THE PERFORMANCE OF MULTIVERSION CONCURRENCY CONTROL ALGORITHMS

by

Michael J. Carey and Waleed A. Muhanna

Computer Sciences Technical Report #550

August 1984

The Performance of Multiversion Concurrency Control Algorithms

Michael J. Carey[†]
Waleed A. Muhanna[*]

[†]Computer Sciences Department
University of Wisconsin
Madison, WI 53706

[*]Graduate School of Business
University of Wisconsin
Madison, WI 53706

This research was supported by an IBM Faculty Development Award and by the Wisconsin Alumni Research Foundation. Earlier related work was supported by AFOSR Grant AFOSR-78-3596, NESC Contract NESC-N00039-81-C-0569, and a California MICRO Fellowship at the University of California, Berkeley.

The Performance of Multiversion Concurrency Control Algorithms

Michael J. Carey[†]
Waleed A. Muhanna[*]

[†]Computer Sciences Department
University of Wisconsin
Madison, WI 53706

[*]Graduate School of Business
University of Wisconsin
Madison, WI 53706

ABSTRACT

A number of multiversion concurrency control algorithms have been proposed in the past few years. These algorithms use previous versions of data items in order to improve the level of achievable concurrency. This paper describes a simulation study of the performance of several multiversion concurrency control algorithms, investigating the extent to which they provide increases in the level of concurrency, and also the CPU, I/O, and storage costs resulting from the use of multiple versions. The performance of the multiversion algorithms are compared with each other and also with their single version counterparts. It is shown that the multiversion algorithms provide significant performance improvements despite the additional disk accesses involved in accessing old versions of data, and that the storage overhead for maintaining old versions that might be required by ongoing transactions is fairly small.

1. INTRODUCTION

A number of papers proposing the use of multiple versions of data to increase the level of concurrency in database systems have appeared in the literature [Reed78, Baye80, Stea81, Chan82, Robi82, Care83b, Reed83]. The basic idea in all of these proposals is to maintain one or more old versions of objects in the database in order to allow work to proceed using both the current version and older versions. Some of these algorithms maintain just one old version of an object [Baye80, Stea81], whereas other algorithms are designed to utilize potentially many versions of an object [Reed78, Chan82, Robi82, Care83b, Reed83]. For most of the algorithms in the latter class, the idea is to permit long, read-only transactions to read older versions of objects while allowing update transactions to concurrently create newer versions. It is this latter class of multiversion algorithm,

---

This research was supported by an IBM Faculty Development Award and by the Wisconsin Alumni Research Foundation. Earlier related work was supported by AFOSR Grant AFOSR-78-3596, NESC Contract NESC-N00039-81-C-0569, and a California MICRO Fellowship at the University of California, Berkeley.

those that do not limit the number of versions in the database, that we address in this paper.

In addition to the papers proposing various new algorithms, multiversion concurrency control algorithms have been the subject of several recent theoretical papers [Bern83, Papa84]. Serializability theory has been extended to include multiversion algorithms, and it has been shown that multiversion algorithms are able to provide strictly more concurrency than single version algorithms. An issue that has not received very much attention yet is the *performance* of multiversion algorithms. In this paper we describe a simulation study of several multiversion algorithms in which several performance and storage issues are addressed. Among the questions studied are:

(1) To what extent do multiple versions provide increases in the level of achievable concurrency that can be exploited in "real" database systems?

(2) How do the CPU and I/O costs associated with locating and accessing old versions affect overall performance?

(3) How severe are the storage costs for maintaining multiple versions?

The answers to these questions are investigated for three multiversion algorithms: Reed's multiversion timestamp ordering algorithm [Reed83], which is based on timestamps; the CCA version pool algorithm [Chan82], which is based on two-phase locking; and a multiversion serial validation algorithm [Care83b], which is based on the optimistic concurrency control algorithm of Kung and Robinson [Kung81]. The performance of the algorithms are compared both with each other and with the performance of their single version counterparts.

Few previous studies have addressed these questions. Several recent papers by Lin and Nolte included throughput results for multiversion timestamp ordering [Lin83a, Lin83b], and one of these papers also included throughput results for an "optimistic" variant of the CCA version pool algorithm in which transactions postpone setting write locks until commit time [Lin83b]. In both papers, however, transaction sizes were quite small (mean sizes ranged from 1 to 32 objects), and the sizes for read-only and update transactions in the mix were identically distributed. In addition, their study was based on models of a *distributed* database system, making it hard to separate the effects of having

multiple versions of data from those of having distributed data. A recent paper by Peinl and Reuter [Pein83] included results for the before-value locking scheme of [Baye80], but these results were based on synthetic performance metrics (the number of restarts and the average number of non-blocked transactions, not throughput or response time). In addition, the nature of the results was strongly related to the fact that the algorithm is a before-value algorithm, permitting only two copies of any data object. Finally, none of these studies have examined the response time or storage overhead characteristics of multiversion concurrency control algorithms.

In this paper we try to overcome the shortcomings of these previous studies. The performance of the multiversion concurrency control algorithms is examined in a centralized database setting so as to isolate the effects of multiple versions on performance. Also, the performance and overheads of the algorithms are analyzed using a variety of metrics. Among the metrics employed are throughput, average response time, number of disk accesses per read, restart ratio, and space required for old versions. Two classes of transactions, each with independently determined characteristics, are used in the study. Several of the performance metrics are examined on a per-class basis as well as an aggregate basis.

2. ALGORITHMS STUDIED

This section briefly describes each of the three algorithms studied in this paper. The descriptions are sketchy, but hopefully sufficient to give the reader the basic idea in each case. For more details, the reader is encouraged to refer to the original papers in which the algorithms were proposed. This section also includes a description of the version maintenance scheme proposed for use with the CCA version pool concurrency control algorithm, as we have used this scheme for maintaining the set of old versions for each of the multiversion concurrency control algorithms that we have studied.

2.1. Multiversion Timestamp Ordering (MVTO)

In this paper, we consider a simplified version of Reed's original proposal [Reed78, Reed83]. In particular, we consider the algorithm as implemented in the SWALLOW data repository project at MIT [Reed83]. This version of the algorithm can be viewed as a multiversion variant of the basic timestamp ordering algorithm (BTO) of Bernstein and Goodman [Bern81]: Write requests are synchronized using basic timestamp ordering on the most recent versions of objects, while read requests are always granted (possibly using old versions of objects).

The basic timestamp ordering algorithm, used for write requests, works as follows: Each transaction T has a startup timestamp, $S\text{-}TS(T)$, which is issued when T begins executing. The most recent version of an item X in the database has a read timestamp, $R\text{-}TS(X)$, and a write timestamp, $W\text{-}TS(X)$, which record the startup timestamps of the youngest reader and the writer (respectively) of this version of X. A write request from T for X is granted only if $S\text{-}TS(T) \geq R\text{-}TS(X)$ and $TS(T) \geq W\text{-}TS(X)$. Transactions whose write requests are not granted are restarted. Once a write request is granted, it is considered *pending* until the writer commits. When a read or write request is granted for a object with a pending write request, the read or write request is blocked until the pending write is no longer pending (i.e., until the writer either commits or aborts).

Read requests are never rejected, though they may sometimes be blocked due to pending write requests. Each version of an object X is marked with $W\text{-}TS(X)$, the startup timestamp of its creating transaction. Read requests from a transaction T for an object X are granted by allowing the transaction to read the most recent version of X such that $S\text{-}TS(T) \geq W\text{-}TS(X)$. Note that although T must have started running more recently than the writer of this version of X, the writer may still be running as well. This is the case which requires a read request to be blocked for some period of time. Also note, however, that pure read-only transactions will *never* be restarted for any reason.

2.2. The CCA Version Pool Algorithm (VP)

The CCA version pool algorithm [Chan82] is a multiversion variant of two-phase locking (2PL), and it works as follows. Each transaction T is assigned a startup timestamp $S\text{-}TS(T)$ when it

begins running and a commit timestamp $C\text{-}TS(T)$ when it reaches its commit point. Also, 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, just as it would in 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, and every version of an item is stamped with the commit timestamp of its creator.

When a read-only transaction T wishes to access an item, no locking is required. Instead, the transaction simply reads the most recent version of the item with a timestamp less than $S\text{-}TS(T)$. Since the timestamp associated with a version is the commit timestamp of its writer, a read-only transaction T is thus made to only read 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.

2.3. Multiversion Serial Validation (MVSV)

The multiversion serial validation algorithm [Care83b] is based on the optimistic concurrency control algorithm of Kung and Robinson [Kung81] known as serial validation (SV). In their algorithm, transactions record their read and write sets as they run. A transaction is restarted at its commit point if any granule in its readset has been written by a transaction which committed during its lifetime. One difference between their algorithm and our version of the algorithm is that we use timestamps to efficiently check for readset/writeset conflicts instead of storing old write sets and explicitly testing for readset/writeset intersections [Care83b]. Each transaction is assigned a startup timestamp, $S\text{-}TS(T)$, at startup time, and a commit timestamp, $C\text{-}TS(T)$, when it later 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. Each transaction T is validated at commit time, being 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 that successfully commits sets $TS(X_w)$ equal to $C\text{-}TS(T)$ for all data items X_w in its writeset.

The CCA version pool algorithm is a multiversion algorithm which enhances a known concurrency control algorithm, two-phase locking, by permitting read-only transactions to read older version of objects. In this way, serializability is guaranteed for update transactions in the usual way, and serializability 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 same idea can be applied to yield a multiversion variant of serial validation.

In multiversion serial validation [Care83b], transactions are again classified as being either read-only or update transactions at startup time. Update transactions record their readsets and writesets and perform commit-time conflict testing, and versions are stamped with the commit timestamp of their creator (as above). As in the CCA version pool algorithm, read-only transactions read the most recent 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. Read-only transactions thus do not have to undergo a validation test in multiversion serial validation, and they are never restarted. (A similar multiversion optimistic concurrency control algorithm is discussed in [Robi82].)

2.4. Maintaining Old Versions

For all three of the algorithms studied here, versions are maintained using a slightly simplified version of the scheme proposed in the CCA version pool paper [Chan82]. Basically, the physical database is divided into two parts, the main segment and the version pool. The main segment contains the current versions of all of the objects in the database, and the version pool contains older versions of database objects. The version pool objects are organized in a large circular buffer with slots numbered from 0 to $vp\text{-}size - 1$. Versions of objects are chained in reverse chronological order, and version pool slots are allocated sequentially. Figure 1 depicts the main segment of the database, the version pool, and a version chain for an object X.

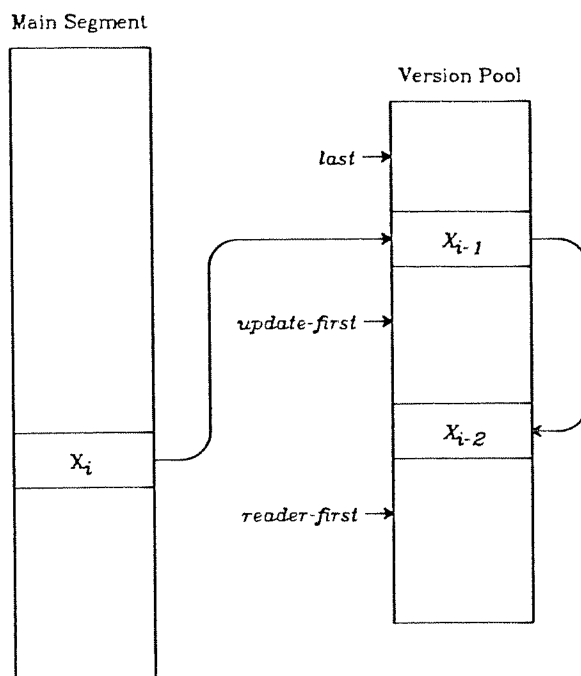

Figure 1: Storing Multiple Versions.

The reclamation of free version pool space is handled efficiently by using the CCA algorithm for maintaining sliding ranges of version pool slots that are in use [Chan82]. Three pointers, *reader-first*, *update-first*, and *last*, where *reader-first* ≤ *writer-first* ≤ *last* (modulo *vp-size*), are used to maintain these sliding ranges. Slots between *reader-first* and *last* contain versions of objects that may be needed to satisfy a read request for some ongoing transaction. Slots between *update-first* and *last* contain object versions that have been written by an ongoing or recently committed update transaction. The objects in this latter range are those objects that may be required to undo the effects of an ongoing update transaction if it is restarted, so this section of the version pool also serves as an UNDO log [Gray79] for recovery purposes [Chan82]. The simplification referred to earlier is that the maximum size of the version pool is made sufficiently large in our simulations so as to avoid problems that arise when the version pool size reaches its maximum limit. In addition, our approach to version selection is based on timestamps rather than the completed transaction lists of [Chan82], slightly simplifying the implementation while preserving the desired semantics.

Of the three algorithms studied, two were designed to be used with this version management scheme [Chan82, Care83b]. Reed proposed a scheme where versions with timestamps newer than some threshold value are kept, and older versions are discarded [Reed83]. Depending on the threshold setting, this scheme may require that some read-only transactions be aborted. We opted to use the CCA version management scheme for multiversion timestamp ordering as well for several reasons. First, it is simple, efficient, and allows us to ignore the problems of a finite version pool limit if we choose the version pool size appropriately for our simulations. Second, Reed's proposal does not update objects in place, which is unacceptable from a database performance standpoint. (Reed's proposal was aimed at providing a transaction-oriented object storage facility in an operating system, not at solving database problems.) Finally, we felt that this would facilitate a fairer comparison of the algorithms, and would allow us to address the question of how much storage is required to maintain all versions that may be needed by in-progress transactions.

3. THE SIMULATION MODEL

This section outlines the structure and details of the simulation model which was used to evaluate the performance of the algorithms. The model was designed to support performance studies for a variety of centralized concurrency control algorithms [Care83a, Care84].

3.1. The Workload Model

An important component of the simulation model is a transaction workload model. When a transaction is initiated in the simulator, it is assigned a readset and a writeset. These determine 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 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 are to be assigned. Transaction classes, readsets, and writesets are generated using the workload parameters shown in Table 1.

Workload Parameters	
mpl	multiprogramming level
restart-delay	mean xact restart delay
db-size	size of database
small-prob	Pr(xact is small)
small-mean	mean size for small xacts
small-xact-type	type for small xacts
small-size-dist	size distribution for small xacts
small-write-prob	Pr(write X | read X) for small xacts
large-mean	mean size for large xacts
large-xact-type	type for large xacts
large-size-dist	size distribution for large xacts
large-write-prob	Pr(write X | read X) for large xacts

Table 1: Workload parameters for simulation.

The parameter *mpl* determines the level of multiprogramming for the workload. The parameter *restart-delay* determines the mean of an exponential delay time required before a transaction can be resubmitted after being restarted during its current execution. The parameter *db-size* determines the number of objects in the database, and objects are represented by integer names ranging from 1 to *db-size*. Objects correspond to disk pages throughout this paper.

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 determined in a similar manner using the parameters *large-mean*, *large-xact-type*, *large-size-dist*, and *large-write-prob*.

The readset size distribution for small transactions is given by *small-dist*. It 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 chosen from a uniform distribution on the range from 1 to 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*, which 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*.

3.2. The Queuing Model

Central to the detailed simulation approach used here is the closed queuing model of a single-site database system shown in Figure 2. This model is an extended version of the model of Ries [Ries77, Ries79a, Ries79b]. There are 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 modeled. Once this phase of transaction processing is complete, the transaction enters the *concurrency control queue* (or *cc queue*) and makes the first of its concurrency control requests. If this request is granted, the transaction proceeds to the *object queue* and accesses its first object. If more than one object is to be accessed prior to the next concurrency control request, the transaction will cycle through this queue several times. When the next concurrency control request is required, the transaction re-enters the concurrency control queue and makes the request. It is assumed for convenience that transactions which read and write objects perform all of their reads before performing any writes.

If the result of a concurrency control request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed. If a request leads to a decision to restart the transaction, it goes to the back of the concurrency control queue after a randomly determined restart delay period of mean *restart-delay*; it then begins making all of its concurrency control requests and object accesses over again. Eventually, the transaction may complete and the concurrency control

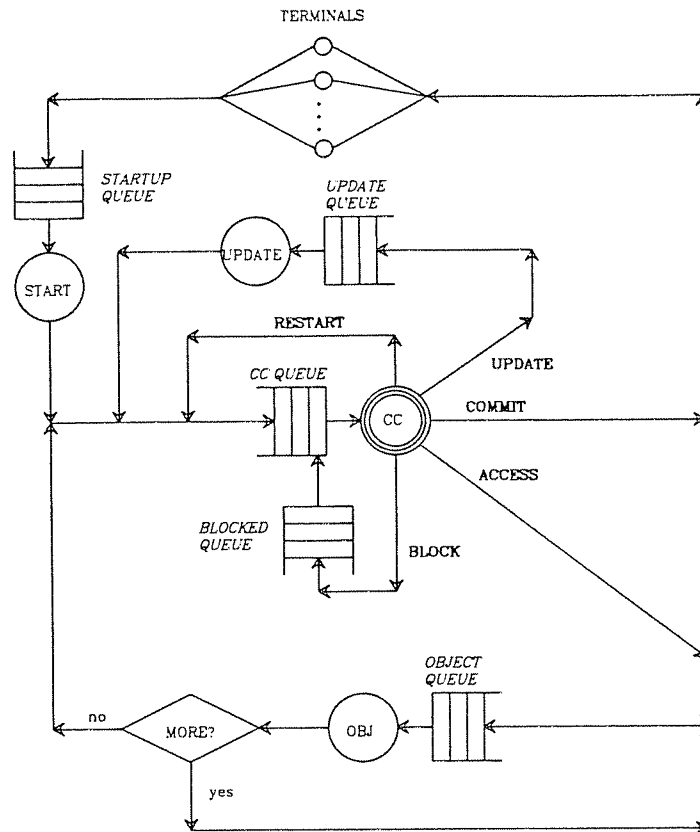

Figure 2: Logical database queuing model.

algorithm may choose to commit the transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution, however, it must first enter the *update queue* and write its updates into the database. (It is assumed that sufficient main memory exists to allow updates to be cached in main memory until end-of-transaction.) When a transaction finally does commit, it is immediately replaced by a new transaction.

Underlying the logical model of Figure 2 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 resources — each involves I/O processing followed by CPU processing. The amounts of CPU and I/O per logical service are 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 3. 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 also FCFS. These policies are again chosen to approximately model the characteristics which a real database system implementation would have. Since a transaction never requests CPU time for processing more than one page in a cycle through the CPU queue, this is approximately equivalent to a round-robin CPU scheduling policy where the quantum exceeds the page processing time. When requests for both concurrency control service and normal service are present at either resource, such as when one or more lock requests are pending while other transactions are processing objects, concurrency control service is given priority.

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, *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 requires the same resources as reading an object, but the time when the *obj-io* portion of the cost is assessed is different. A cost of *obj-cpu* is charged at the time of the write request and a cost of *obj-io* is charged later, at transaction completion time. (It is assumed that updates reside in main memory buffers until being flushed out when the transaction commits.) The parameters *cc-io* and *cc-cpu* are the amounts of I/O and CPU associ-

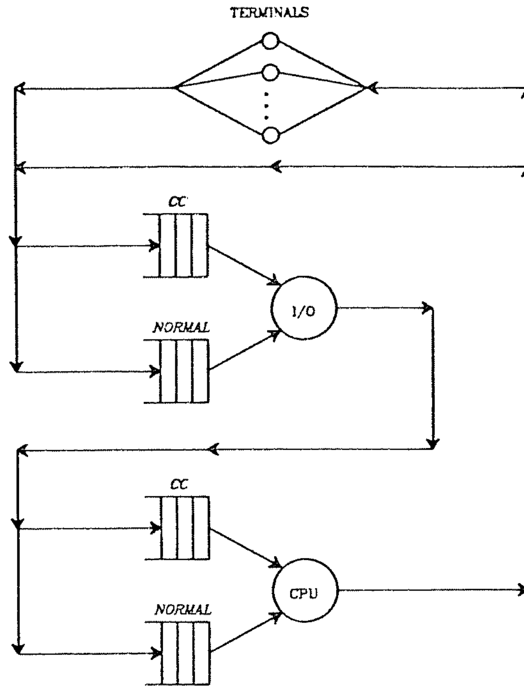

Figure 3: Physical database queuing model.

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

Table 2: System parameters for simulation.

ated with a concurrency control request. All these parameters represent constant service time requirements rather than stochastic ones for simplicity. All parameters are specified in internal simulation time units, which can be interpreted in whatever manner is desired. In the studies reported here, one simulation time unit will represent one millisecond of simulated time.

3.3. 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 [Brya80], the language in which the simulator itself is implemented. *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 out its updates. Each of the latter three routines returns information to the simulator about how much simulation time to charge for CPU and I/O associated with concurrency control processing.

3.4. Concurrency Control Costs

In order to simulate the concurrency control algorithms of interest, it is necessary to make some assumptions about their costs. This section briefly describes how the simulation cost parameters are used in modeling the costs for each of the multiversion algorithms in order to evaluate them using the simulation model.

It is assumed that transactions are issued startup timestamps (for all three algorithms) and registered as being either read-only or update transactions (for VP and MVSV) at transaction startup time. The cost of doing so is assumed to be included in the *startup-cpu* and *startup-io* costs. Read-only transactions incur no additional concurrency control costs in VP or MVSV, but read-only transactions in MVTO and update transactions in all three algorithms will incur costs for setting locks, checking timestamps, or performing validation tests (depending on the algorithm being considered). These latter costs are charged at the points where the algorithms require the associated actions. It is assumed that the costs for the actions of setting a lock, checking a transaction timestamp, or

performing a validation test step are all equal, and each results in charges of *cc-cpu* and *cc-io* per action [Care84].

When a transaction accesses a version of an object, costs of *obj-io* and *obj-cpu* are assessed for each disk access required. The number of disk accesses required to satisfy a read request using an old version of an object depends on whether or not there is a pending (uncommitted) update for the object and on the number of versions accessed. If there is no pending update for the object, one disk access is required to access the current version, two are required to access the second most current version, three are required for the most recent version before that, and so on. If there is a pending update for the object, we assume that the address for the object's before-image in the version pool is stored in the lock or timestamp table in main memory along with its concurrency control state. In this case, one disk access is required for either the current version or the second most recent version, two accesses are required for the most recent version before that, and so on. Thus, in effect, the cost of accessing the most recently committed version of an object is always one, even if it happens to be the before-image of a version undergoing an update. Other read and write charges are assessed as described previously in the discussion of the queuing model.

In addition to the costs for concurrency control processing and following version chains, multiversion algorithms incur costs for version maintenance. When an object is to be updated, the before-image of the object must be read from the main segment and written into the version pool before the actual update can be permitted. In this paper we ignore this source of costs. Our argument for doing so is based on the fact (mentioned earlier) that this copying cost occurs instead of the UNDO logging portion of the normal recovery costs for a transaction processing system. Since we chose not to include other recovery costs, such as UNDO and REDO logging costs for the single version algorithms or REDO logging costs for the multiversion algorithms, we chose also to ignore the cost of this before-image copying.

3.5. Statistical Analysis

In the simulation experiments reported here, one of the main performance metrics used is the transaction throughput rate. Mean throughput results and 90% confidence intervals for these results were obtained from the simulations using a variant of the *batch means* [Sarg76] approach to simulation output analysis. The approach used is due to Wolf [Wolf83]; it differs from the usual batch means approach in that an attempt is made to account for the correlation between adjacent batches. Briefly, we assume that adjacent batches are positively correlated, that non-adjacent batches are uncorrelated, and that the correlation between a pair of adjacent batches is independent of the pair under consideration. We then estimate this correlation and use it in computing a confidence interval for the mean throughput. In the remainder of this paper, we omit most confidence interval data for brevity, presenting just mean throughput figures. However, we only point out performance differences which are significant in the sense that their confidence intervals do not overlap. More information on the statistical approach used in our experiments may be found in Appendix 3 of [Care83a].

4. EXPERIMENTS AND RESULTS

In this section we present the results of three experiments designed to examine the performance and storage characteristics of the multiversion concurrency control algorithms. Experiment 1 examines the algorithms under the type of workload for which they are expected to be beneficial, a mix of small update transactions and large read-only transactions. Experiment 2 investigates the effects of the fraction of update transactions in the mix on the degree of benefit obtained. Experiment 3 investigates the relative performance of the three multiversion algorithms for a workload consisting of larger update transactions, to see how the algorithms behave when update conflicts are likely to occur.

Table 3 contains the settings for parameters which are fixed throughout all of the three experiments. The database size used for the experiments is 500 pages. This (rather small) size was chosen to allow the use of transaction size values that represent a significant fraction of the database without requiring prohibitively long simulation times. The multiprogramming level for the experiments is

ten, so ten transactions will be in the system at all times (although some of these transactions may be blocked or waiting to be restarted). Transactions incur costs of a 35 millisecond disk access and 10 milliseconds of CPU time at startup time. The unit cost for a concurrency control decision is 1 millisecond of CPU time but no I/O.[†] The cost associated with accessing a page is a 35 millisecond disk access and 10 milliseconds of CPU time to process the page. Finally, restarted transactions are delayed for an exponential period with a mean of 1 second.

4.1. Experiment 1: Read-Only Transactions

This experiment examines the behavior of the algorithms under a mix of transactions for which the multiple version algorithms were designed to be beneficial. The mix used here consists of update transactions and read-only transactions. Update transactions are small, and the size of read-only transactions is varied from small to very large (with respect to the overall database size). We study the relative performance of each of the multiversion algorithms, first as compared to their single version counterparts, and then as compared to each other. The performance metrics used are throughput and response time (both aggregate and by transaction class). Also studied are the size of the version pool and the number of disk accesses required to satisfy read requests from transactions.

The workload parameters for Experiment 1 are shown in Table 4. 80% of the transactions in the mix are small, reading two randomly-chosen pages and updating each one with 50% probability.

Fixed Parameter Settings	
db-size	500 pages
mpl	10
startup-cpu	10 ms
startup-io	35 ms
cc-cpu	1 ms
cc-io	0 ms
obj-cpu	10 ms
obj-io	35 ms
restart-delay	1000 ms

Table 3: Fixed Parameters.

[†] It is assumed that all concurrency control information is kept in tables in main memory.

The other 20% of the transactions in the mix are read-only transactions. Each read-only transaction reads a uniformly-distributed number of sequential pages, and the mean size of these transactions is varied from 1 to 250 pages as shown in the table. With these parameter settings, conflicts between update transactions are unlikely. In the single version case, however, conflicts between update transactions and read-only transactions are quite likely for the larger read-only transaction sizes. The point of this experiment is to see how much is gained by having multiple versions available to eliminate this latter source of conflicts.

Figure 4 shows the overall transaction throughput rate (in transactions per second of simulated time) versus the mean size of read-only transactions in the mix. The three multiversion algorithms all performed identically with respect to throughput. The explanation for this is that, given the small size of the update transactions in this experiment, almost all conflicts are between read-only and update transactions. All three of the multiversion algorithms totally eliminate this source of conflicts, so the three algorithms provide the same throughput. Of the single version algorithms, two-phase locking (2PL) has the highest throughput, basic timestamp ordering (BTO) is next, and serial validation (SV) has the worst overall throughput of the algorithms studied. It is evident from Figure 4 that 2PL has only slightly lower throughput than the three multiversion algorithms, and only for the very largest read-only transaction sizes. Thus, as far as throughput is concerned, it would appear that multiple versions are not a significant source of improved performance for 2PL. For BTO and SV,

Workload Parameters	
small-prob	0.8
small-mean	2
small-write-prob	0.5
small-xact-type	random
small-size-type	fixed
large-mean	1, 5, 10, 25, 50, 100, 250
large-xact-type	sequential
large-size-type	uniform
large-write-prob	0.0

Table 4: Workload Parameters, Experiment 1.

however, the addition of multiple versions does indeed improve throughput — the throughputs for MVTO and MVSV are several times larger than those for BTO and SV (respectively) when the mean size of read-only transactions is large.

The throughput results for the six algorithms are explained by Figure 5. This figure shows the restart ratio for each algorithm, where the restart ratio is defined as the ratio of the number of transactions restarted to the number of transactions which started running for the first time. The three multiversion algorithms have extremely small restart ratios, 2PL has a slightly higher restart ratio, BTO has a restart ratio which is much greater than that for 2PL, and SV has the highest restart ratio among the algorithms studied. The greater the restart ratio for an algorithm, the more resources that are wasted due to restarted transactions; this necessarily leads to a lower overall throughput [Care83a, Care84]. The reason for the high restart ratios for BTO and SV are that both algorithms are quite biased against large read-only transactions: When a large read-only transaction tries to read an object in BTO, it is restarted if the object was written by an update transaction that started up more recently. The same is true for SV, though the restart will occur after the large read-only transaction has finished all of its work. Such restarts become more and more likely as read-only transaction size is increased, as more update transactions can run during the lifetime of each read-only transaction in the mix.

Figure 6 shows the overall average response times obtained for the six algorithms in this experiment. The relationships between the algorithms are similar to those for throughput — the three multiversion algorithms have the same average response time, 2PL has the next best response time, and then come BTO and SV with significantly worse response times. There is an interesting change in the relative performance of 2PL and VP here, however. Although the throughput for 2PL is not much lower than that of VP, the response time for 2PL is about 25% worse than that of VP for the largest read-only transaction size studied. This shows that, while VP does not offer improved performance when measured in terms of throughput, it does lead to an improved average response time by reducing the amount of blocking due to lock conflicts between read-only and update transactions.

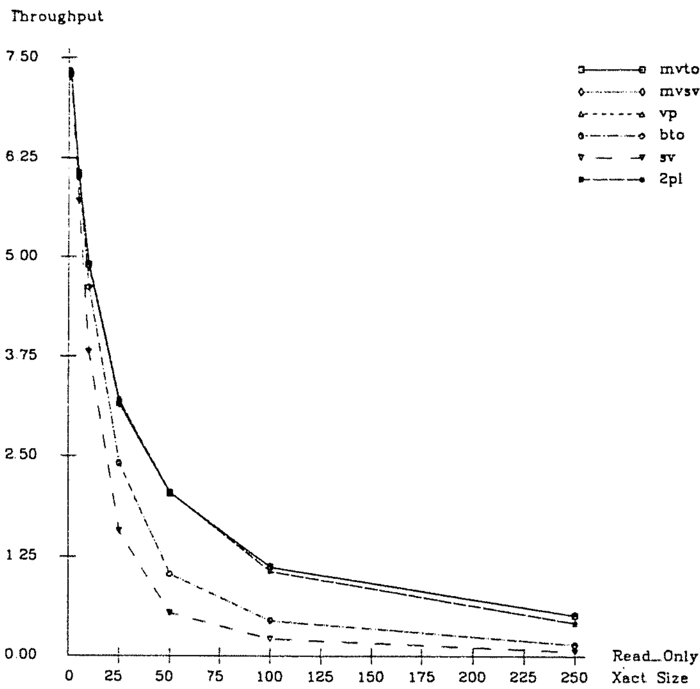

Figure 4: Overall Throughput.

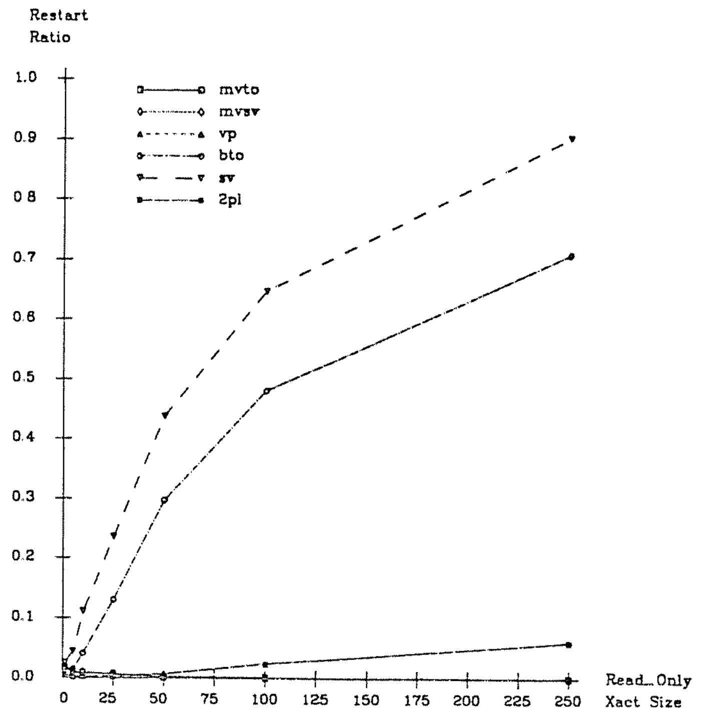

Figure 5: Restart Ratio.

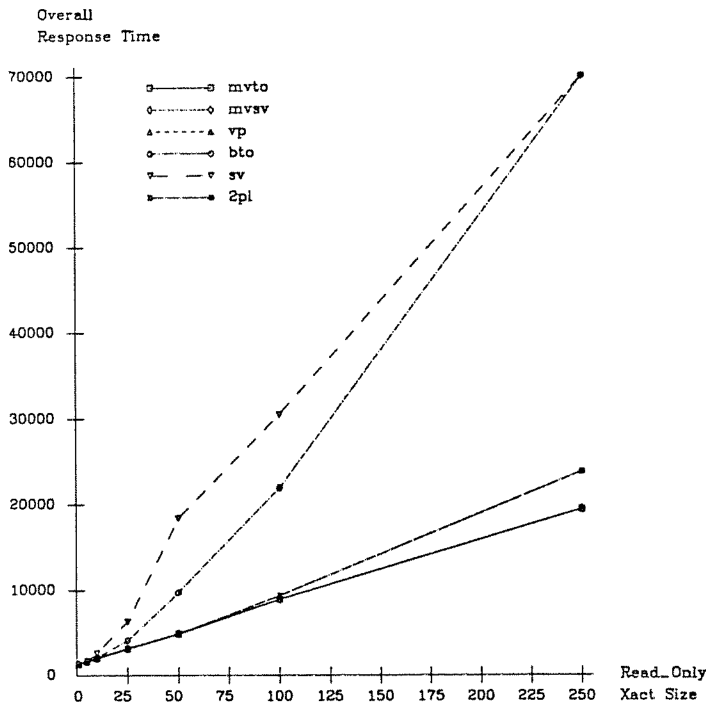

Figure 6: Overall Response Time.

Figures 7 through 10 give throughputs and response times by transaction class, showing how the six algorithms treat the small update transactions and large read-only transactions in the mix. Figures 7 and 8 show that the algorithms have the same per-class throughput curves, in terms of their shape and their relative position to one another, as was seen above in the overall throughput curves of Figure 4. The multiversion algorithms perform identically, 2PL is just slightly worse, and BTO and then SV have significantly lower throughputs. However, there are several enlightening differences between Figures 9 and 10, the per-class response time curves, and the overall average response time curves of Figure 6. In Figure 9, with the exception of 2PL, all of the algorithms provide about the same response time for the small update transactions. This is because, except for 2PL, small update transactions never have to wait for large read-only transactions to complete before performing their updates. For 2PL, however, the response time is much worse — this is because small update transactions are waiting for large read-only transactions to release locks, and this waiting time goes up as the size of read-only transactions is increased. In spite of this increased waiting time, though, Figure 7 shows that the throughput of small update transactions does not suffer.

A second interesting observation concerning the per-class response times of the algorithms comes from Figure 10. The response time for large read-only transactions is actually slightly better for 2PL than for the three multiversion algorithms. This is because, in the multiversion algorithms, large read-only transactions incur extra disk accesses — updates occur while they execute, causing some of the data read by these transactions to be old versions of data. More disk accesses, in turn, lead to longer average response times. In going from 2PL to VP, then, one is trading much improved response times for small update transactions for slightly worse response times for large read-only transactions. The fact that the three multiversion algorithms provide higher overall throughputs and lower overall response times than the three single-version algorithms, and yet large read-only transactions have to do more work, illustrates a key point — the decrease in the number of restarts (see Figure 5) due to reader-updater conflicts for the multiversion algorithms provides more of a resource savings than the amount of resources lost due to the extra disk accesses for reading old versions.

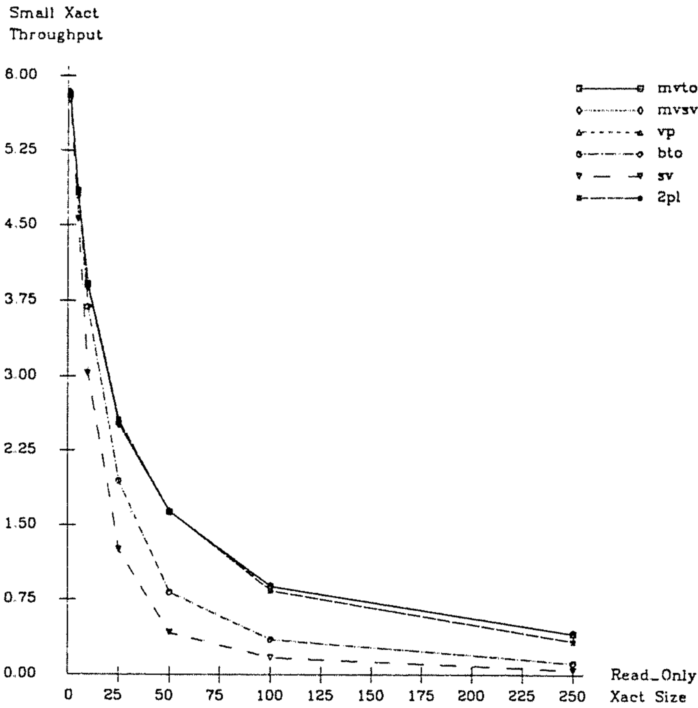

Figure 7: Small Xact Throughput.

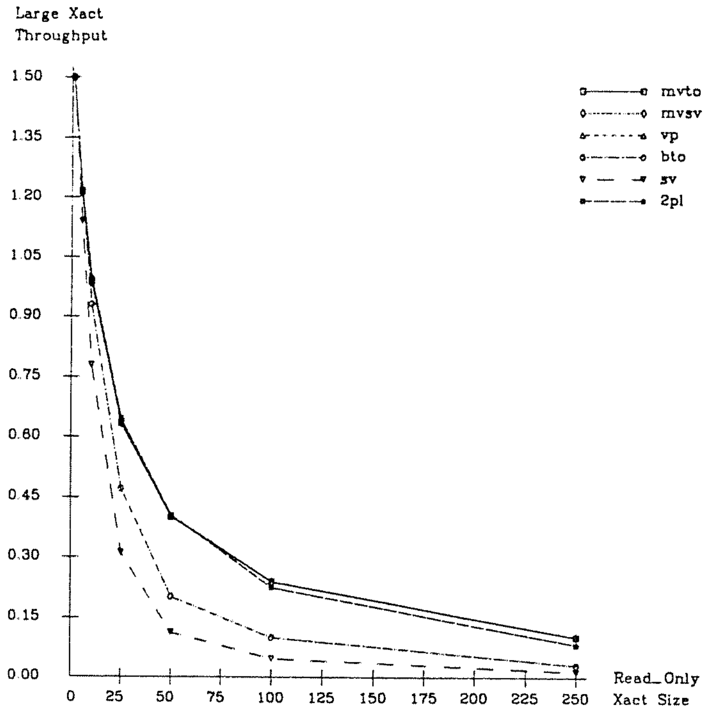

Figure 8: Large Xact Throughput.

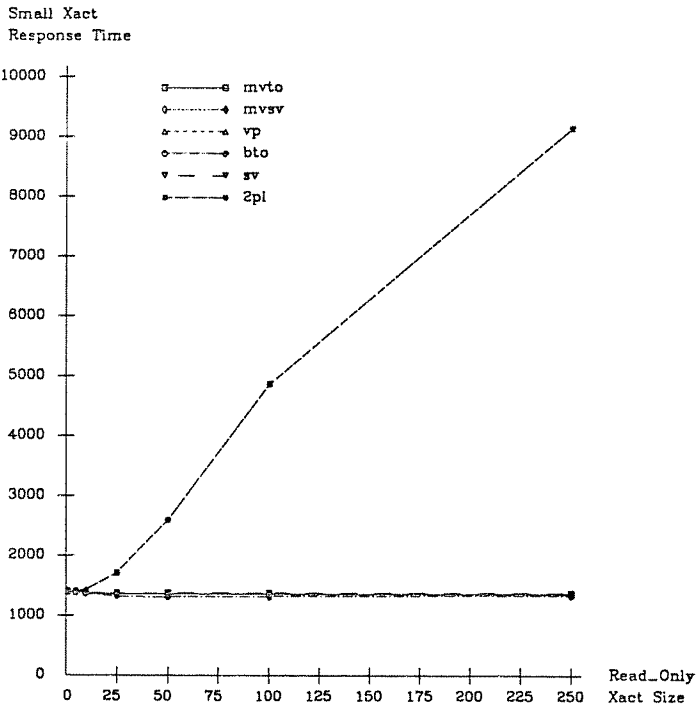

Figure 9: Small Xact Response Time.

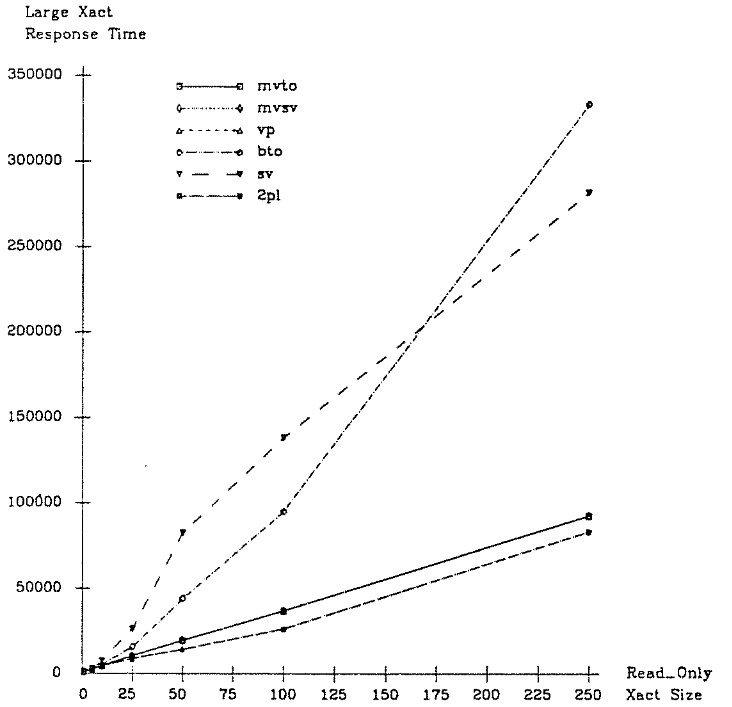

Figure 10: Large Xact Response Time.

Having studied the performance of the multiversion algorithms for this experiment, we now consider their storage overheads. Figure 11 shows how the relative version pool size, defined as the average ratio of the size of the version pool to the size of the database, behaves as a function of read-only transaction size. All three algorithms have the same overhead, and this overhead increases with read-only transaction size. The larger the ratio of read-only transaction response time to update transaction response time, the larger the number of old versions of objects that a read-only transaction may have to read (because more updates occur during its lifetime). Thus, more old versions have to be maintained in the version pool, which explains why the relative size of the version pool increases with read-only transaction size. However, as shown in the figure, the average version pool size is never more than about 11% of the size of the database itself in this experiment, even though the average read-only transaction reads as many as one-half of the objects in the database in some cases.

Figure 12 shows how the average of the maximum number of disk accesses per read request by transactions behaves as a function of read-only transaction size. The statistics in the figure were compiled by recording the maximum number of disk accesses among all read requests for each committed transaction, then averaging these results over all of the committed transactions. As shown in the figure, all three multiversion algorithms behaved similarly. The maximum number of disk accesses per read for transactions increases, expectedly, with the size of read-only transactions in the mix. Small update transactions almost always access the most recent versions of objects,[†] so they have an average number of disk accesses per read of 1. Figure 13 shows how the maximum number of disk accesses per read for large read-only transactions behaves — these transactions account for the increase in the curves of Figure 12. It is quite interesting to note that, despite the large fraction of the database accessed by read-only transactions, the maximum number of disk accesses per read never exceeds an average value of about 2.1 for the read-only transactions in this experiment.

Tables 5 and 6 each contain four results pertaining to the version access behavior of transactions for the VP algorithm for read-only transaction sizes of 50 and 250 (respectively). The results

[†] Since update transactions update each of two objects with 50% probability, an update transaction actually has a 25% chance of not updating any object (i.e., of being a small read-only transaction).

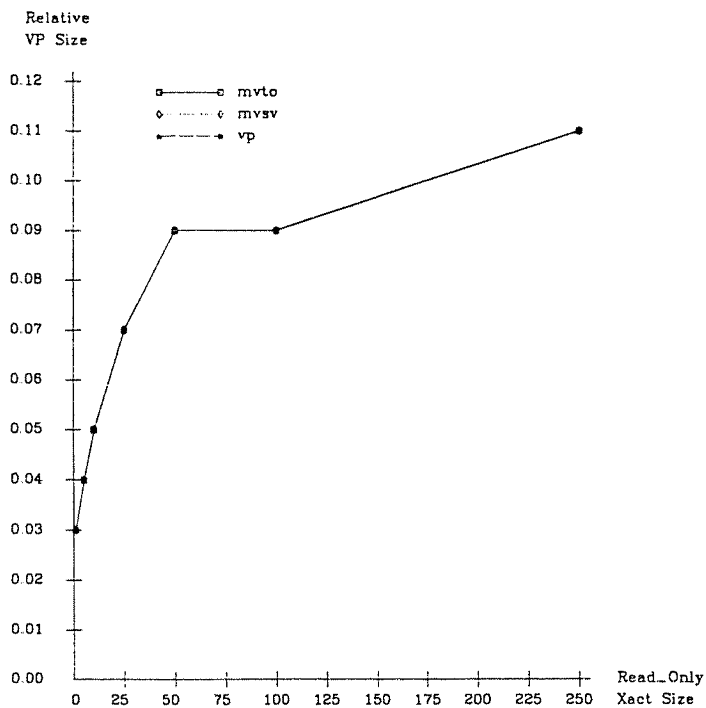

Figure 11: Relative Version Pool Size.

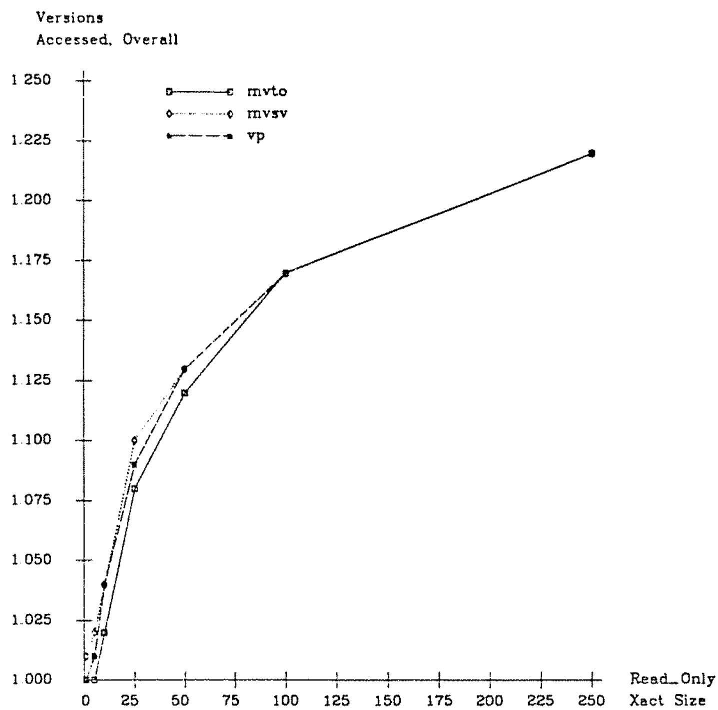

Figure 12: Versions Accessed, Overall.

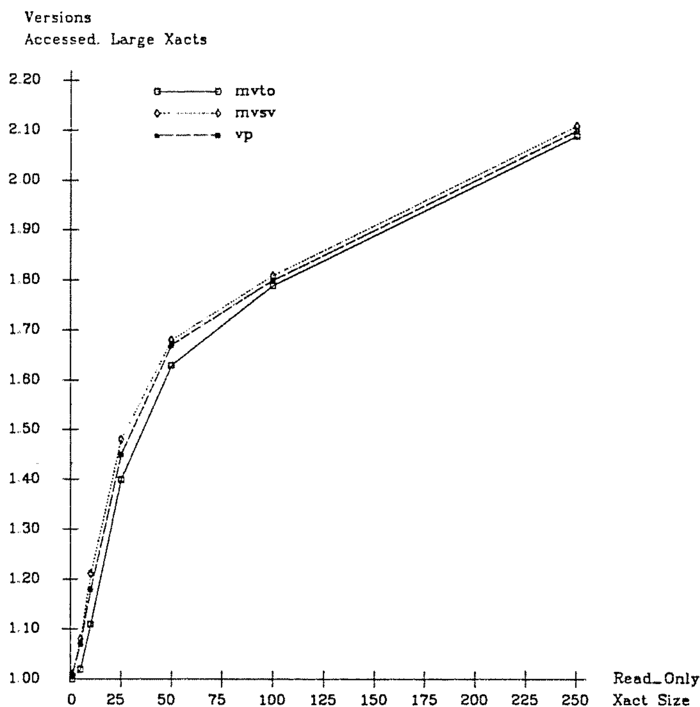

Figure 13: Versions Accessed, Large Xacts.

for the MVTO and MVSV algorithms were very similar, so VP results alone are given to illustrate the interesting points. The first column in each table is the number of disk accesses, or versions accessed, per read. The next three columns in each table pertain to the maximum number of disk accesses that a transaction had to perform in order to read an object, with the first column of data giving overall results and the next two columns giving the results by transaction class. The last column in each table shows the number of disk accesses required to satisfy read requests (recorded on a per-read basis instead of on a per-transaction maximum basis).

Table 5 shows that 88% of all transactions were able to avoid extra disk accesses, and only 1% of all transactions ever required more than one or two disk accesses. The results by transaction class are also quite enlightening. Small update transactions rarely needed more than one disk access. For large read-only transactions, 38% never needed more than one disk access, another 58% required only two accesses in the worst case, and another 5% required a maximum of three disk

Versions Accessed, Large Read-only Transaction Size = 50				
Disk Accesses	Percent of All Transactions	Percent of Small Transactions	Percent of Large Transactions	Percent of All Reads
1	87.66	99.91	37.69	96.88
2	11.43	0.09	57.71	3.04
3	0.88	0.0	4.48	0.08
4	0.03	0.0	0.12	0.002†

Table 5: Versions Accessed, Size = 50, Experiment 1.

Versions Accessed, Large Read-only Transaction Size = 250				
Disk Accesses	Percent of All Transactions	Percent of Small Transactions	Percent of Large Transactions	Percent of All Reads
1	83.68	100.00	17.81	95.92
2	11.00	0.00	55.39	3.91
3	5.03	0.00	25.33	0.16
4	0.29	0.00	1.47	0.01

Table 6: Versions Accessed, Size = 250, Experiment 1.

† This value is here to indicate that one transaction required a maximum of four disk accesses — the value should really be 0.00 for the percentages to add up properly.

accesses. Less than 1% of all large read-only transactions ever required more than three accesses, and no transaction ever had to perform five disk accesses to satisfy a read request. The last column in the table shows the number of disk accesses required to satisfy read requests, and it shows that about 97% of all read requests were satisfied using only one disk access.

Table 6 shows the same collection of statistics for the case where the mean size of large read-only transactions is 250, or one-half the overall database size. The results are quite similar to those in Table 5, though the percentages for more than one disk access are slightly higher in this case. Still, very few transactions ever needed four disk accesses in order to satisfy a read request, and never did any transaction require more than this number. Viewed on a per-read basis, 96% of all read requests were still satisfied in just one disk access. In other words, even in extreme cases, the vast majority of reads did not need to use versions other than the most recently committed version. Note, however, that the performance results discussed earlier do indicate that having old versions around for those accesses that need them helps performance immensely.

Tables 7 through 9 give throughput results for the three single version algorithms and for their multiversion counterparts using two different cost models. For each multiversion algorithm, results are given for a model where no extra cost is incurred when accessing an old version and for a model where the extra cost for accessing old versions is included (the model used throughout the rest of the paper). Model 1 shows the performance gains due strictly to the additional concurrency offered by the multiversion algorithms, and model 2 shows how the cost of reading old versions affects this performance. (Also given in the tables are the sizes of the 90% confidence intervals for these results, given as a percentage of the throughput values.)

Examining the tables, it is apparent that both cost models for the multiversion algorithms provide very similar throughput results, except when the mean size for read-only transactions is very large. In this case, the average cost for executing a transaction begins to be affected by the addition of the cost for accessing old versions. Compared to the differences in performance between the single version variant of each algorithm and the multiversion models, though, the differences between

Throughputs and Confidence Intervals			
Size	BTO	MVTO1	MVTO2
1	7.310±1.98%	7.310±1.98%	7.310±1.98%
5	6.004±1.10%	6.066±1.05%	6.062±1.08%
10	4.622±2.05%	4.917±1.67%	4.903±1.61%
25	2.419±7.86%	3.215±3.88%	3.172±3.81%
50	1.030±9.74%	2.088±5.48%	2.042±5.23%
100	0.452±10.72%	1.162±5.84%	1.124±6.66%
250	0.147±13.31%	0.538±6.80%	0.514±7.75%

Table 7: BTO and MVTO Throughput, Experiment 1.

Throughputs and Confidence Intervals			
Size	SV	MVSV1	MVSV2
1	7.250±2.63%	7.330±2.28%	7.330±2.00%
5	5.704±1.70%	6.062±1.14%	6.034±1.24%
10	3.810±2.68%	4.915±1.70%	4.879±1.36%
25	1.570±9.28%	3.214±3.52%	3.154±4.29%
50	0.539±7.57%	2.088±5.45%	2.039±5.05%
100	0.218±9.04%	1.162±5.82%	1.124±6.40%
250	0.062±19.47%	0.539±6.74%	0.513±7.12%

Table 8: SV and MVSV Throughput, Experiment 1.

Throughputs and Confidence Intervals			
Size	2PL	VP1	VP2
1	7.340±1.64%	7.340±2.32%	7.340±1.81%
5	6.076±1.26%	6.074±1.11%	6.060±1.08%
10	4.929±1.65%	4.927±1.65%	4.901±1.51%
25	3.212±3.76%	3.221±3.95%	3.165±3.99%
50	2.048±5.05%	2.088±5.49%	2.039±5.03%
100	1.070±5.70%	1.162±5.80%	1.124±6.40%
250	0.421±5.22%	0.539±6.79%	0.514±7.28%

Table 9: 2PL and VP Throughput, Experiment 1.

the multiversion models themselves are quite small. (In fact, the multiversion model differences lie within the range for which statistical variations cannot be ruled out as their cause.) These tables illustrate that the benefits of having multiple versions, at least in a performance sense, significantly outweigh the costs. The cost of accessing old versions simply does not significantly degrade overall performance.

4.2. Experiment 2: Transaction Mix

This experiment examines the behavior of the algorithms under a mix of transactions similar to that of Experiment 1. In this experiment, however, read-only transaction size is held fixed. The variable here is the fraction of update versus read-only transactions in the mix. The point of this experiment is to find out the type of mixes for which multiple versions provide the greatest performance benefits. Also investigated is the way in which the storage cost (i.e., the size of the version pool) varies with the transaction mix.

Table 10 gives the parameter settings used in this experiment. The update transactions in the mix again read two objects, updating each with 50% probability. The mean size of read-only transactions is 50 in this case, meaning that each read-only transaction reads between 1 and 100 sequential pages. As mentioned above, the mix of transactions in the workload is varied in this experiment. In particular, workloads with 0%, 20%, 40%, 60%, 80%, and 100% update transactions are studied, with the remainder of the workload consisting of read-only transactions.

Figures 14 through 16 show the overall throughput, restart ratio, and average response time results for Experiment 2. Since small transactions execute much more quickly than large transactions, the throughput increases in Figure 14 with the fraction of small transactions in the mix. The interesting thing about the throughput results in this experiment is that the difference between the single version and multiversion concurrency control algorithms is the greatest when the mix consists

Workload Parameters	
small-prob	0.0, 0.2, 0.4, 0.6, 0.8, 1.0
small-mean	2
small-xact-type	random
small-size-type	fixed
small-write-prob	0.5
large-mean	50
large-xact-type	sequential
large-size-type	uniform
large-write-prob	0.0

Table 10: Workload Parameters, Experiment 2.

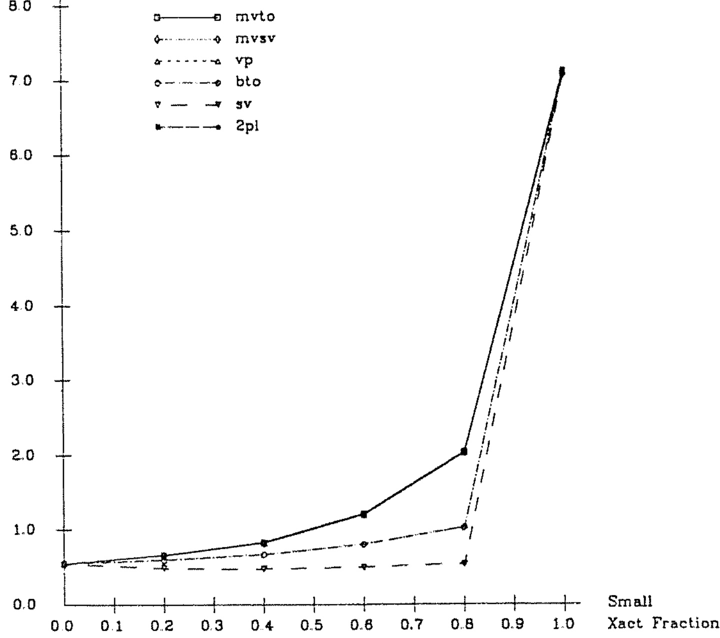

Figure 14: Overall Throughput.

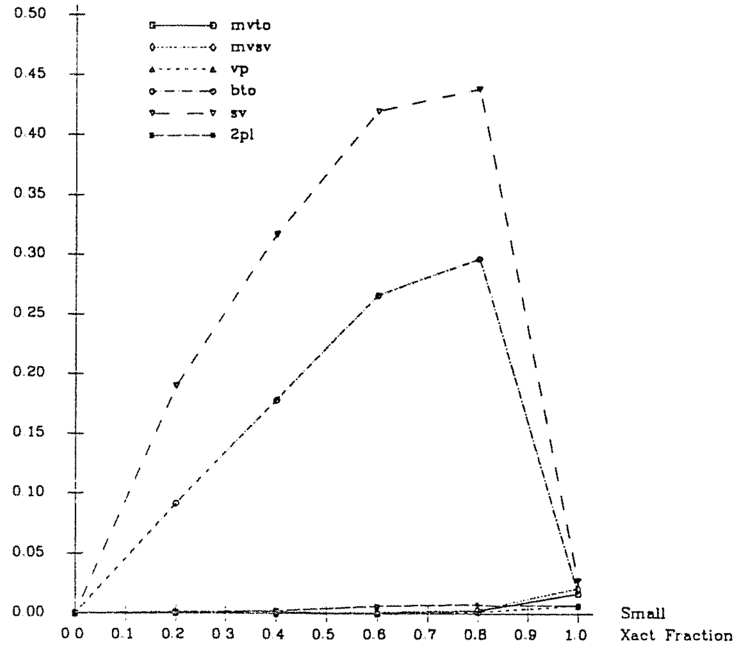

Figure 15: Restart Ratio.

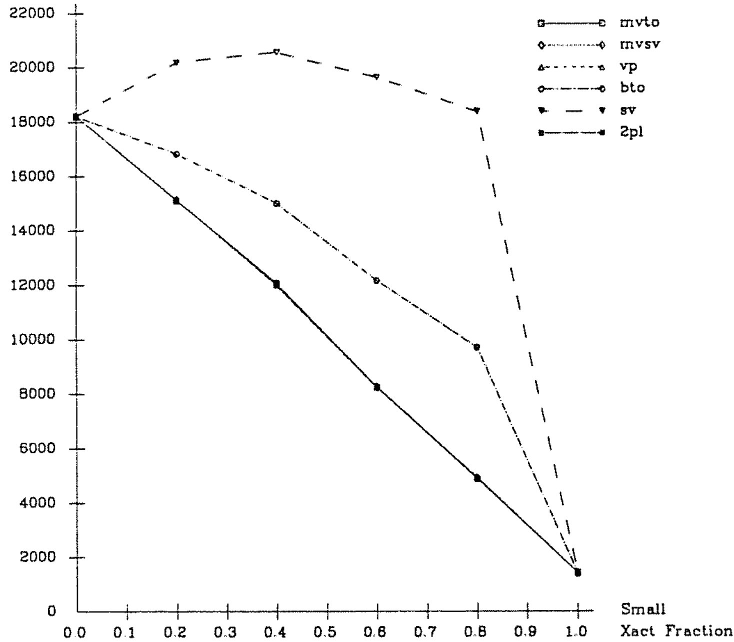

Figure 16: Overall Response Time.

of 80% small update transactions and 20% large read-only transactions. This is easy to explain — the more update transactions there are in the mix with read-only transactions, the more likely the read-only transactions are to be interfered with by updates in the single version case. This is clear in Figure 15, which shows that the restart ratios for BTO and SV are the largest with this mix. Figure 16 shows that the differences in response times for BTO and SV versus MVTO and MVSV (respectively) are the greatest for this mix as well. 2PL and VP perform approximately the same in all the figures. This is to be expected, as differences between 2PL and VP did not come out in Experiment 1 until the size of read-only transactions was larger than 50.

Figures 17 through 19 show the storage results obtained in this experiment. The results are basically what one would expect, given the storage results of Experiment 1 and the performance results of Figures 14 through 16. The relative size of the version pool is greatest with 80% updaters and 20% large readers, which was where the multiple version algorithms outperformed their single version counterparts by the most significant amounts. The maximum number of disk accesses needed to satisfy read requests by large transactions, on the average, is also the greatest for this mix of transactions. Figure 18 shows the overall average, and Figure 19 shows how the results for large read-only transactions behave. The overall average actually peaks at the mix of 60% updaters and 40% large readers — this is because small update transactions have their requests satisfied in a single disk access, and the 80% of the mix that are update transactions pull the overall average value down towards 1.0.

4.3. Experiment 3: Update Conflicts

This experiment examines the behavior of the algorithms under a workload consisting purely of update transactions. Whereas experiments 1 and 2 examined how each of the multiple version algorithms performs under conditions which were favorable for the algorithms, in this experiment we examine the performance of the algorithms under less favorable conditions. The size of update transactions is varied in order to see how each of the algorithms performs under varying conflict probabilities.

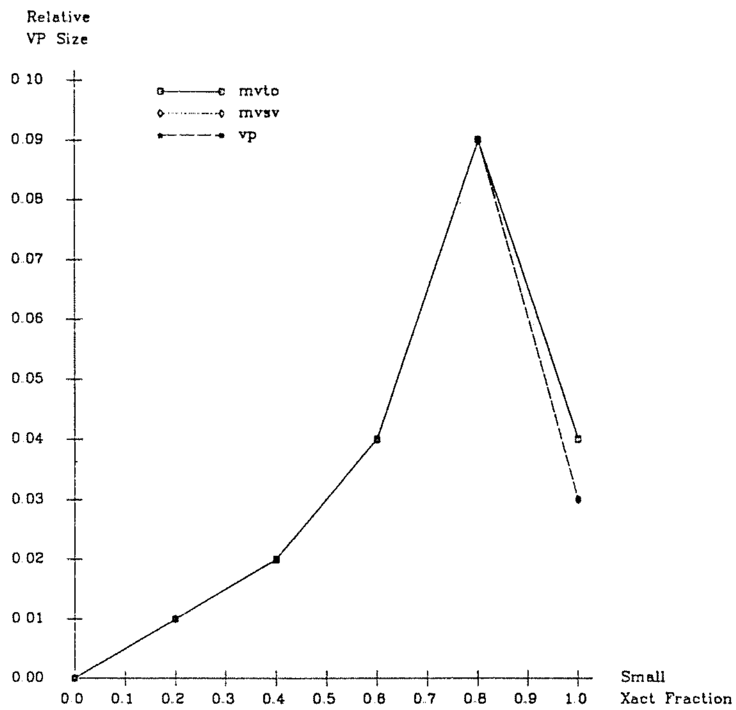

Figure 17: Relative Version Pool Size.

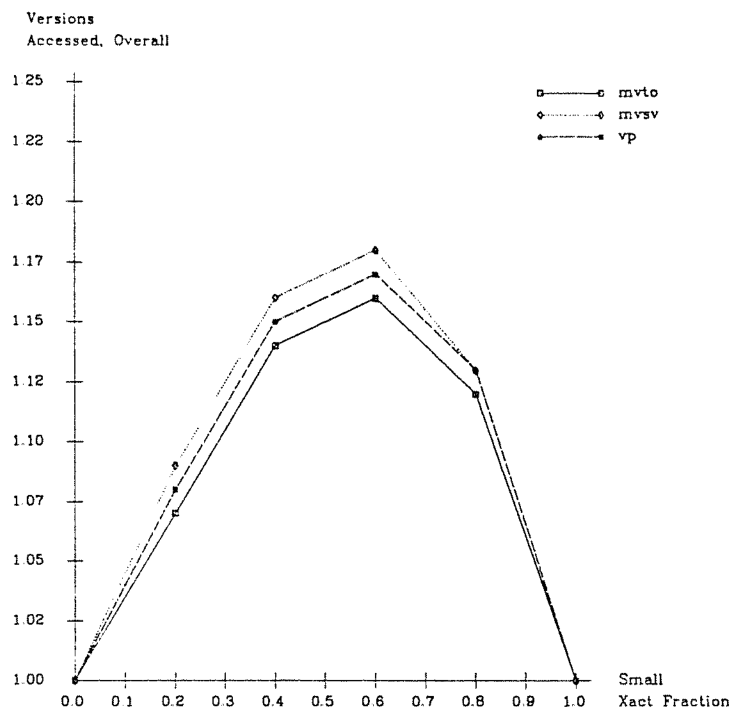

Figure 18: Versions Accessed, Overall.

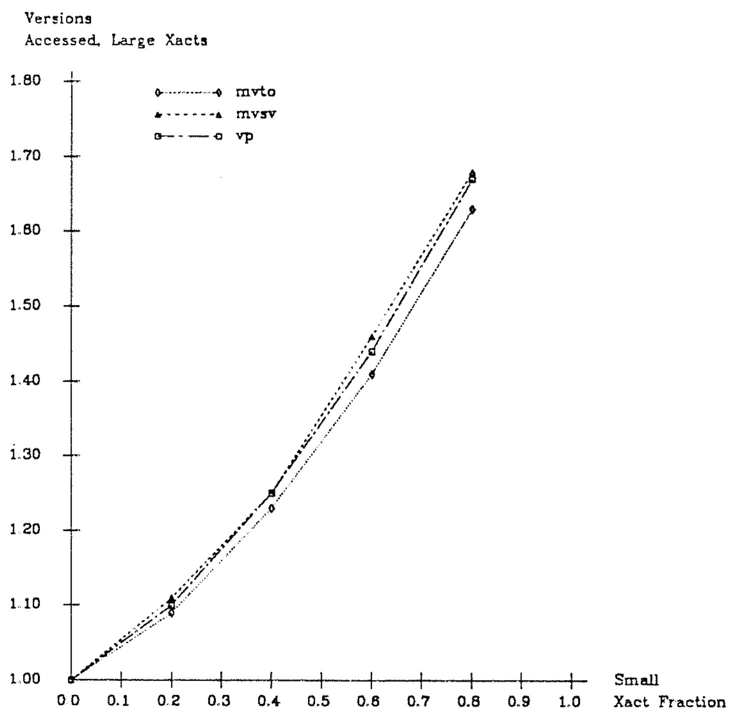

Figure 19: Versions Accessed, Large Xacts.

The workload parameters for Experiment 3 are given in Table 11. All of the transactions randomly read and then update some number of pages in the database. The number of pages accessed is varied from 1 to 100, as indicated in the table. At the extreme, then, conflicts between update transactions are highly probable — each update transaction reads and then updates 1/5 of the database, and the multiprogramming level for the experiment is ten (as always).

Figures 20 through 23 give the overall throughput, restart ratio, response time, and relative version pool size results for Experiment 3. The throughput results for update transactions of size 1 were omitted so as to make the ranges covered by the graph more helpful: all algorithms performed alike at this size anyway. In this experiment, where the probability of conflicts between transactions is non-negligible even with multiple versions, significant differences in performance are visible among the three multiversion concurrency control algorithms. Looking at Figure 20, VP provides the best throughput with update transactions of size 10 or less, and MVSV provides the best throughput for larger update transaction sizes. MVTO performs significantly worse than the other two algorithms for all but the smallest of transaction sizes. Figure 21 indicates that the response time results follow the same basic trend, and Figure 22 shows that the explanation behind the trend lies with the restart ratio — the algorithm causing the least number of restarts offers the best overall performance.

The reason that VP outperforms MVSV initially is that it blocks conflicting transactions, restarting them only when deadlocks require it to do so. MVSV, on the other hand, uses end-of-transaction restarts to resolve conflicts. The reason that the MVSV and VP curves cross over, with

Workload Parameters	
small-mean	1, 5, 10, 25, 50, 100
small-xact-type	random
small-size-type	fixed
small-write-prob	1.0

Table 11: Workload Parameters, Experiment 3.

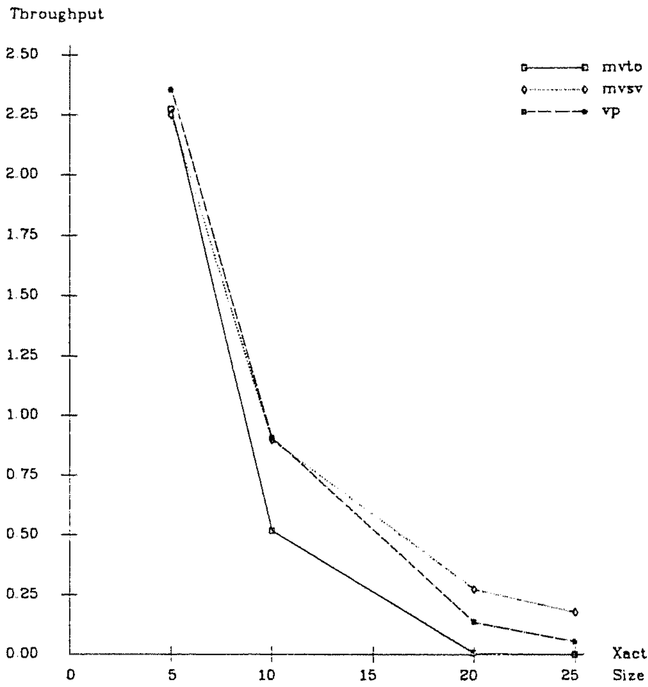

Figure 20: Overall Throughput.

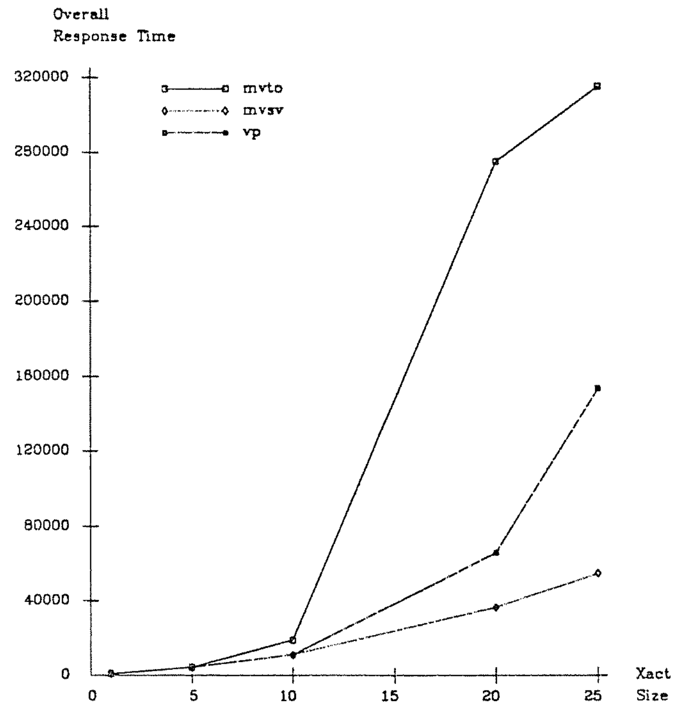

Figure 21: Overall Response Time.

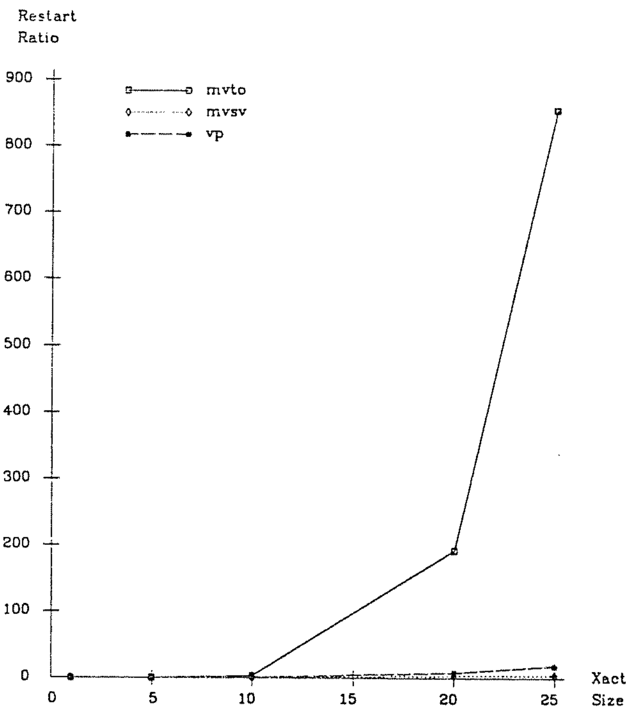

Figure 22: Restart Ratio.

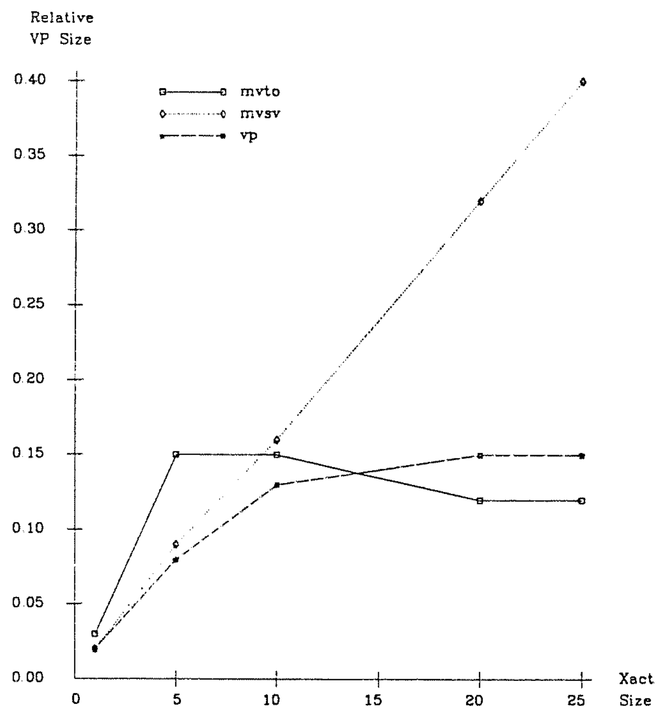

Figure 23: Relative Version Pool Size.

MVSV eventually outperforming VP, is that the deadlock victim selection criteria which we chose to use is poor and leads to thrashing. The policy used in our VP implementation is to check for deadlock every time a transaction blocks, and to restart the current blocker (the transaction whose blocking induces the deadlock). It is possible for transactions to repeatedly restart each other under this policy [Care83a, Pein83]; this is evident from the fact that the restart ratio for VP significantly exceeds 1 for VP (it reaches about 18 for the largest transaction size). With SV on the other hand, a transaction is only restarted when it conflicts with another, *committed*, transaction. SV thus has a sort of stability, in the sense that it guarantees that some useful work will get done in the system. We expect that a more stable victim selection criteria, such as restarting the youngest transaction in the deadlock cycle, would allow VP to uniformly outperform (or at least perform equally as well as) MVSV.

The reason for the poor showing by MVTO in this experiment is a phenomenon known as cyclic restarts [Date82, Care83a, Ull83]. This phenomenon has been known to be a problem for BTO, and it can also arise between updaters in the MVTO algorithm. Basically, two updaters that wish to access the same object can get into a mutually infinite loop, with one restarting the other, then the other coming back and restarting the first one, etc. Such loops can also involve more than two transactions, and more than one such loop can exist at a time. The serious performance impact

Step	*Action*	*Result*
1	T1: begin;	S-TS(T1) = 1
2	T2: begin;	S-TS(T2) = 2
3	T1: read *X*;	R-TS(*X*) = 1
4	T2: read *X*;	R-TS(*X*) = 2
5	T1: write *X*;	Restart(T1) with S-TS(T1) = 3
6	T1: read *X*;	R-TS(*X*) = 3
7	T2: write *X*;	Restart(T2) with S-TS(T2) = 4
8	T2: read *X*;	R-TS(*X*) = 4
9	T1: write *X*;	Restart(T1) with S-TS(T1) = 5
10	T1: read *X*;	R-TS(*X*) = 5
11	etc.	etc.

Figure 24: Example of cyclic restart anomaly.

of such restart loops is evident from the *extremely* high restart ratios for MTVO (i.e., values >> 1). Figure 24 illustrates the phenomenon for a pair of update transactions which each read and then update the most recent version of an object X. Each restart in the figure is due to an attempt to write when the startup timestamp of the writer is less than the read timestamp of the most recent version of X. The solution to the problem is to delay resubmitting restarted transactions for awhile, until the conflict that caused the restart will be gone with high probability. Our one-second restart delay was not sufficient to solve this problem for the larger update transactions. In fact, no fixed length delay can solve the problem if transaction sizes can be arbitrarily large and are not known a priori. An adaptive delay solution, perhaps analogous to the probabilistic collision control algorithm used in the Ethernet [Metc76], seems necessary. With such an adaptive delay, we expect that BTO would yield performance closer to that of VP and MVSV, although its restart delays would still cause some degradation in performance with respect to these other algorithms.

Returning to the graphs, Figure 23 shows how the relative version pool size behaves as a function of the size of the transactions in this experiment. Note that, with no read-only class of transactions in the mix, the entire version pool size is due to the before-images of objects which are currently (or were recently) undergoing updates. For VP and MVTO, the size of the version pool gets up to about 15% of the database size, then seems to flatten out. For MVSV, however, the version pool size grows steadily to 40% of the database size as the size of the transactions in the workload increases. This occurs because VP and MVTO use blocking, to some extent, to resolve conflicting updates; MVSV relies completely on end-of-transaction restarts. In MVSV, many conflicting transactions can concurrently attempt to update objects in the database, and the version pool becomes increasingly cluttered with the before-images of objects that restarted transactions were in the process of updating. (Recall that the version pool is maintained as a sliding range of objects, so these useless objects will not be reclaimed until the *update-first* and *reader-first* pointers advance beyond their position in the version pool.) The problem of increasing version pool size in MVSV could be solved by delaying the copying of before-images from the main segment into the version pool until commit time, after the updating transaction has been successfully validated. Doing this,

however, is not without cost — the updater will have to re-read the before-image of the object in the main segment in order to perform the transfer at commit time.

5. CONCLUSIONS

In this paper, we have examined the performance and storage overheads of three multiversion concurrency control algorithms, Reed's multiversion timestamp ordering algorithm, the CCA version pool algorithm, and a multiversion variant of Kung and Robinson's serial validation algorithm. We have also compared the performance of the algorithms to their single version counterparts (basic timestamp ordering, two-phase locking, and serial validation, respectively). Our study of these algorithms was based on a detailed simulation model of a centralized (i.e., single-site) database management system.

Experiment 1 examined the performance of the algorithms under a mix of small update transactions and large read-only transactions of various sizes. It was found that all three multiversion algorithms outperform their single version counterparts under such a workload. Since the probability of conflicts was virtually zero for the multiversion algorithms with this workload, all three algorithms performed identically. It was seen that MVSV significantly outperformed SV and that MVTO significantly outperformed BTO when the size of read-only transactions was fairly large, but that VP only moderately improved on the performance of 2PL using throughput as the performance metric of interest. In examining the response time results, however, it was found that VP provides a significant savings in the average response time for transactions, doing so by trading a slight increase in the average response time for large transactions for a huge decrease in the average response time for small transactions.

In terms of their storage characteristics, all three multiversion algorithms were again very similar in Experiment 1. The relative size of the version pool was seen to increase with the size of read-only transactions, but its average size never exceeded about 11% of the total database size — even with very large read-only transactions. It was observed that most read requests (95% or more) could be satisfied in a single disk access. Over 83% of all transactions were able to execute without ever

needing more than one disk access for any of their reads, and never were more than four disk accesses required. Finally, it was seen that the cost of these accesses to old versions only slightly degraded the throughput of the system, with the benefits of the multiversion algorithms (the reduction in blocking and/or restarts) outweighing the costs.

Two other experiments were also performed. In Experiment 2, the fraction of update transactions in the mix was varied. It was found that mixes with a large fraction of updaters are the ones that benefit the most from having multiple versions, as these are the mixes with the greatest probability of conflicts between updaters and large readers for the single version algorithms. In Experiment 3, a workload consisting solely of update transactions was examined, and the size of these transactions was varied in order to see how the three multiversion algorithms would behave when conflicts were more likely. It was found that VP and MVSV outperformed MVTO, with VP doing the best with smaller update transactions and MVSV doing the best for the larger update transaction sizes (where VP's simple deadlock victim selection criteria was insufficient to achieve reasonable performance).

In summary, multiversion concurrency control algorithms *can* definitely provide improvements in performance by allowing large read-only transactions to access previous versions of data items. The added costs that arise due to following version chains for read requests are not significant, as the majority of read requests can be satisfied in a single disk access, and most of the remaining requests require just one additional access. Finally, the storage overhead for maintaining all old versions which might be required to satisfy read requests from ongoing transactions is surprisingly small — in our experiments, the average size of the version pool rarely exceeded 10-15% of the overall size of the current version of the database.

REFERENCES

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

[Bern81] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys* 13(2), June 1981.

[Bern83] Bernstein, P., and Goodman, N., "Multiversion Concurrency Control — Theory and Algorithms", *ACM Transactions on Database Systems* 8(4), December 1983.

[Brya80] Bryant, R., *SIMPAS -- A Simulation Language Based on PASCAL*, Technical Report No. 390, Computer Sciences Department, University of Wisconsin-Madison, June 1980.

[Care83a] Carey, M., *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. Thesis, Computer Science Division (EECS), University of California, Berkeley, August 1983.

[Care83b] Carey, M., *Multiple Versions and the Performance of Optimistic Concurrency Control*, Technical Report No. 517, Computer Sciences Department, University of Wisconsin, Madison, October 1983.

[Care84] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems", *Proceedings of the Tenth International Conference on Very Large Data Bases*, Singapore, 1984.

[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*, 1982.

[Date82] Date, C., *An Introduction to Database Systems (Volume II)*, Addison-Wesley Publishing Company, 1982.

[Gray79] Gray, J., "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, Springer-Verlag, 1979.

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

[Lin83a] Lin, W., and Nolte, J., "Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking", *Proceedings of the Ninth International Conference on Very Large Data Bases*, Florence, Italy, 1983.

[Lin83b] Lin, W., and Nolte, J., "Performance of Distributed Concurrency Control", in *Distributed Database Control and Allocation*, Final Technical Report, Volume II, Computer Corporation of America, Cambridge, MA, 1983.

[Metc76] Metcalfe, R. and Boggs, D., "Ethernet: Distributed Packet Switching For Local Computer Networks", *Communications of the ACM* 19(7), July 1976.

[Papa84] Papadimitriou, C., and Kannelakis, P., "On Concurrency Control by Multiple Versions", *ACM Transactions on Database Systems* 9(1), March 1984.

[Pein83] Peinl, P., and Reuter, A., "Empirical Comparison of Database Concurrency Control Schemes", *Proceedings of the Ninth International Conference on Very Large Data Bases*, Florence, Italy, 1983.

[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.

[Reed83] Reed, D., "Implementing Atomic Actions on Decentralized Data", *ACM Transactions on Computer Systems* 1(1), February 1983.

[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.

[Robi82] Robinson, J., *Design of Concurrency Controls for Transaction Processing Systems*, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1982.

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

[Stea81] Stearns, R., and Rosenkrantz, D., "Distributed Database Concurrency Controls Using Before-Values", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1981.

[Ullm83] Ullman, J., *Principles of Database Systems*, Second Edition, Computer Science Press, Rockville, Maryland, 1983.

[Wolf83] Wolff, R., personal communication.