Optimistic Concurrency Control for Abstract Data Types

Maurice Herlihy
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
8 May 1986

Abstract

A concurrency control technique is optimistic if it allows transactions to execute without synchronization, relying on commit-time validation to ensure serializability. This paper describes several 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. These techniques have the following novel aspects. First, 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 are derived directly from an object's data type specification. Second, these techniques are 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. Third, when integrated with quorum-consensus replication, these techniques circumvent certain trade-offs between concurrency and availability imposed by comparable pessimistic techniques. Finally, the accuracy and efficiency of validation are further enhanced by some technical improvements: distributed validation is performed as a side-effect of the commit protocol, and validation takes into account the results of operations, accepting certain interleavings that would have produced delays in comparable pessimistic schemes.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520.

This paper appeared in the proceedings of the Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, August, 1987.

1. Introduction

Informally, *optimistic* concurrency control is based on the premise that it is more effective 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 only for applications where reads predominate; they are poorly suited for general-purpose applications such as banking or reservations where 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 are 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 (cited below) have shown that the success rate of validation depends critically on the nature and frequency of transaction conflict. In large systems, it is reasonable to expect that different objects will have different patterns of conflict, and that individual objects' patterns may change over time. These observations suggest that optimistic techniques are cost-effective only under specialized circumstances, while pessimistic techniques are more robust. If optimistic techniques are to be useful in general-purpose systems, it must be possible to apply them selectively in conjunction with appropriate pessimistic techniques. (See Lausen [19] for a similar argument.) Even for pessimistic techniques, however, the compatibility of distinct mechanisms is a non-trivial question. For example, two-phase locking [8] and multiversion timestamping [24] 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.

Optimistic and pessimistic methods behave differently when integrated with quorum-consensus replication [15]. Pessimistic techniques trade concurrency for availability; weakening the constraints

on one may tighten the constraints on the other [13, 14]. Optimistic techniques are different: enhancing validation to accept more interleavings has no effect on availability.

The accuracy and efficiency of validation are further enhanced by two technical improvements. First, distributed validation requires no additional messages, since it is performed as a side-effect of the commit protocol. Second, because operations are validated after they have occurred, validation can take into account the results of invocations, permitting certain interleavings to be validated that would have produced delays in comparable locking schemes.

This paper is organized as follows. Section 2 surveys some related work, and Section 3 describes a model of computation. Section 4 describes *conflict-based* validation, a simple validation technique based on predefined conflicts. Section 5 describes a scheme that permits pessimistic and optimistic techniques to be combined in a single object. Section 6 describes *state-based* validation, a more complex scheme that validates additional interleavings by exploiting knowledge about the object's state. Section 7 examines how optimism interacts with replication, and Section 8 closes with a discussion.

2. Related Work

Perhaps the earliest concurrency control scheme to use validation is that of Thomas [26]. Kung and Robinson [17] have proposed a centralized optimistic method based on Read/Write conflicts. Ceri and Owicki [4] have extended Kung and Robinson's method to permit validation in distributed systems. Lausen [19] 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 [20]. Härder [12] has distinguished 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.

IMS/VS [10] uses an optimistic technique to reduce contention for shared counters. Like the more general techniques proposed in this paper, IMS/VS mixes pessimistic and optimistic techniques, and exploits type-specific properties of counters to make validation effective.

A system of *logical clocks* [18] keeps track of orderings and dependencies among events. Timestamps generated by logical clocks provide a simple and efficient technique for extending the natural partial order of events in a distributed system to an arbitrary total order. 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 [4].

Numerous studies have compared the performance of pessimistic and optimistic techniques [1, 2, 3, 9, 21, 25]. These studies have yielded a variety of conclusions, some in apparent disagreement. Nevertheless, one particular conclusion seems justified: the effectiveness of optimistic techniques depends on the distribution of conflicts in subtle and complex ways. In general-purpose distributed systems, where such predictions may be difficult and miscalculation expensive, optimistic techniques are most likely to be useful if they can be applied to individual objects rather than to entire systems.

Weihl [27] 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 [8, 16, 22], as well as schemes that combine locking with timestamps [5, 6, 14].

3. Assumptions and Definitions

Distributed systems are subject to two kinds of faults: sites may crash and communication links may be interrupted. A widely-accepted technique for preserving consistency in the presence of failures and concurrency is to organize computations as sequential processes called *transactions*. Transactions are *atomic*, that is, serializable and recoverable. Serializability means that transactions appear to execute in a serial order [23], and recoverability means that a transaction either succeeds completely, or has no effect. A transaction that completes all its changes successfully *commits*; otherwise it *aborts*, and any changes it has made are undone. A transaction that has neither committed nor aborted is *active*. Some form of atomic commitment protocol [7, 11] ensures that commits are atomic, and non-volatile storage ensures that changes are not destroyed by later failures.

The basic containers for data are called *objects*. Each object has a *type*, which defines a set of possible *states* and a set of primitive *operations* that provide the (only) means to create and manipulate objects of that type. An *event* is a pair consisting of an operation invocation and a response. For example, a bank account might be represented by an object of type Account whose state is given by a non-negative 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 (Overdraft).
```

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 *successful debit*, otherwise it is an *attempted overdraft*.

An object's state is modeled by a sequence of events called a *history*. For example,

Credit($5)/Ok()
Credit($6)/Ok()
Debit($10)/Ok()
Debit($2)/Overdrawn()

is a history for an Account. A *specification* for an object is the set of permissible histories for that object. For example, the specification for an Account object consists of histories in which the balance covers any successful debit, and fails to cover all unsuccessful debits. A *legal history* is one that is included in the object's specification. Histories are denoted by lower-case letters.

In the presence of failure and concurrency, an object's state is given by a *schedule*, which is a sequence of operation executions, *transaction commits*, and *transaction aborts*. To keep track of interleaving, a transaction identifier is associated with each step in a schedule. For example, the following is a schedule for an Account:

Credit($5)/Ok() A
Credit($6)/Ok() B
Commit A
Debit($10)/Ok() B
Commit B

Here, A and B 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. Schedules are denoted by upper-case letters.

(Serial) histories and (concurrent) schedules are related by the notion of *atomicity*. Let $\gg$ denote a total order on committed and active transactions, and let H be a schedule. The *serialization* of H in the order $\gg$ is the history h constructed by reordering the events in H so that if $B \gg A$ then the subsequence of events associated with A precedes the subsequence of events associated with B. H is *serializable in the order* $\gg$ 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. An object is atomic if all of its schedules are atomic.

A *system* encompassing multiple objects is atomic if all component objects are atomic and serializable in a common order. The optimistic mechanisms introduced in this paper serialize transactions in the order they commit (as observed by a system of logical clocks [18]), and are thus compatible with pessimistic methods that induce the same ordering (e.g., [5, 6, 8, 14, 16, 22]). Following the terminology of Weihl [27], a schedule is *hybrid atomic* if it is serializable in commit order.

4. Conflict-Based Validation

This section introduces *conflict-based validation*, an optimistic concurrency control mechanism which uses predefined conflicts between pairs of events for validation. This approach is the optimistic analog of locking mechanisms, which use similar predefined conflicts to introduce delays. A precise definition of conflict is given below, but for now it is enough to say that two transactions that execute no conflicting events can be serialized in either order, thus neither can invalidate the other. (This notion is weaker than commutativity, which requires that both serializations define equivalent states.)

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

Each transaction is validated during the first phase of commitment. When an object receives the prepare message, it validates the transaction locally (using techniques described below) before recording the transaction's intentions list on non-volatile storage. If all participants validate the transaction, the co-ordinator issues the timestamped commit messages. An object can validate transactions concurrently if neither transaction's events conflict with the other's, but the object must apply the intentions lists in the order of commit. Validation requires no messages in addition to those needed for the standard commit protocol.

The following extension to the two-phase commit protocol ensures that intentions lists are applied in the proper order. When a site receives the *prepare* message from the coordinator, it generates a *prepared timestamp* for that transaction before responding with its acknowledgment. After the coordinator has received acknowledgments from all participants, it generates a *commit timestamp*, which is later than any of its prepared timestamps. The commit timestamp is forced to stable storage along with the commit record, and is included with each *commit* message to participants. A site may apply a transaction's intentions as soon as it has processed commit or abort messages from all transactions with earlier prepared timestamps. Note that no messages have been added to the standard commit protocol.

This section considers two distinct validation techniques [12]: *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. Under conflict-based validation, the two approaches have comparable run-time

costs.

4.1. Serial Dependency

This section gives a formal characterization of what it means for events to conflict. Let $\succ$ be a relation between pairs of events, and let h be a legal history. A legal subhistory g of h is *closed* under $\succ$ if whenever it contains an event e it also contains every earlier event e' of h such that $e \succ e'$. A subhistory g is a *view* of h for e under $\succ$ if g is closed under $\succ$, and if g contains every e' of h such that $e \succ e'$. Informally, $\succ$ is a *serial dependency* relation if whenever an event is legal for a view, it is legal for the complete history. More precisely, let "•" denote concatenation:

> **Definition 1:** A relation $\succ$ is a *serial dependency* relation if $g \bullet e$ is legal implies that $h \bullet e$ is legal, for all events e and all legal histories h, such that g is a view of h for e under $\succ$.

The optimistic techniques proposed here are correct if and only if conflict between events is defined by a serial dependency relation. Of primary interest are *minimal* relations, having the property that no smaller relation is also a serial dependency relation. As discussed below in Section 7, serial dependency is also important for quorum-consensus replication.

The Account data type has a unique minimal serial dependency relation, shown in Table 4-1. Here, successful debits do not depend on prior credits, because the debit cannot be invalidated by increasing the balance. Attempted overdrafts do depend on prior credits, however, because the Overdraft exception can be invalidated by increasing the balance. The FIFO Queue data type has two distinct minimal serial dependency relations, shown in Tables 4-2 and 4-3. (Here, Deq blocks when the queue is empty.) In the first relation, Enq events depend on no other events, but Deq events depend on all other events. In the second relation, Enq events depend on one another, Deq events depend on one another, but Enq events do not depend on Deq events, and vice-versa.

The next two sections present formal models for forward and backward validation, together with proofs of correctness and optimality. Correctness means that an object whose conflict relation is a serial dependency relation will validate only hybrid atomic schedules, and optimality means that an object whose conflict relation is *not* a serial dependency relation will validate some schedule that is not hybrid atomic.

4.2. Forward Validation

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

	Credit/Ok	Debit/Ok	Debit/Over
Credit/Ok			
Debit/Ok		X	
Debit/Over	X		

Table 4-1: Serial Dependency Relation for Account

	Enq/Ok	Deq/Ok
Enq/Ok		
Deq/Ok	X	X

Table 4-2: First Serial Dependency Relation for Queue

	Enq/Ok	Deq/Ok
Enq/Ok	X	
Deq/Ok		X

Table 4-3: Second Serial Dependency Relation for Queue

An 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, EVENT is the set of events, TIMESTAMP is a totally ordered set of timestamps with minimal element $\perp$. The derived domain HISTORY is the set of sequences of events. $X \rightharpoonup Y$ denotes the set of partial maps from X to Y.

A *forward validation automaton* has the following state components:

Perm: HISTORY

Intentions: TRANS $\rightharpoonup$ HISTORY

O-Lock: EVENT $\rightarrow 2^{\text{TRANS}}$

Clock: TIMESTAMP

Committed: 2^{TRANS}

Aborted: 2^{TRANS}

The *Perm* component represents the object's permanent state, initially empty. *Intentions(A)* is the

sequence of events executed by transaction *A*, initially none. *O-Lock(e)* is the set of active transactions that hold an optimistic lock for *e*, initially none. The *Clock* component models a system of logical clocks. *Committed* and *Aborted* keep track of the transactions that have committed and aborted; each is initially empty.

Each transition has a precondition and a postcondition. In postconditions, primed component names denote new values, and unprimed names denote old values. For transaction *A* to execute event *e*,

Pre: $A \notin$ Committed.

Perm•Intentions(A)•e is legal.

Post: Clock' > Clock

Intentions'(A) = Intentions(A)•e

O-Lock'(e) = O-Lock(e) ∪ {A}.

The transition can occur only if the transaction has not already committed, and if the operation appears to be legal. The transition causes the clock to be advanced, the event to be appended to the transaction's intentions list, and the transaction to be given an optimistic lock for the event.

Validation is governed by a *conflict relation* $\succ_O \subseteq \text{EVENT} \times \text{EVENT}$. For *A* to commit,

Pre: $A \notin$ Committed ∪ Aborted.

If e is in Intentions(A) and $e \succ_O e'$ then O-Lock(e')—{A} = ∅.

Post: Clock' > Clock

Perm' = Perm•Intentions(A)

O-Lock'(e) = O-Lock(e)—{A}

A transaction may commit only if it has not already committed or aborted, and only if no other transaction holds an optimistic lock for a conflicting event. Afterwards, the clock is advanced, the transaction's intentions list is appended to the permanent state, and the optimistic locks are released. Finally, a transaction may abort only if it has not already committed. When a transaction aborts, it is added to the set of aborted transactions and its optimistic locks are released.

The first step toward proving correctness is the following lemma, which states that any sequence of events can be inserted into the middle of a history provided no later event serially depends on an inserted event.

Lemma 2: If $\succ$ is a serial dependency relation, *f*, *g*, and *h* histories such that *f*•*g* and *f*•*h*

are legal, and there is no e in h and e' in g such that $e \succ e'$, 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 $h = h' \cdot e$. $f \cdot h'$ is a view of $f \cdot g \cdot h'$ for e, because $f \cdot g \cdot h'$ is legal by the induction hypothesis and $f \cdot h'$ is legal by assumption. Because $f \cdot h' \cdot e$ is legal and $\succ$ is a serial dependency relation, $f \cdot g \cdot h' \cdot e = f \cdot g \cdot h$ is legal.

The following lemma states that 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 3: For any forward validation automaton whose conflict relation is a serial dependency relation, *Perm•Intentions(A)* is legal for all active A.

Proof: The argument proceeds by induction on the number of transactions that have committed, showing that *Perm•Intentions(A)* remains legal when another transaction A' commits. By the induction hypothesis, *Perm'* = *Perm•Intentions(A')* is legal. The precondition for the commit of A' implies that there is no e in *Intentions(A)* and e' in *Intentions(A')* such that $e \succ_O e'$. Because $\succ_O$ is a serial dependency relation, *Perm'•Intentions(A)* is legal by Lemma 2.

The correctness theorem for forward validation is a direct consequence of Lemma 3:

Theorem 4: A forward validation automaton whose conflict relation is a serial dependency relation will accept only hybrid atomic schedules.

Proof: *Perm* is the serialization in commit order of the schedule accepted by the automaton, and Lemma 3 implies that each commit carries *Perm* from one legal state to another.

Serial dependency is optimal in the following sense: if a forward validation automaton's conflict relation is not a serial dependency relation, then it will accept some schedule that is not hybrid atomic. The proof is based on the following lemma, which states that if $\succ$ is *not* a serial dependency relation, then there exists a counterexample in which the view is missing exactly one event.

Lemma 5: If $\succ$ is not a serial dependency relation, then there exist a history h and an event e such that h has a view g of e missing exactly one event, $g \cdot e$ is legal, but $h \cdot e$ is not.

Proof: If $\succ$ is not a serial dependency relation, then there exist a history h and an event e such that h has a view g of e where $g \cdot e$ is legal, but $h \cdot e$ is not. Suppose g is missing k events of h. Consider the sequence of histories $\{h_i \mid i = 0,...,k\}$, where $h_0 = g$, $h_k = h$, and h_{i+1} is derived from h_i by restoring its earliest missing event.

If there exists an i such that h_i is legal but h_{i+1} is not, then there exist histories a, b, and c, and events e_1 and e_2 such that h_i can be written as $a \cdot b \cdot e_2 \cdot c$, and h_{i+1} as $a \cdot e_1 \cdot b \cdot e_2 \cdot c$, where $a \cdot e_1 \cdot b$ is legal, and $a \cdot e_1 \cdot b \cdot e_2$ is not. But $a \cdot b$ is a view of $a \cdot e_1 \cdot b$ for e_2, proving the lemma.

Otherwise, suppose h_i is legal for all i between 0 and k. Because $h_0 \cdot e$ is legal and $h_k \cdot e$ is not, there must exist an i such that $h_i \cdot e$ is legal but $h_{i+1} \cdot e$ is not. This h_i is a view of h_{i+1} for e, proving the lemma.

Lemma 5 provides a simple proof of optimality.

Theorem 6: Any forward validation automaton whose conflict relation is not a serial dependency relation will accept a schedule that is not hybrid atomic.

Proof: Since $\succ_O$ is not a serial dependency relation, there exist by Lemma 5 an event e and a legal history h such that g is a view of h for e missing exactly one event e', $g \cdot e$ is legal, but $h \cdot e$ is not. Let $g = a \cdot b$ and $h = a \cdot e' \cdot b$. Transaction A executes a and commits, leaving $Perm = a$. B executes b followed by e, and C executes e'. C commits, followed by B. Both B and C are validated, but the final value of $Perm$ is the illegal history $a \cdot e' \cdot b \cdot e = h \cdot e$.

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(e)*, the most recent commit timestamp for a transaction that executed the event e. For each active transaction A, each object also keeps track of *First(A,e)*, the logical time when A first executed e. An object will validate A if and only if *Last(e')* < *First(A,e)* for each event e' that conflicts with each event e executed by A. This condition ensures that A has not been invalidated by a transaction that committed since A executed e.

To model backward validation, the *O-Lock* component is replaced by:

First: TRANS × EVENT → TIMESTAMP

Last: EVENT → TIMESTAMP

The precondition for A to execute e is unchanged. The postcondition is slightly different: instead of granting an optimistic lock to A, *First(A,e)* is updated if necessary.

First'(A,e) = if (First(A,e) = ⊥)
then Clock
else First(A,e)

For A to commit,

Pre: A ∉ Committed ∪ Aborted.

If e is in Intentions(A) and $e \succ_O e'$ then First(A,e) > Last(e').

Post: Clock' > Clock

Perm' = Perm•Intentions(A)

If e is in in Intentions(A) then Last'(e) = Clock.

A transaction may commit only if no recently committed transaction has executed a conflicting event. Afterwards, the *Last* timestamp is updated for each event executed by the 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:

Lemma 7: For any backward validation automaton whose conflict relation is a serial

dependency relation, *Perm•Intentions(A)* is legal for any valid *A*.

Proof: It is enough to show that if the commit of *A'* does not invalidate *A*, then *Perm•Intentions(A)* remains legal. If *A* remains valid, there is no *e* in *Intentions(A)* and *e'* in *Intentions(A')* such that $e \succ_O e'$, therefore *Perm'•Intentions(A)* is legal by Lemma 2.

The basic correctness theorem for backward validation is a direct consequence of Lemma 7:

Theorem 8: Any backward validation automaton whose conflict relation is a serial dependency relation will accept only hybrid atomic schedules.

Proof: *Perm* is the serialization in commit order of the schedule accepted by the automaton, and Lemma 7 implies that each commit carries *Perm* from one legal state to another.

Serial dependency is also optimal for backward validation:

Theorem 9: Any backward validation automaton whose 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 6.

4.4. Discussion

The Account data type illustrates how a type-specific definition of conflict allows more transactions to commit. 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 to invalidate transactions: a credit can invalidate an overdraft, and a successful debit can invalidate another successful debit.

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. An advantage of forward validation is that all transactions observe serializable states, even those for which validation fails. 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.

How do these optimistic techniques compare to pessimistic locking schemes? In most pessimistic schemes, a lock is acquired before invoking an operation, thus conflicts are typically defined between invocations, not between complete events. In optimistic schemes, by contrast, validation occurs after the invocations' results are known, thus conflicts can be defined between complete events. This additional information can be used to validate interleavings that would be prohibited by invocation

locking. For example, compare the event/event conflict relation for Account in Table 4-1 and the invocation/invocation conflict relation in Table 4-4. Invocation locks for credit and debit must conflict, but conflict-based validation will permit a credit to occur concurrently with a successful debit (but not an attempted 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 unsuccessful debit 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.

	Credit/Ok	Debit/Ok	Debit/Over
Credit/Ok		X	X
Debit/Ok	X	X	X
Debit/Over	X	X	X

Table 4-4: Invocation/Invocation Conflict for Account

5. Mixing Pessimistic and Optimistic Methods

The previous section showed 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 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 event. Pessimistic locks are related by a *pessimistic conflict* relation. If any other transaction holds a conflicting lock, the lock is refused, the event is discarded, and the operation is later retried. (The invocation may return a different result when it is retried.) If the lock is granted, the intentions list is updated, and the response is returned to the client. 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. The fundamental constraint governing an object's optimistic and pessimistic conflict relations is the following: their

union 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. 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:

P-Lock: EVENT $\rightarrow 2^{TRANS}$

P-Lock(e) is the set of transactions that hold pessimistic locks for *e*. Initially, all such sets are empty. Pessimistic lock conflicts are governed by a pessimistic conflict relation $\succ_P \subseteq$ EVENT×EVENT. The precondition for *A* to execute *e* has an additional clause:

If $e \succ_P e'$ or $e' \succ_P e$ then P-Lock(e')—{A} = Ø.

Note that lock conflicts are determined by the symmetric closure of $\succ_P$. Afterwards, the transaction is granted a pessimistic lock for the event.

P-Lock'(e) = P-Lock(e) ∪ {A}

Finally, a transaction's pessimistic locks are released when it commits or aborts.

Pessimistic conflicts prevent concurrent transactions from executing conflicting events:

Lemma 10: For a mixed (forward or backward) validation automaton, if *A* and *A'* are concurrent active transactions, and *e* is an event in *Intentions(A)*, then there is no *e'* in *Intentions(A')* such that $e \succ_P e'$.

Proof: The precondition for *A* to execute *e* ensures that the property holds initially, and it prevents any other transaction from violating the property while *A* is active.

Define a mixed automaton's conflict relation to be $\succ_P \cup \succ_O$.

Lemma 11: For any mixed forward validation automaton whose conflict relation is a serial dependency relation, *Perm•Intentions(A)* is legal for all active *A*.

Proof: As before, it is enough to show that *Perm•Intentions(A)* remains legal after the commit of a distinct transaction *A'*. By the induction hypothesis, *Perm'* = *Perm•Intentions(A')* is legal. There is no *e* in *Intentions(A)* and *e'* in *Intentions(A')* such that $e \succ_O e'$ (Lemma 3) or $e \succ_P e'$ (Lemma 10). Because $\succ_P \cup \succ_O$ is a serial dependency relation, *Perm'•Intentions(A)* is legal by Lemma 2.

Lemma 12: For any mixed backward validation automaton whose conflict relation is a serial dependency relation, *Perm•Intentions(A)* is legal for all valid *A*.

Proof: If *A'* commits without invalidating *A*, there is no *e* in *Intentions(A)* and *e'* in *Intentions(A')* such that $e \succ_O e'$ (Lemma 7) or $e \succ_P e'$ (Lemma 10), therefore *Perm'•Intentions(A)* is legal by Lemma 2.

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

Theorem 13: A mixed forward validation automaton whose conflict relation is a serial dependency relation will accept only hybrid atomic schedules.

Theorem 14: A mixed forward validation automaton whose conflict relation is a not serial dependency relation will accept a schedule that is not hybrid atomic.

Theorem 15: A mixed backward validation automaton whose conflict relation is a serial dependency relation will accept only hybrid atomic schedules.

Theorem 16: A mixed backward validation automaton whose conflict relation is a not serial dependency relation will accept a schedule that is not hybrid atomic.

6. 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 events. Instead, the accuracy of validation can be enhanced only by taking objects' states into account, as in the optimistic counter management scheme of IMS/VS [10]. Such *state-based* validation may be more expensive than conflict-based validation, since it may (at worst) amount to re-executing 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.

Both forward and backward validation can exploit state information. For forward validation, *A* may commit only if:

For all *A'* active and distinct from *A*, Perm•Intentions(A)•Intentions(A') is legal.

For backward validation, *A* may commit only if:

Perm•Intentions(A) is legal.

These formulations reveal an important practical asymmetry between forward and backward state-based validation: forward validation is strictly more expensive. This observation is in contrast to conflict-based validation, where forward and backward validation are roughly equivalent in run-time cost. Consequently, only backward validation is considered in this section.

A predicate *P[e]* on histories is a *validating predicate* for *e* if:

$P[e](h) \Rightarrow h \bullet e$ is legal.

A validating predicate is *optimal* if the implication is mutual. Validating predicates can be extended to histories in the obvious way:

$P[g \bullet e](h) = P[g](h) \wedge P[e](h \bullet g)$

As it executes, each transaction builds up a validating predicate for its intentions list. The transaction is validated by applying this predicate to the object's permanent state.

The cost of conflict-based validation is largely type-independent, but the cost of state-based

validation depends on type-specific properties: how compactly validating predicates can be represented, how efficiently they can be evaluated, and how many additional interleavings they validate. The following idealized implementation of an Account provides an "existence proof" that state-based validation can be effective for certain data types. An Account is modeled as an automaton with the following components:

Bal: INT

Clock: TIMESTAMP

Low: TRANS → INT

Committed: 2^{TRANS}

High: TRANS → INT

Aborted: 2^{TRANS}

Change: TRANS → INT

Bal is the object's permanent state, represented here as a balance. Each transaction's validating predicate is encoded by two quantities: *Low(A)* is the transaction's lower bound on the current balance (initially zero), and *High(A)* is the transaction's current upper bound (initially unbounded). *Change(A)* is the transaction's intentions list, represented here simply as a net change to the balance.

Account events have the following pre- and postconditions. For *A* to execute Debit(k)/Ok(),

Pre: Bal + Change(A) $\geq$ k

Post: Change'(A) = Change(A)—k

Low'(A) = max(Low(A),k—Change(A)),

for Debit(k)/Overdraft(),

Pre: Bal + Change(A) < k

Post: High'(A) = min(High(A),k—Change(A))

and for Credit(k)/Ok(),

Pre: *true*

Post: Change'(A) = Change(A) + k.

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

Low(A) $\leq$ Bal < High(A)

After *A* commits, its changes are applied to the balance.

Bal' = Bal + Change(A)

An inductive argument (not given) shows that the validating predicates employed by this automaton are correct and optimal. Moreover, the run-time cost of validation is comparable to the cost of conflict-based validation for this particular data type.

7. Interaction With Replication

This section gives a brief informal summary of how optimistic techniques interact with quorum-consensus replication [15, 14]. Conflict-based validation extends readily to replication, supporting availability properties identical to those supported by pessimistic conflict-based techniques. State-based validation, however, has an interesting property: it places fewer constraints on availability than pessimistic state-based techniques.

A *replicated object* is an object whose state is stored redundantly at multiple sites. Replicated objects are implemented by two kinds of modules: *repositories* and *front-ends*. Repositories provide long-term storage for the object's state, while front-ends carry out operations for clients. A client executes an operation by sending the invocation to a front-end. The front-end merges the data from an *initial quorum* of repositories, performs a local computation, and records the response at a *final quorum* of repositories. A *quorum* for an operation is any set of sites that includes both an initial and a final quorum for that operation. Each operation's availability is determined by its set of quorums, thus constraints on quorum assignment determine the range of availability properties realizable by replication.

Because state information for replicated objects is distributed, it is convenient for validation to use conflicts between invocations and events rather than between pairs of events. Validation based on invocation/event conflict is less effective, since it uses less information, but it requires less message traffic. The notion of serial dependency is extended to relations between invocations and events as follows. A relation $\succ$ between invocations and events induces a relation $\succ'$ between pairs of events: $e \succ' e'$ if $e.inv \succ e'$, where $e.inv$ denotes the invocation part of the event e. $\succ \subseteq$ INVOCATION $\times$ EVENT is an (invocation/event) *serial dependency relation* if the induced relation between events is an (event/event) serial dependency relation.

For forward validation, each repository maintains an optimistic lock for each invocation. A transaction acquires an optimistic lock at each repository in the invocation's initial quorum. A transaction is validated if and only if no other transaction holds a conflicting optimistic lock at any repository in the transaction's final quorums. For backward validation, each repository keeps track of *First(A,i)*, the logical time when *A* first executed the invocation *i* at that repository, and *Last(e)*, the commit timestamp of the most recent transaction to execute *e* at that repository. A transaction is validated if and only if for each invocation *i* of *A* and each conflicting event *e*: $Last(e) < First(A,i)$ at each repository in the transaction's initial quorums.

The following is a necessary and sufficient correctness condition for both schemes: the intersection of the conflict relation with the quorum intersection relation must be an

(invocation/event) serial dependency relation. (In practice, they would be identical.) This requirement ensures that any conflict will be detected at the non-empty intersection of two quorums, causing validation to fail. This condition is identical to that imposed by *consensus locking* [14], a conflict-based pessimistic scheme.

While conflict-based validation can be done at repositories, state-based validation must done at front-ends, because the state information at any individual repository may be incomplete. For simplicity, assume that each transaction uses a single front-end for each object. A transaction is validated by reconstructing the permanent state from the transaction's initial quorums and applying the validation predicate for its intentions list. Here, a necessary and sufficient correctness condition is that the quorum intersection relation be a serial dependency relation.

State-based validation provides a way to circumvent certain trade-offs between concurrency and availability imposed by pessimistic state-based methods. Using pessimistic techniques, concurrency is enhanced at the cost of tightening constraints on quorum assignment. This trade-off is best illustrated by an example; a systematic treatment appears elsewhere [13, 14]. Consider an Account replicated among n identical sites. Consensus locking permits $\lceil n/2 \rceil$ distinct quorum assignments: Debit requires any m sites, where $m > n/2$, and Credit requires any $n-m+1$ sites. A more complex pessimistic scheme that takes full advantage of state information permits exactly one quorum assignment: both Credit and Debit require a majority of sites. The interesting observation here is that state-based validation permits all $\lceil n/2 \rceil$ quorum assignments, yet it validates every interleaving permitted by the more restrictive pessimistic scheme. The disadvantage of pessimistic schemes is that they must perform the equivalent of both forward and backward validation: each event must be legal when appended to its transaction's view, and the event must not invalidate any concurrent transaction's view. In short, after-the-fact conflict detection places fewer constraints on quorum assignment than dynamic conflict avoidance.

8. Conclusions

I have proposed two reasons why conventional optimistic techniques are inappropriate for general-purpose distributed systems. First, such techniques are typically designed for database applications in which read operations predominate, an implausible assumption for many non-database applications. Second, such techniques are typically monolithic, applying to all the data encompassed within a system, an undesirable property in an open-ended, decentralized distributed system. This paper has proposed new techniques to address these limitations:

- 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.

- 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 such as two-phase locking can be used to resolve "high-risk" conflicts.

- 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.

- Forward and backward conflict-based validation have comparable run-time costs, but backward state-based validation is easier than forward state-based validation.

- Conflict-based validation is readily integrated with quorum-consensus replication, at a slight loss in concurrency. State-based validation, however, places fewer constraints on availability than pessimistic methods that support a comparable level of concurrency.

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

Acknowledgments

I would like to thank Beth Bottos, Dean Daniels, Dan DuChamp, Ellen Siegel, and Bill Weihl for comments and for help in tracking down citations.

References

[1] R. Agrawal.
Concurrency control and recovery in multiprocessor database machines: design and performance evaluation.
PhD thesis, University of Wisconsin, 1983.

[2] D. Z. Badal.
Concurrency Control overhead or closer look at blocking vs. non-blocking concurrency control mechanisms.
In *Proceedings of the 5th Berkeley Workshop*, pages 55-103. 1981.

[3] M. Carey.
Modeling and Evaluation of Database Concurrency Control Algorithms.
PhD thesis, University of California, Berkeley, September, 1983.

[4] S. Ceri and S. Owicki.
On the use of optimistic methods for concurrency control in distributed databases.
In *Proceedings of the 6th Berkeley Workshop*, pages 117-130. 1982.

[5] A. Chan, S. Fox, W. T. Lin, A. Nori, and D. Ries.
The implementation of an integrated concurrency control and recovery scheme.
In *Proceedings of the 1982 SIGMOD Conference*. ACM SIGMOD, 1982.

[6] D. J. Dubourdieu.
Implementation of distributed transactions.
In *Proceedings 1982 Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 81-94. 1982.

[7] C. Dwork and D. Skeen.
The Inherent Cost of Nonblocking Commitment.
In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 1-11. ACM, August, 1983.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger.
The notion of consistency and predicate locks in a database system.
Communications ACM 19(11):624-633, November, 1976.

[9] P. Franaszek, and J. T. Robinson.
Limitations of concurrency in transaction processing.
ACM Transactions on Database Systems 10(1):1-28, March, 1985.

[10] D. Gawlick.
Processing 'hot spots' in high performance systems.
In *Proceedings COMPCON'85*. 1985.

[11] J. Gray.
Notes on Database Operating Systems.
Lecture Notes in Computer Science 60.
Springer-Verlag, Berlin, 1978, pages 393-481.

[12] T. Härder.
Observations on optimistic concurrency control schemes.
Information Systems 9:111-120, June, 1984.

[13] M.P. Herlihy.
Comparing how atomicity mechanisms support replication.
In *Proceedings of the 4th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. August, 1985.

[14] M.P. Herlihy.
Availability vs. atomicity: concurrency control for replicated data.
Technical Report CMU-CS-85-108, Carnegie-Mellon University, February, 1985.

[15] M.P. Herlihy.
A quorum-consensus replication method for abstract data types.
ACM Transactions on Computer Systems 4(1), February, 1986.

[16] H. F. Korth.
Locking primitives in a database system.
Journal of the ACM 30(1), January, 1983.

[17] H.T. Kung and J.T. Robinson.
On optimistic methods for concurrency control.
ACM Transactions on Database Systems 6:213-226, June, 1981.

[18] L. Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7):558-565, July, 1978.

[19] G. Lausen.
Concurrency control in data base systems: a step towards the integration of optimistic methods and locking.
In *Proceedings of ACM '82*. 1982.

[20] G. Lausen.
Formal Aspects of optimistic concurrency control in a multiversion data base system.
Information Systems 8(4):291-301, 1983.

[21] D. A. Menasce, and N. Nakanishi.
Optimistic versus pessimistic concurrency control mechanisms in data base management systems.
Information Systems 7(1):13-27, 1982.

[22] J. E. B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology Laboratory for Computer Science, April, 1981.

[23] C.H. Papadimitriou.
The serializability of concurrent database updates.
Journal of the ACM 26(4):631-653, October, 1979.

[24] D. Reed.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[25] Y. C. Tay, N. Goodman, and R. Suri.
Performance evaluation of locking in databases: a survey.
Technical Report TR-17-84, Harvard Aiken Laboratory, 1984.

[26] R.H. Thomas.
A solution to the concurrency control problem for multiple copy databases.
In *Proc. 16th IEEE Comput. Soc. Int. Conf. (COMPCON)*. Spring, 1978.

[27] W. Weihl.
Data-Dependent concurrency control and recovery.
In *Proceedings of the 2nd annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. August, 1983.