accepted by a state-based validation automaton SV.

LEMMA 24. *Suppose RSV and SV have accepted the same schedule. If Q is an active transaction and S a set of repositories, then Perm(S) (in RSV) is a* **C***-closed subhistory of Perm (in SV).*

PROOF. By induction on the length of the accepted schedule. It is enough to check that the property is preserved as transactions commit.

Suppose transaction Q is validated by both automata, where IQ (FQ) is the union of initial (final) quorums for operations of Q in RSV. If $S \cap FQ = \emptyset$, then $\text{Perm}'(S) = \text{Perm}(S)$, which remains closed in $\text{Perm}' = \text{Perm} \cdot \text{Intent}(Q)$. Otherwise, $\text{Perm}'(S) = (\text{Perm}(S) \cup \text{Perm}(IQ)) \cdot \text{Intent}(Q)$. $\text{Perm}(S)$ and $\text{Perm}(IQ)$ are **C**-closed by the induction hypothesis, their merger is closed by Lemma 20; hence $\text{Perm}'(S)$ is closed because IQ encompasses an initial quorum for every operation in $\text{Intent}(Q)$. □

COROLLARY 25. *If Q is an active transaction and IQ is in Initial(q), then View(IQ, Q) is a* **C***-view of View(Q) for q.*

THEOREM 26. *If* **C** *is a serial dependency relation, then every schedule accepted by RSV is accepted by SV.*

PROOF. By induction on the length of the schedule. The result is immediate for schedules of length zero, so we show that any additional step accepted by RSV is accepted by SV. Because the precondition for accepting an operation step is trivial, it suffices to show that any transaction validated by RSV is validated by SV.

Suppose RSV validates Q with initial and final quorums IQ and FQ. We argue by induction on the length of $\text{Intent}(Q)$. The result is immediate if Q executed no operations; so assume the result for intentions of length n, and let $\text{Intent}(Q) = h \cdot q$, where h has length n. Because IQ encompasses an initial quorum for q, Corollary 25 implies that $\text{Perm}(IQ) \cdot h$ is a **C**-view of $\text{Perm} \cdot h$ for q. $\text{View}(IQ, Q) = \text{Perm}(IQ) \cdot h \cdot q$ is legal by assumption, $\text{Perm} \cdot h$ is legal by the induction hypothesis, and **C** is a serial dependency relation; hence $\text{Perm} \cdot h \cdot q = \text{View}(Q)$ is legal, and SV validates Q. □

6.4 Discussion

Replicated state-based validation provides a way to circumvent certain tradeoffs between concurrency and availability imposed by pessimistic state-based methods. The pessimistic analog of a replicated state-based validation automaton is a *consensus scheduling* automaton [21], which places the following constraints on availability. Let *Concur* be any prefix-closed set of on-line hybrid atomic schedules. If **C** is a binary relation on operations, the notions of **C**-closed and **C**-view can be extended to schedules and subschedules in the obvious way.

Definition 27. Let G be a **C**-view of H for q. **C** is an *atomic dependency* relation for *Concur* if $G \cdot \langle q\ Q \rangle$ is in *Concur* implies $H \cdot \langle q\ Q \rangle$ is in *Concur*.

The basic restriction governing quorum assignments for consensus scheduling is the following:

THEOREM 28. *If the quorum intersection relation is an atomic dependency relation for Concur, then every schedule accepted by a consensus scheduling automaton is in Concur. Moreover, if* **C** *is not an atomic dependency relation for Concur, there exists a consensus scheduling automaton with* **C** *as its quorum intersection relation that accepts a schedule not in Concur.*

Table VII. State-Based Validation: Quorum Assignments for Five-Site Account

Credit	(0, 1)	(0, 2)	(0, 3)
Debit	(5, 1)	(4, 2)	(3, 3)

Although the definitions of serial and atomic dependency are formally quite similar, there are important differences. Every atomic dependency relation for a data type is also a serial dependency relation, since it must ensure hybrid atomicity even when all operations are executed serially by a single transaction. The converse, however, is false: not all serial dependency relations are atomic dependency relations. For example, let Hybrid(Account) be the set of on-line hybrid atomic schedules for Account, and let **C** be the symmetric closure of the serial dependency relation shown in Table I. **C** is not an atomic dependency relation for Hybrid(Account). To see why, let p be the operation Credit($5)/OK( ), and H the following schedule:

Debit($10)/Overdraft( ) R
Credit($5)/Ok( ) Q

Let G be the subschedule of H that omits the last step. G is a **C**-view of H for p, $G \cdot \langle p\ P \rangle$ is on-line hybrid atomic, but $H \cdot \langle p\ P \rangle$ is not, because an illegal serialization results if the transactions commit in the order P, Q, and R.

It follows that a consensus scheduling requires initial and final Credit quorums to intersect, while state-based validation does not. This difference affects the availability tradeoff permitted by each technique. A state-based validation Account replicated among n identical sites permits $\lceil n/2 \rceil$ different ways to assign quorums to operations: for each $m > n/2$, any m repositories constitute an initial Debit quorum, and any $n - m + 1$ sites constitute a final Debit or final Credit quorum. Quorum assignments when $n = 5$ are shown in Table VII, where an entry of the form (i, j) indicates that the operation has initial (final) quorums consisting of any i (j) repositories. By contrast, the corresponding (pessimistic) consensus scheduling automaton permits only one quorum assignment: all initial and final quorums require a majority of repositories. In short, after-the-fact conflict detection places fewer constraints on availability than dynamic conflict avoidance.

7. CONCLUSIONS

We have suggested two reasons why conventional optimistic techniques seem poorly suited for decentralized distributed systems. First, such techniques are typically designed for database applications in which most concurrent operations are reads, an implausible assumption for many nondatabase applications. Second, such techniques are typically monolithic, applying to all the data encompassed within a system, an undesirable property in an administratively decentralized distributed system. This paper has proposed new techniques to address these limitations:

(1) Conflict-based validation systematically exploits type-specific properties to provide more effective validation than conventional techniques employing a

simple model of read/write conflicts. The problem of identifying a correct and minimal set of conflicts for an object is shown to be equivalent to the algebraic problem of identifying a minimal serial dependency relation for the data type.

(2) The optimistic techniques proposed here are modular, permitting individual objects to choose independently from optimistic, pessimistic, or mixed techniques. Optimistic techniques can be used to resolve "low-risk" conflicts, while standard pessimistic techniques can be used to resolve "high-risk" conflicts.

(3) Besides permitting more accurate validation, the notion of serial dependency also provides an "upper bound" on the concurrency realizable by conflict-based validation. An application that needs additional concurrency must use a validation technique that takes the object's state into account. *State-based* validation is a general technique that can validate any interleaving permitted by a pessimistic method, although its run-time cost is type dependent.

(4) State-based validation substantially reduces the algorithmic complexity of achieving high levels of concurrency. Pessimistic scheduling decisions that are NP-complete in the number of concurrent transactions can be validated after the fact in constant time.

(5) Conflict-based validation can be extended to replicated objects in a straightforward way, placing the same restrictions on availability as comparable pessimistic techniques. State-based validation, however, places fewer constraints on availability than comparable pessimistic techniques.

These results suggest that optimistic concurrency control may yet have a place in general-purpose distributed systems.

ACKNOWLEDGMENTS

I am grateful to Dean Daniels, Dan DuChamp, Ellen Siegel, Doug Tygar, Bill Weihl, and the referees for their careful readings of earlier drafts of this paper.

REFERENCES

1. AGRAWAL, R. Concurrency control and recovery in multiprocessor database machines: Design and performance evaluation. Ph.D. thesis, University of Wisconsin, Madison, 1983.
2. AGRAWAL, D., BERNSTEIN, A. J., GUPTA, P., AND SENGUPTA, S. Distributed optimistic concurrency control with reduced rollback. *Distributed Syst. 2*, 1 (1987).
3. AGRAWAL, R., CAREY, M. J., AND LIVNY, M. Models for studying concurrency control performance: Alternatives and implications. In *Proceedings of the International Conference on Management of Data* (Austin, Tex., May 1985). ACM, New York, 1985, pp. 108–121.
4. BADAL, D. Z. Concurrency control overhead or closer look at blocking vs. non-blocking concurrency control mechanisms. In *Proceedings of the 5th Berkeley Workshop*, 1981, pp. 55–103.
5. BERNSTEIN, P. A., AND GOODMAN, N. The failure and recovery problem for replicated databases. In *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing*, 1983.
6. BERNSTEIN, P. A., GOODMAN, N., AND LAI, M. Y. Two-part proof schema for database concurrency control. In *Proceedings of the 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1981.
7. BORAL, H., AND GOLD, I. Towards a self-adapting centralized concurrency control algorithm. In *SIGMOD 84*. ACM, New York, 1984, pp. 18–31.
8. CAREY, M. Modeling and evaluation of database concurrency control algorithms. Ph.D. dissertation, University of California, Berkeley, 1983.

9. CERI, S., AND OWICKI, S. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop*, of the 1982, pp. 117–130.
10. CHAN, A., FOX, S., LIN, W. T., NORI, A., AND RIES, D. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the 1982 SIGMOD Conference*. ACM, New York, 1982.
11. DUBOURDIEU, D. J. Implementation of distributed transactions. In *Proceedings of the 1982 Berkeley Workshop on Distributed Data Management and Computer Networks*, 1982, pp. 81–94.
12. DWORK, C., AND SKEEN, M. D. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd Annual Symposium on Principles of Distributed Computing* (Aug. 1983). ACM, New York, 1983, pp. 1–11.
13. ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The notion of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
14. FRANASZEK, P., AND ROBINSON, J. T. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst. 10*, 1 (March 1985), 1–28.
15. GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.
16. GAWLICK, D. Processing "not spots" in high performance systems. In *Proceedings COMPCON '85*, 1985.
17. GIFFORD, D. K. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles* (Dec. 1979). ACM, New York, 1979.
18. GRAY, J. Notes on database operating systems. *Lecture Notes in Computer Science 60*. Springer-Verlag, Berlin, 1978, pp. 393–481.
19. HÄRDER, T. Observations on optimistic concurrency control schemes. *Inf. Syst. 9* (June 1984), 111–120.
20. HERLIHY, M. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst. 4*, 1 (Feb. 1986), 32–53.
21. HERLIHY, M. P. Availability vs. concurrency: Atomicity mechanisms for replicated data. *ACM Trans. Comput. Syst. 4*, 3 (Aug. 1987), 249–274.
22. HERLIHY, M. P. Extending multiversion timestamping protocols to exploit type information. *IEEE Trans. Comput. C-35*, 4 (April 1987), 443–449.
23. HERLIHY, M. P., LYNCH, N. A., MERRITT, M., AND WEIHL, W. E. On the correctness of orphan elimination algorithms. In *Proceedings of the 17th Symposium on Fault-Tolerant Computer Systems (FTCS)* (July 1987).
24. HERLIHY, M. P., AND WEIHL, W. E. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM-SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)* (March 1988), pp. 201–210.
25. JACOBSON, D. M. A protocol for optimistic transactions on abstract data types. Tech. Rep. TR 83-12-04, Department of Computer Science, University of Washington, Seattle, 1984.
26. KORTH, H. F. Locking primitives in a database system. *J. ACM 30*, 1 (Jan. 1983), 55–79.
27. KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.
28. LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
29. LAUSEN, G. Concurrency control in data base systems: A step towards the integration of optimistic methods and locking. In *Proceedings of ACM '82*. ACM, New York, 1982.
30. LAUSEN, G. Formal aspects of optimistic concurrency control in a multiversion data base system. *Inform. Syst. 8*, 4 (1983), 291–301.
31. MENASCE, D. A., AND NAKANISHI, N. Optimistic versus pessimistic concurrency control mechanisms in data base management systems. *Inform. Syst. 7*, 1 (1982), 13–27.
32. MOSS, J. E. B. Nested transactions: An approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, April 1981.
33. PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.
34. REED, D. P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst. 1*, 1 (Feb. 1983), 3–23.

35. REIMER, M. Solving the phantom problem by predictive optimistic concurrency control. In *Proceedings of the 9th IFIP Symposium on Very Large Data Bases* (1983).
36. REUTER, A. Concurrency on high-traffic data elements. In *ACM Symposium on Principles of Database Systems*. ACM, New York, 1982, pp. 83–92.
37. SCHWARZ, P. M., AND SPECTOR, Z. Synchronizing shared abstract types. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 223–250.
38. SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Trans. Database Syst. 13*, 1 (March 1988), 53–90.
39. SINHA, M., NANDIKAR, P. D., AND MEHNDIRATTA, S. L. Timestamp-based certification for transactions in distributed database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Austin, Tex., May 1985). ACM, New York, 1985, pp. 402–413.
40. TAY, Y. C., GOODMAN, N., AND SURI, R. Performance evaluation of locking in databases: A survey. Tech. Rep. TR-17-84, Harvard Aiken Laboratory, Cambridge, Mass., 1984.
41. THOMAS, R. H. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst. 4*, 2 (June, 1979), 180–209.
42. WEIHL, W. E. Commutativity-based concurrency control for abstract data types. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences* (Jan. 1988), pp. 205–214.
43. WEIHL, W. E. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst. 11*, 2 (April 1989), 249–282.
44. WEIHL, W. E. Specification and implementation of atomic data types. Tech. Rep. TR-314, M.I.T. Lab Computer Science, Cambridge, Mar. 1984.

Received October 1987; revised April and October 1988; accepted August 1988