MULTIPLE VERSIONS AND THE PERFORMANCE OF OPTIMISTIC CONCURRENCY CONTROL

by

Michael J. Carey

Computer Sciences Technical Report #517

October 1983

Multiple Versions and the Performance of Optimistic Concurrency Control

Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was supported by AFOSR Grant AFOSR-78-3596, NESC Contract NESC-N00039-81-C-0569, and a California MICRO Fellowship while the author was at the University of California, Berkeley.

Multiple Versions and the Performance of Optimistic Concurrency Control

Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

This paper presents an optimistic concurrency control algorithm which uses multiple versions of data to increase concurrency for read-only transactions. The algorithm is developed incrementally from the original serial validation proposal of Kung and Robinson. The performance of this multiversion serial validation algorithm is investigated and compared to that of its single version counterpart using a simulation model. Results obtained from the model indicate that multiversion serial validation can significantly improve performance for certain transaction mixes.

1. INTRODUCTION

Optimistic concurrency control algorithms have been proposed for use in both centralized and distributed database systems by a number of researchers [Bada79, Casa79, Baye80, Kung81, Bhar82, Ceri82]. Underlying most of these proposals is the belief that locking is an overly pessimistic approach to concurrency control because transactions are blocked in some circumstances where they could actually proceed without compromising serializability. Optimistic algorithms allow transactions to run unimpeded until they reach their commit point, requiring that they be subjected to a test at commit time to ensure that committing a transaction will not compromise the consistency of the database. Transactions failing the commit-time test are restarted to avoid inconsistencies. Various optimistic algorithm proposals differ mainly in the details of their commit-time tests.

The focus of this paper is a modified version of the optimistic concurrency control algorithm which Kung and Robinson refer to as *serial validation* [Kung81]. The original algorithm is reviewed, and then a new optimistic algorithm based on the use of timestamps is presented and shown to have exactly the same semantics as the algorithm of Kung and Robinson. This timestamp-based serial validation algorithm is then extended

to use multiple versions of data to enhance concurrency for read-only transactions. Simulation studies performed using a centralized concurrency control performance model [Care83] are used to demonstrate that multiversion serial validation indeed offers significant performance improvements for mixes of small update transactions and large read-only transactions.

2. SERIAL VALIDATION

The serial validation (SV) algorithm [Kung81] requires that the readsets and writesets of all transactions be recorded as they execute. These readsets and writesets are the sets of items which the transaction reads and writes, respectively. Transactions are allowed to execute freely until commit-time, writing their database changes into a list of deferred updates. Each transaction is subjected to a commit-time validation procedure in a critical section (a section of code which excludes other transactions from making concurrency control requests simultaneously). This validation

```
procedure validate(T);
begin
  valid := true;
  foreach T_rc in RC(T) do
    foreach x_r in readset(T) do
      foreach x_w in writeset(T_rc) do
        if x_r = x_w then
          valid := false;
        fi;
      od;
    od;
  od;
  if valid then
    commit writeset(T) to database;
  else
    restart(T);
  fi;
end;
```

Figure 1: Informal description of original SV algorithm.

procedure is used to ensure that committing the transaction will not leave the database in an inconsistent state.

Let $RC(T)$ be the set of *recently committed* transactions, i.e., those which commit between the time when T starts executing and the time at which T enters the critical section for validation. Transaction T is validated if $readset(T) \cap writeset(T_{rc}) = \phi$ for all transactions $T_{rc} \in RC(T)$. If T is validated, its updates are applied to the database; otherwise, it is restarted. Intuitively, T is allowed to commit if and only if no other transaction has updated any data items which T read during the time while it was performing its reads and computing its database updates. An informal description of the serial validation algorithm is given in Figure 1.

3. TIMESTAMP-BASED SERIAL VALIDATION

To facilitate the addition of multiple versions to the serial validation algorithm, a new version of serial validation with different but provably equivalent semantics will be described. In this version, each transaction is assigned a startup timestamp, $S\text{-}TS(T)$, at startup time, and each transaction receives a commit timestamp, $C\text{-}TS(T)$, when it enters its commit processing phase. A write timestamp, $TS(x)$, is maintained for each data item x; $TS(x)$ is the commit timestamp of the most recent (committed) writer of x. A transaction T will now be allowed to commit if and only if $S\text{-}TS(T) > TS(x_r)$ for each object x_r in its readset. Each transaction T which successfully commits will update $TS(x_w)$ to be $C\text{-}TS(T)$ for all data items x_w in its writeset.

An informal description of the timestamp-based SV algorithm is given in Figure 2. This algorithm is semantically equivalent to the original SV algorithm. The equivalence proof is based on showing that the new algorithm commits exactly those transactions which would be committed by the original SV algorithm, restarting all transactions which it would restart as well.

Lemma 1: All transactions which are committed by the original SV algorithm (O-SV) are also committed by the timestamp-based SV algorithm (T-SV).

```
procedure validate(T);
begin
  valid := true;
  foreach x_r in readset(T) do
    if S-TS(T) < TS(x_r) then
      valid := false;
    fi;
  od;
  if valid then
    foreach x_w in writeset(T) do
      TS(x_w) := C-TS(T);
    od;
    commit writeset(T) to database;
  else
    restart(T);
  fi;
end;
```

Figure 2: Informal description of timestamp-based SV algorithm.

Proof: Suppose some transaction T is committed by O-SV but restarted by T-SV. Let $RC(T)$ be the set of recently committed transactions, those which committed between the time when T started executing and the time at which it entered the validation critical section. Since T is committed by O-SV, it must be true that $readset(T) \cap writeset(T_{rc}) = \phi$ for all transactions $T_{rc} \in RC(T)$. Since T is restarted by T-SV, it must also be true that $TS(x) > S\text{-}TS(T)$ for some $x \in readset(T)$. However, $TS(x) > S\text{-}TS(T)$ implies that x was written by a transaction which committed subsequent to the startup of T, as $TS(x)$ is the commit timestamp of the most recent writer of x and $S\text{-}TS(T)$ is the startup timestamp of T. Thus, x must be in $writeset(T_{rc})$ for some transaction $T_{rc} \in RC(T)$. This contradicts the assumption that O-SV committed T, proving the lemma. ▪

Lemma 2: All transactions which are restarted by the O-SV algorithm are also restarted by the T-SV algorithm.

Proof: Suppose some transaction T is restarted by O-SV but committed by T-SV. Let $RC(T)$ be the set of recently committed transactions, those which committed between the time when T started executing and the time at which it entered the validation critical section. Since T is restarted by O-SV, it must be true that some $x \in readset(T)$ is also in $writeset(T_{rc})$ for some transaction $T_{rc} \in RC(T)$. Since T is committed by T-SV, it must also be true that $TS(x) < S\text{-}TS(T)$ for all $x \in readset(T)$. However, $TS(x) < S\text{-}TS(T)$ implies that x has not been written by any transaction which committed subsequent to the startup of T, as $TS(x)$ is the commit timestamp of the most recent writer of x. Thus, x cannot be in $writeset(T_{rc})$ for any transaction $T_{rc} \in RC(T)$. This contradicts the assumption that O-SV restarted T, proving the lemma. ▪

Theorem: The set of transactions committed by the T-SV algorithm is precisely that set of transactions which would be committed by the O-SV algorithm, so the T-SV algorithm preserves the semantics of the O-SV algorithm.

Proof: The theorem follows directly from a combination of Lemmas 1 and 2. ▪

A consequence of this theorem is that, since the original SV algorithm is known to guarantee serializability [Kung81], the timestamp-based SV algorithm also guarantees serializability.

For typical transaction mixes, it is expected that $RC(T)$ will tend to be larger than one and the writesets of transactions will not be overly large. The timestamp-based SV algorithm will entail less CPU cost than the original SV algorithm for such mixes. In the original version, the commit-time test involves checking $|RC(T)|$ writesets for each object x_r, whereas a single timestamp is checked for each x_r in the timestamp-based version. The timestamp-based SV algorithm involves an additional cost for updating $TS(x_w)$ for each x_w, but this is unlikely to be significant compared to the cost reduction for testing the readset. Thus, the new algorithm is likely to be more efficient than the original version of the algorithm.

4. MULTIVERSION SERIAL VALIDATION

There have been a number of recent papers proposing the use of multiple versions of data to increase potential concurrency [Reed78, Baye80, Stea81, Svob81, Bern82, Chan82]. In most of these algorithms, the idea is to allow long read-only transactions to read older versions of data objects while allowing update transactions to create newer versions. This section describes a multiple version variant of serial validation which is based on ideas borrowed from a proposed multiversion locking algorithm [Chan82].

The multiversion locking algorithm of interest here was proposed for use in the Ada-compatible database management system under development at CCA [Chan82]. This algorithm, which will be referred to as the *CCA version pool* algorithm, uses two-phase locking to synchronize update transactions and allows read-only transactions to run using older versions of data items. The CCA proposal includes schemes for implementing version selection efficiently and for dealing with maintenance and garbage collection of old versions in a bounded buffer pool, but we will only be concerned with the concurrency control aspects of the proposal.

The semantics of the CCA version pool algorithm are actually quite simple, and can be explained as follows. As in timestamp-based serial validation, transactions are assigned startup timestamps when they begin running and commit timestamps when they reach their commit point. In addition, transactions are classified at startup time as being either *read-only* or *update* transactions. When an update transaction reads or writes a data item, it locks the item, as it would in normal two-phase locking, and it reads or writes the most recent version of the item. When an item is written, a new version of the item is created; versions of items are stamped with the commit timestamp of their creator. When a read-only transaction wishes to access an item, no locking is needed. Instead, it simply reads the latest version of the item with a timestamp less than its startup timestamp. Since the timestamp associated with a version is the

commit timestamp of its writer, each read-only transaction T is made to read only versions which were written by transactions which committed before T even began running. Thus, T is serialized after all transactions which committed prior to its startup, but before all transactions which are active but uncommitted during any portion of its lifetime.

The CCA version pool algorithm is an enhancement of a known concurrency control algorithm, two-phase locking, that permits read-only transactions to read older versions of objects. In this way, serializability is guaranteed for update transactions in the usual way, and it is guaranteed for read-only transactions by having them read a consistent set of older versions of data determined by their startup time. Conflicts between read-only transactions and update transactions are eliminated, increasing the level of concurrency which can be achieved using the algorithm. This idea can be applied outside the domain of locking; in particular, it can also be applied to serial validation.

A multiversion SV algorithm can be developed in a manner which follows naturally from the CCA version pool algorithm. Transactions are again classified as read-only or update transactions at startup time. Update transactions record their readsets and writesets, performing the timestamp-based validation test developed in the previous section. As in the CCA version pool algorithm, versions are stamped with the commit timestamp of their creators, and read-only transactions read the latest versions of items with timestamps less than their startup timestamps. As a result, the serializability of update transactions is guaranteed by SV semantics and the serializability of read-only transactions is guaranteed by making sure they read consistent, committed versions of data.

An informal description of this multiversion SV algorithm is given in Figure 3. It is assumed that an appropriate version selection mechanism provides each transaction T with either $x[TS(T)]$ or $x[current]$ when it reads x. $x[TS(T)]$ denotes the most

```
procedure validate(T);
begin
  valid := true;
  if not readOnly(T) then
    foreach x_r in readset(T) do
      if S-TS(T) < TS(x_r) then
        valid := false;
      fi;
    od;
    if valid then
      foreach x_w in writeset(T) do
        TS(x_w) := C-TS(T);
      od;
      commit writeset(T) to database;
    else
      restart(T);
    fi;
  fi;
end;
```

Figure 3: Informal description of multiversion SV algorithm.

recent version of x with a timestamp less than $TS(T)$, and $x[current]$ denotes the most recent committed version of x. The version selection mechanism returns $x[TS(T)]$ in response to a read request from a read-only transaction, and it returns $x[current]$ in response to a read request from an update transaction. It is also assumed that new versions are created and stamped with $C\text{-}TS(T)$ when $writeset(T)$ is committed to the database.

In the course of his experimental work using Cm*, Robinson also suggested and implemented a multiversion variant of serial validation [Robi82a, Robi82b]. The algorithm presented in this paper differs in the versions which are selected for use by read-only transactions. Here a read-only transaction reads the most recent committed version which preceded its startup; in Robinson's implementation it may read somewhat older versions. As Robinson pointed out, his scheme has the property that a read-only transaction which executes after the completion of an update transaction

may not see the effects of the update transaction. This can be a problem, especially in the case where a single user submits both transactions. The algorithm of this paper does not have this property.†

5. PERFORMANCE STUDIES

In this section, the performance characteristics of the SV and multiple version SV algorithms are investigated using a simulation model presented in [Care83]. Before reporting the experimental results, the simulation model and some modeling details for the two algorithms will be described.

5.1. The Simulation Model

This section outlines the structure and details of the simulation model used to evaluate the performance of the two algorithms. The model was actually designed to support the performance evaluation of a variety of centralized concurrency control algorithms [Care83].

5.1.1. The Workload Model

An important component of the simulation model is a transaction workload model. When a transaction is initiated from a terminal in the simulator, it is assigned a workload, consisting of a readset and a writeset, which determines the objects that the transaction will read and write during its execution. Two transaction classes, *large* and *small*, are recognized in order to aid in the modeling of realistic transaction workloads. The class of a transaction is determined at transaction initiation time and is used to determine the manner in which the readset and writeset for the transaction are to be assigned. Transaction classes, readsets, and writesets are generated using the workload parameters shown in Table 1.

†The version selection mechanism presented here is equivalent to a "fix" mentioned by Robinson [Robi82a], but the details of this mechanism are different.

Workload Parameters	
db_size	size of database
gran_size	size of granules in database
num_terms	level of multiprogramming
delay_mean	mean xact restart delay
small_prob	Pr(xact is small)
small_mean	mean size for small xacts
large_mean	mean size for large xacts
small_xact_type	type for small xacts
large_xact_type	type for large xacts
small_size_dist	size distribution for small xacts
large_size_dist	size distribution for large xacts
small_write_prob	Pr(write X | read X) for small xacts
large_write_prob	Pr(write X | read X) for large xacts

Table 1: Workload parameters for simulation.

The parameter *num_terms* determines the number of terminals, or level of multiprogramming, for the workload. The parameter *restart_delay* determines the mean of an exponential delay time required for a terminal to resubmit a transaction after finding that its current transaction has been restarted. The parameter *db_size* determines the number of objects in the database. The parameter *gran_size* determines the number of objects in each granule of the database. When a transaction reads or writes an object, any associated concurrency control request is made for the granule which contains the object. In modeling read and write requests, objects and granules are given integer names ranging from 1 to *db_size* and 1 to $\lceil db\_size / gran\_size \rceil$, respectively. Object i is contained in granule $\lceil (i-1)/gran\_size \rceil + 1$.

The readset and writeset for a transaction are lists of the numbers of the objects to be read and written, respectively, by the transaction. These lists are assigned at transaction startup time. When a terminal initiates a transaction, *small_prob* is used to randomly determine the class of the transaction. If the class of the transaction is small, the parameters *small_mean*, *small_xact_type*, *small_size_dist*, and *small_write_prob* are used to choose the readset and writeset for the transaction as described below. Readsets and writesets for the class of large transactions are deter-

mined in the same manner using the *large_mean*, *large_xact_type*, *large_size_dist*, and *large_write_prob* parameters.

The readset size for a small transaction is determined by the *small_dist* and *small_mean* parameters. The readset size distribution, given by *small_dist*, may be constant, uniform, or exponential. If it is constant, the readset size is simply *small_mean*. If it is uniform, the readset size is selected from a uniform distribution on the range [1, 2*small_mean*]. The exponential case is not used in the experiments of this paper. The particular objects accessed are determined by the parameter *small_xact_type*. This parameter determines the type, either random or sequential, for small transactions. If they are random, the readset is assigned by randomly selecting objects without replacement from the set of all objects in the database. In the sequential case, all objects in the readset are adjacent, so the readset is selected randomly from among all possible collections of adjacent objects of the appropriate size. Finally, given the readset, the writeset is determined as follows using the *small_write_prob* parameter: It is assumed that transactions read all objects which they write ("no blind writes"). When an object is placed in the readset, it is also placed in the writeset with probability *small_write_prob*.

5.1.2. The Queuing Model

Central to the simulation model is the closed queuing model of a centralized database system shown in Figure 4. This model is an extended version of the model of Ries [Ries77, Ries79a, Ries79b]. There is a fixed number of terminals from which transactions originate. When a new transaction begins running, it enters the *startup queue*, where processing tasks such as query analysis, authentication, and other preliminary processing steps are performed. Once this phase of transaction processing is complete, the transaction enters the *concurrency control queue* (or *cc queue*). For serial validation, the transaction is immediately granted permission to access all objects in its readset and writeset (subject to the constraint that it must be validated later). The

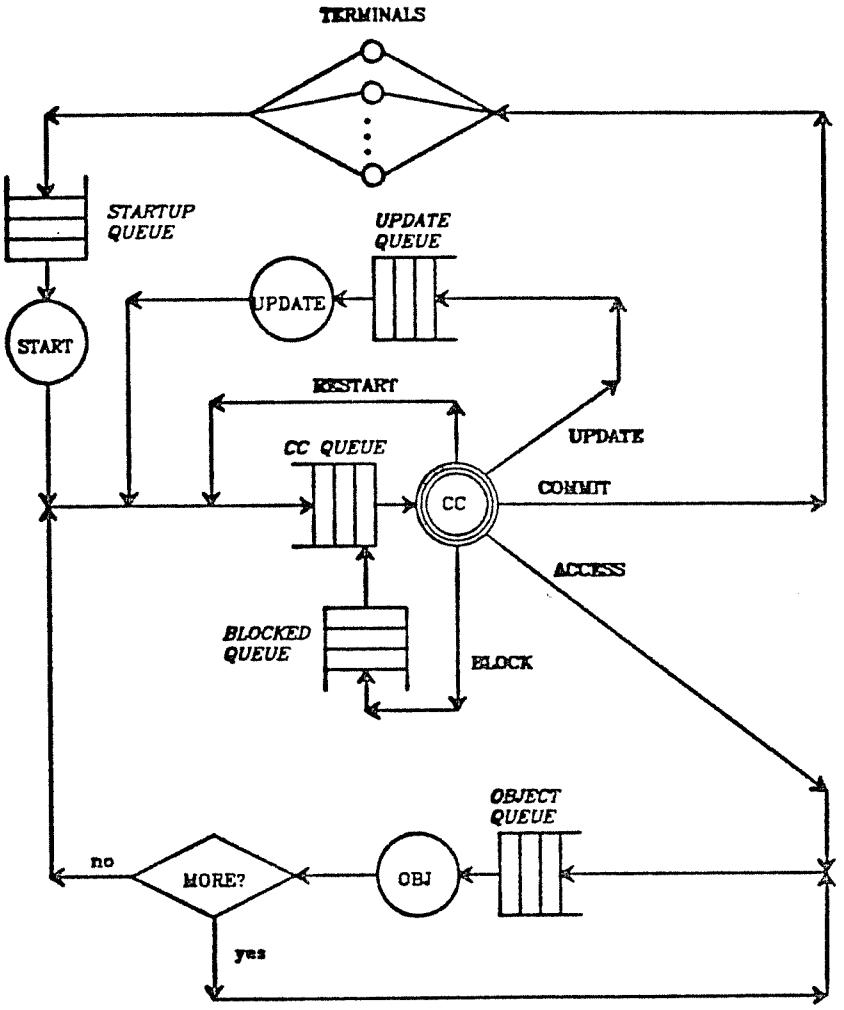

Figure 4: Logical database queuing model.

transaction proceeds to the *object queue* and accesses the objects in its readset and writeset, cycling through this queue once per read or write. It is assumed for convenience that transactions which read and write objects perform all of their reads before performing any writes.

Eventually, the transaction will finish reading and writing and re-enter the concurrency control queue. The SV validation test is applied at this time. If a decision is made to restart the transaction, it goes to the back of the concurrency control queue

after a random restart delay; it then begins running over again, re-reading and re-writing each of the objects in its read and write sets. (It does not repeat its initial startup processing phase, however.) Alternatively, the outcome of validation may be a decision to commit the transaction. In this case, if the transaction is read-only, it is finished. Otherwise, since it has written one or more objects, it must first enter the *update queue* and write its deferred updates into the database.

Underlying the logical model of Figure 4 are two physical resources, the CPU and I/O (disk) resources. Associated with each logical service depicted in the figure (startup, concurrency control, object accesses, etc.) is some use of each of these two global resources. When a transaction enters the startup queue, it first performs its startup-related I/O processing and then performs its startup-related CPU processing. The same is true of each of the other services in the logical model. Each involves I/O processing followed by CPU processing, with the amounts of CPU and I/O per logical service being specified as simulation parameters. All services compete for portions of the global I/O and CPU resources for their I/O and CPU cycles. The underlying physical system model is depicted in Figure 5. As shown, the physical model is simply a collection of terminals, a CPU server, and an I/O server. Each of the two servers has one queue for concurrency control service and another queue for all other service.

The scheduling policy used to allocate resources to transactions in the concurrency control I/O and CPU queues of the underlying physical model is FCFS (first-come, first-served). Concurrency control requests are thus processed one at a time, as they would be in an actual implementation. The resource allocation policies used for the normal I/O and CPU service queues of the physical model are FCFS and round-robin scheduling, respectively. These policies are again chosen to approximately model the characteristics which a real database system implementation would have. When requests for both concurrency control service and normal service are present at either resource, such as when one or more validation requests are pending while other transactions are processing objects, concurrency control service is given priority.

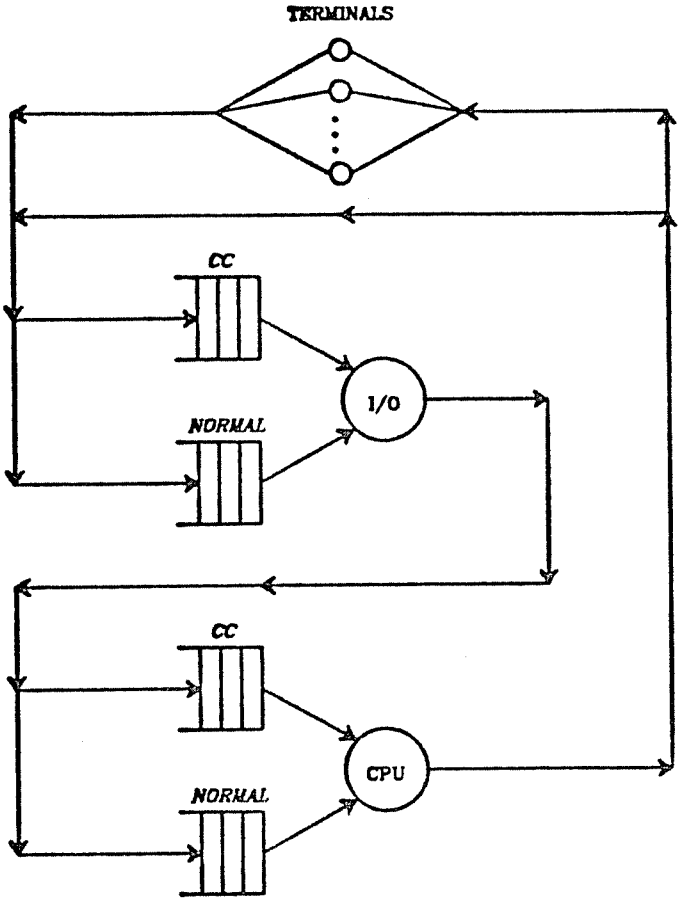

Figure 5: Physical database queuing model.

System Parameters	
startup_io	I/O time for transaction startup
startup_cpu	CPU time for transaction startup
obj_io	I/O time for accessing an object
obj_cpu	CPU time for accessing an object
cc_io	basic unit of concurrency control I/O time
cc_cpu	basic unit of concurrency control CPU time
stagger_mean	mean of exponential randomizing delay

Table 2: System parameters for simulation.

The parameters determining the service times (I/O and CPU) for the various logical resources in the model are given in Table 2. The parameters *startup_io* and *startup_cpu* are the amounts of I/O and CPU associated with transaction startup. Similarly, the parameters *obj_io* and *obj_cpu* are the amounts of I/O and CPU

associated with reading or writing an object. Reading an object takes resources equal to *obj_io* followed by *obj_cpu*. Writing an object takes resources equal to *obj_cpu* at the time of the write request and *obj_io* at deferred update time, as it is assumed that the deferred update list is maintained in buffers in main memory. The parameters *cc_io* and *cc_cpu* are the amounts of I/O and CPU associated with a concurrency control request. All these parameters represent constant service time requirements rather than stochastic ones for simplicity. Finally, the *stagger_mean* parameter is the mean of an exponential time distribution which is used to randomly stagger transaction initiation times from terminals (not to model user thinking) each time a new transaction is started up. All parameters are specified in internal simulation time units, the unit of CPU time allocated to a transaction in one sweep of the round-robin allocation code for the simulator.

5.1.3. Algorithm Descriptions

Concurrency control algorithms are described for simulation purposes as a collection of four routines, *Init_CC_Algorithm*, *Request_Semantics*, *Commit_Semantics*, and *Update_Semantics*. Each routine is written in SIMPAS, a simulation language based on extending PASCAL with simulation-oriented constructs [Brya80a, Brya80b]. SIMPAS is the language in which the rest of the simulator is implemented as well. *Init_CC_Algorithm* is called when the simulation starts up, and it is responsible for initializing all algorithm-dependent data structures and variables. The other three routines are responsible for implementing the semantics of the concurrency control algorithm being modeled. *Request_Semantics* handles concurrency control requests made by transactions before they reach their commit point. *Commit_Semantics* is invoked when a transaction reaches its commit point. *Update_Semantics* is called after a transaction has finished writing its deferred updates. For the algorithms considered in this paper, *Commit_Semantics* is the routine of primary importance, as it implements the validation test. Each of these routines returns information to the simulator about how

much simulation time to charge for CPU and I/O associated with concurrency control processing.

5.2. Algorithm Modeling Issues

The *cc_cpu* and *cc_io* parameters are the basis of the algorithm cost models. For a transaction that makes N_r granule read requests and N_w granule write requests under the SV algorithm, a CPU cost of *cc_cpu* and an I/O cost of *cc_io* are charged for each granule read or written. These charges are assessed together at transaction commit time. The read-related charges model the testing of readset granules to make sure that none have timestamps which indicate that a recently committed transaction wrote the granule, and the write-related charges model the timestamp updating process required when a transaction commits. Thus, the total concurrency control costs in the absence of restarts for SV are $(N_r+N_w)cc\_cpu$ and $(N_r+N_w)cc\_io$. The costs for the multiversion SV algorithm are similar. It has the same costs as single version SV for update transactions, but read-only transactions simply pay a CPU cost of *cc_cpu* and an I/O cost of *cc_io* at transaction startup time. The costs assessed for read-only transactions model the cost of recognizing them as read-only and marking them as such.

In order to simulate the multiple version SV algorithm, it is assumed that old versions of objects are accessible in as little time as the most recent version of each object. This assumption is reasonable if access paths for locating versions of active data items can be kept in primary memory. Such caching of version location information can be probably be achieved using algorithms such as those described in [Chan82]. Otherwise, the results reported here will be optimistic about the degree of performance improvement which is obtainable using the multiversion SV algorithm. The simulation of multiversion SV was implemented by modifying the implementation of SV to always allow read-only transactions to commit.

The system parameters used for this experiment are given in Table 3. Each transaction incurs a startup cost of one 35 millisecond disk access and 10 milliseconds of CPU time. In addition, this same cost is incurred for each read or write of an object. Charges for reading and writing objects are assessed in the manner described in the section which presented the details of the queuing model. The cost associated with each concurrency control request (i.e., each read or write set granule processed at commit time) is 1 millisecond of CPU time. A 20 millisecond random delay time is used to stagger transaction startups.

The workload parameter settings for this experiment are given in Table 4. The database consists of 10,000 objects. The number of terminals used is 10. Small update transactions, which are eighty percent of the mix, each read two objects and then update them each with fifty percent probability. Large transactions, the other twenty percent of the mix, each read a uniformly distributed number of objects sequentially. The mean size of these large read-only transactions is 30.

The results of this experiment are shown in Table 5, where throughput rates for the algorithms are given for various granularities. Transaction restart counts for the simulations which produced these throughput results are given in Table 6. The advantage of multiple versions is fairly pronounced, especially at coarser granularities where the probability of conflicts is significant. The poor performance of SV (compared to

System Parameter Settings	
System Parameter	Time (Milliseconds)
startup_io	35
startup_cpu	10
obj_io	35
obj_cpu	10
cc_io	0
cc_cpu	1
stagger_mean	20

Table 3: System parameters for experiment 1.

Workload Parameters	
db_size	10000 objects
num_terms	10
delay_mean	1 second
small_prob	0.8
small_mean	2 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5
large_mean	30
large_xact_type	sequential
large_size_dist	uniform
large_write_prob	0.0

Table 4: Workload parameters for experiment 1.

multiversion SV) for this mix occurs because transactions are not checked for conflicts until transaction commit time in SV, a practice that strongly biases single version SV against large read-only transactions: They perform all their reads and then test to see if any of the granules have been updated, a likely occurrence with many small update transactions in the mix. Allowing read-only transactions to read older versions of data alleviates this problem. The fact that more restarts actually occur with MVSV than SV for the coarsest granularities is not surprising, as many more small update transactions can run under MVSV (and these are the transactions being restarted here). Note that the penalty associated with restarting small update transactions is much less than that for restarting large read-only transactions because less work must be repeated by small update transactions.

Throughput versus Granularity		
Grans	SV	MVSV
1	0.407±11.60%	2.364±2.61%
10	1.183±8.36%	2.863±2.93%
100	2.397±6.28%	2.999±4.46%
1000	2.691±5.01%	3.012±4.31%
10000	2.755±4.55%	3.013±4.36%

Table 5: Throughput, experiment 1.

Restarts versus Granularity		
Grans	SV	MVSV
1	494	3340
10	545	811
100	214	77
1000	82	7
10000	56	0

Table 6: Restarts, experiment 1.

5.3.2. Experiment 2: Read-Only Transaction Size

This experiment investigates the performance characteristics of the algorithms under a workload similar to that of the previous experiment, but the size of the read-only transactions in the mix is varied here. The purpose of this experiment is to observe the behavior of the algorithms while varying the degree to which old versions may actually be beneficial. The workload parameters used in this experiment were selected in order to emphasize situations in which multiple versions are especially beneficial (where the probability of conflicts between update transactions and read-only transactions is fairly significant in the absence of multiple versions).

The system parameter settings for this experiment are the same as those used for the previous experiment (see Table 3). The workload parameter settings used for this experiment are given in Table 7. The database consists of 100 objects, with a granularity of one object per granule. The number of terminals used is 10. Small update transactions, which are forty percent of the mix, again read two objects and update each with fifty percent probability. Large read-only transactions, the other sixty percent of the mix, sequentially read a uniformly distributed number of objects. In this experiment, the mean size for large transactions is varied from 1 to 30 objects.

The motivation for selecting such a small database size was two-fold. First, it was desired that read-only transactions read a significant fraction of the database so that the probability of conflicts with update transactions would be significant. Second, it was necessary to keep the size of read-only transactions small enough in terms of the

Workload Parameters	
db_size	100 objects
gran_size	1 object/granule
num_terms	10
delay_mean	1 second
small_prob	0.4
small_mean	2 objects
small_xact_type	random
small_size_dist	fixed
small_write_prob	0.5
large_mean	vary from 1 to 30 objects
large_xact_type	sequential
large_size_dist	uniform
large_write_prob	0.0

Table 7: Workload parameters for experiment 2.

number of objects accessed so that reasonably tight confidence intervals could be obtained without using unreasonable amounts of simulation time. This tradeoff led to the selection of a relatively small database size for this experiment. One can also view these parameter settings as an approximation to a large database with much larger read-only transactions (but with fairly coarse granularity).

The throughput results of this experiment are shown in Table 8, and Table 9 gives the associated restart counts. The advantages of multiple versions are again very pronounced due to the bias of SV against large read-only transactions. This experiment illustrates how the relative performance of SV and MVSV varies with the size of the read-only transactions. As one would expect, the performance advantage of MVSV increases as read-only transaction size is increased. Two factors explain this trend:

(1) With larger read-only transactions, more work is lost each time one of them is restarted.

(2) The longer it takes for a read-only transaction to reach its validation point, the more time there is for an update transaction to update an object in its readset; hence, the probability of a successful commit using SV is smaller for larger read-only transactions.

Several conclusions can be drawn from these results. First, the performance of SV is quite poor for workloads where update transactions are likely to conflict with read-only transactions. This performance problem probably makes SV an unreasonable choice for use in many database application environments, as fairly large read-only transactions would be expected to arise from report writers. Second, multiple versions appear to solve this problem quite successfully. In this experiment, when read-only transactions in the mix read an average of 30 objects, MVSV outperformed SV by nearly a factor of three. Were the size of read-only transactions much larger, as one might expect in an actual database system, the performance benefits associated with multiple versions would be even more pronounced.

Throughput versus Read-Only Transaction Size		
Size	SV	MVSV
1	7.386±0.60%	7.669±0.42%
2	6.110±1.41%	6.610±0.60%
5	3.722±1.85%	4.660±1.20%
10	1.957±4.30%	3.177±2.57%
15	1.271±6.06%	2.464±2.35%
30	0.483±10.71%	1.336±4.06%

Table 8: Throughput, experiment 2.

Restarts versus Read-Only Transaction Size		
Size	SV	MVSV
1	536	133
2	618	103
5	706	58
10	654	27
15	562	17
30	373	2

Table 9: Restarts, experiment 2.

5.3.3. Experiment 3: Read-Only Transaction Fraction

This experiment investigates the performance of the algorithms under a workload similar to that of the previous experiment, but the balance between small update tran-

sactions and large read-only transactions in the mix is varied here. The purpose of this experiment is to observe the effects of this balance on the relative performance of the algorithms.

The system parameter settings for this experiment are the same as those used for the two previous experiments (see Table 3). The workload parameter settings for this experiment are the same as those of the previous experiment when *large_mean* = 30 was used (see Table 7). The balance between small update transactions and large read-only transactions in the mix is varied by choosing settings for the *small_prob* parameter, which controls the fraction of update transactions in the mix, ranging from 0.0 to 1.0.

The throughput results of this experiment are shown in Table 10. The restart counts are shown in Table 11. The main interesting result of this experiment is that MVSV outperforms SV most significantly when the workload contains mostly small update transactions. With eighty percent small update transactions in the mix, MVSV outperformed SV by more than a factor of five. An explanation is that, with more update transactions executing during the lifetime of each large read-only transaction, it is more likely that some update transaction will write something that the read-only transaction reads. Hence, the probability that a large read-only transaction will be able to commit upon reaching its validation point is minimized by a workload containing many small update transactions.

Throughput versus Update Transaction Fraction		
Pr(Sm)	SV	MVSV
0.0	0.878±4.67%	0.878±4.67%
0.2	0.540±9.88%	1.043±4.33%
0.4	0.483±10.71%	1.336±4.06%
0.6	0.526±11.15%	1.867±4.81%
0.8	0.546±13.05%	2.943±4.90%
1.0	6.691±0.56%	6.790±0.59%

Table 10: Throughput, experiment 3.

Restarts versus Update Transaction Fraction		
Pr(Sm)	SV	MVSV
0.0	0	0
0.2	286	2
0.4	373	2
0.6	418	23
0.8	453	83
1.0	908	702

Table 11: Restarts, experiment 3.

6. CONCLUSIONS

This paper described an optimistic concurrency control algorithm which uses multiple versions of data to increase concurrency for read-only transactions. The algorithm was developed from the original single version proposal of Kung and Robinson based on ideas derived from the CCA version pool algorithm. The performance of this multiversion serial validation algorithm was compared to that of its single version counterpart using a simulation model. It was found that serial validation performs poorly for mixes of small update transactions and large read-only transactions, but that multiple versions solve this problem. In several cases, multiversion serial validation outperformed single version serial validation by factors of three to five when the mean readset size for read-only transactions was 30 objects. It was also found that the performance advantage of multiple versions was most pronounced for workloads consisting of mostly small update transactions and just a few read-only transactions.

ACKNOWLEDGEMENTS

The author would like to thank Michael Stonebraker for his helpful comments and suggestions on various aspects of this research.

REFERENCES

[Bada79] Badal, D., "Correctness of Concurrency Control and Implications in Distributed Databases", Proceedings of the COMPSAC '79 Conference, Chicago, Illinois, November 1979.

[Baye80] Bayer, R., Heller, H., and Reiser, A., "Parallelism and Recovery in Database Systems", ACM Transactions on Database Systems 5(2), June 1980.

[Bern82] Bernstein, P., and Goodman, N., "Multiversion Concurrency Control Theory and Algorithms", Technical Report No. TR-20-82, Aiken Computation Laboratory, Harvard University, June 1982.

[Bhar82] Bhargava, B., "Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems", Second IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, PA, July 1982.

[Brya80a] Bryant, R., "SIMPAS -- A Simulation Language Based on PASCAL", Technical Report No. 390, Computer Sciences Department, University of Wisconsin-Madison, June 1980.

[Brya80b] Bryant, R., SIMPAS User Manual, Computer Sciences Department and Madison Academic Computing Center, University of Wisconsin-Madison, December 1980.

[Care83] Carey, M., "Modeling and Evaluation of Database Concurrency Control Algorithms", Ph.D. Thesis, Computer Science Division (EECS), University of California, Berkeley, August 1983.

[Casa79] Casanova, M., "The Concurrency Control Problem for Database Systems", Ph.D. Thesis, Computer Science Department, Harvard University, 1979.

[Ceri82] Ceri, S., and Owicki, S., "On the Use of Optimistic Methods for Concurrency Control in Distributed Databases", Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, February 1982.

[Chan82] Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D., "The Implementation of An Integrated Concurrency Control and Recovery Scheme", Proceedings of the ACM SIGMOD International Conference on Management of Data, March 1982.

[Ferr78] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems 6(2), June 1981.

[Reed78] Reed, D., "Naming and Synchronization in a Decentralized Computer System", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[Ries77] Ries, D., and Stonebraker, M., "Effects of Locking Granularity on Database Management System Performance", ACM Transactions on Database Systems 2(3), September 1977.

[Ries79a] Ries, D., "The Effects of Concurrency Control on Database Management System Performance", Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1979.

[Ries79b] Ries, D., and Stonebraker, M., "Locking Granularity Revisited", ACM Transactions on Database Systems 4(2), June 1979.

[Robi82a] Robinson, J., "Design of Concurrency Controls for Transaction Processing Systems", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1982.

[Robi82b] Robinson, J., "Experiments with Transaction Processing on a Multi-Microprocessor", Report No. RC9725, IBM Thomas J. Watson Research Center, December 1982.

[Sarg76] Sargent, R., "Statistical Analysis of Simulation Output Data", Proceedings of the Fourth Annual Symposium on the Simulation of Computer Systems, August 1976.

[Saue81] Sauer, C., and Chandy, N., Computer Systems Performance Modeling, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Stea81] Stearns, R., and Rosenkrantz, D., "Distributed Database Concurrency Controls Using Before-Values", Technical Report, SUNY Albany, February 1981.

[Svob81] Svoboda, L., "A Reliable Object-Oriented Repository for a Distributed Computer System", Proceedings of the Eighth Symposium on Operating Systems Principles, Pacific Grove, California, December 1981.

[Wolf83] Wolff, R., personal communication.