The Performance of Multiversion Concurrency Control Algorithms

MICHAEL J. CAREY and WALEED A. MUHANNA
University of Wisconsin

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 multiversion algorithms are compared with regard to performance with their single-version counterparts and also with each other. It is shown that each multiversion algorithm offers significant performance improvements despite the additional disk accesses involved in accessing old versions of data; the nature of the improvement depends on the algorithm in question. It is also shown that the storage overhead for maintaining old versions that may be required by ongoing transactions is not all that large under most circumstances. Finally, it is demonstrated that it is important for version maintenance to be implemented efficiently, as otherwise the cost of maintaining old versions could outweigh their concurrency benefits.

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*simulation*; H.2.2 [**Database Management**]: Physical Design—*deadlock avoidance*; *recovery and restart*; H.2.4 [**Database Management**]: Systems—*transaction processing*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Concurrency control, multiple versions

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 [2, 7, 9, 11, 23, 24, 28, 31]. 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 [2, 31], whereas other algorithms are designed to utilize potentially many versions of an object [7, 9, 11, 23, 24, 28]. For most of the algorithms in the latter class, the idea is to permit long

This research was supported in part by an IBM Faculty Development Award, by National Science Foundation grant DCR-8402818, and by the Wisconsin Alumni Research Foundation.
Authors' addresses: M. J. Carey, Department of Computer Sciences, University of Wisconsin, Madison, WI 53706; W. A. Muhanna, Graduate School of Business, University of Wisconsin, Madison, WI 53706.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1986 ACM 0734-2071/86/1100-0338 $00.75

read-only transactions to read older versions of objects while allowing update transactions to create newer versions concurrently. It is this latter class of multiversion algorithms, 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 has been the subject of several recent theoretical papers [4, 21]. 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 three 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 costs (particularly the storage cost) for maintaining multiple versions?

The answers to these questions are investigated for three multiversion algorithms: Reed's multiversion timestamp ordering algorithm [24], which is based on timestamps; the version pool algorithm used by the Computer Corporation of America (CCA) in their LDM database system [9, 11], which is an algorithm based on two-phase locking;[1] and a multiversion serial validation algorithm [7], which is based on the optimistic concurrency control algorithm of Kung and Robinson [16]. The algorithms are compared with regard to performance both with each other and with their single-version counterparts. We do not attempt to predict the absolute performance of the various algorithms, as this will of course depend on specific implementation and workload details. Instead, our goal is to examine the relative performance of the algorithms and to analyze the performance trade-offs involved.

Few previous studies have addressed these questions. Several recent papers by Lin and Nolte included throughput results for multiversion timestamp ordering [18, 19], 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 [19]. In both papers, however, transaction sizes were fairly 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 studies were 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 [22] included results for the before-value locking scheme of [2], but these results were based on synthetic performance metrics (the number of restarts and the average number of nonblocked transactions, not throughput or response

[1] The CCA algorithm is based on a similar algorithm used in Prime's Database Management System [13], as noted in [9] and [11].

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 some of 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, work wasted due to restarts, 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.

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 [23, 24]. In particular, we consider the algorithm as implemented in the SWALLOW data repository project at MIT [24]. This version of the algorithm can be viewed as a multiversion variant of the basic timestamp ordering algorithm (BTO) of Bernstein and Goodman [3] (although Reed's algorithm came first historically): 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 $S\text{-TS}(T) \geq W\text{-TS}(X)$. 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 made for an object with a pending write request from an older transaction, 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 that 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.

Since MVTO is a multiversion variant of basic timestamp ordering, it shares BTO's potential for having the *cyclic restart problem* [6, 12, 32] arise among two or more update transactions. The basic problem is that a pair of update transactions wishing to concurrently read and then write a common object can restart each other over and over again, and this "restart loop" can persist indefinitely. (Note that a transaction receives a new startup timestamp each time it restarts, so the timestamp ordering of such a pair reverses every time one of them restarts.) We addressed this problem in our implementation by dynamically maintaining an estimate of the mean response time for transactions; we used this estimate to delay restarted transactions for an exponentially distributed time period with a mean of one response time. We chose this scheme because it has worked well for other algorithms [1]; Ullman proposed another random delay-based scheme that would also work [32]. Adding this adaptive delay drastically improved performance for both BTO and MVTO as compared with some earlier results (from [6]) that were obtained with a constant restart delay. (For more information on the cyclic restart problem, the reader is referred to one of [6], [12], or [32].)

2.2 The CCA Version Pool Algorithm (MV2PL)

The CCA version pool algorithm [10, 11] 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. Transactions block when they cannot obtain a lock, and deadlock must be dealt with in one of the usual ways. Our implementation checks for deadlocks whenever a transaction blocks, restarting the youngest transaction in a deadlock cycle when one is discovered. 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.[2]

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 that were written by transactions that committed before T even began running. Thus, T is serialized after all transactions that committed prior to its startup, but before all transactions that are active during any portion of its lifetime.

[2] The CCA algorithm actually stores the creator's transaction identifier with the item, maintaining a separate list that associates transactions with their commit timestamps [10, 11].

2.3 Multiversion Serial Validation (MVSV)

The multiversion serial validation algorithm [7] is based on the single-version optimistic concurrency control algorithm of Kung and Robinson [16] known as *serial validation* (SV). In the SV algorithm, transactions record their read and write sets as they run. A transaction is restarted at its commit point if any item in its readset has been written by a transaction that committed during its lifetime. One difference between their algorithm and our version of the algorithm, which we shall describe shortly, is that we use timestamps to efficiently check for readset/writeset conflicts instead of storing old writesets and explicitly testing for readset/writeset intersections [7]. 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 $\text{TS}(X)$ is maintained for each data item X; $\text{TS}(X)$ 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 $S\text{-TS}(T) > \text{TS}(X_r)$ for each object X_r in its readset. Each transaction T that successfully commits sets $\text{TS}(X_w)$ equal to $C\text{-TS}(T)$ for all data items X_w in its writeset.

The CCA version pool algorithm is a multiversion algorithm that enhances a known concurrency control algorithm, two-phase locking, by permitting read-only transactions to read older versions 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 that can be achieved using the algorithm. This same idea can be applied to yield a multiversion variant of serial validation.

In multiversion serial validation [7], 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. (Similar multiversion optimistic concurrency control algorithms are discussed in [17], [28], and [29].)

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 [11]. 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 *max-vp-size* −1. Versions of objects are chained

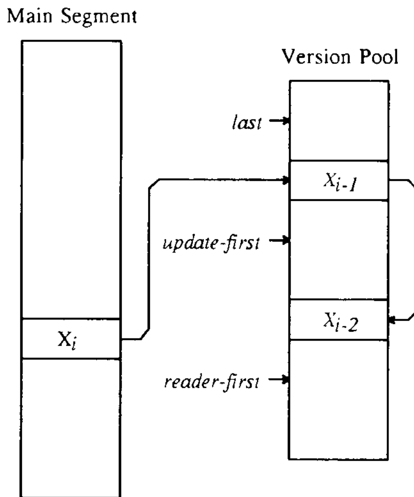

Fig. 1. Storing multiple versions.

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.

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 [11]. Three pointers, *reader-first*, *update-first*, and *last*, where *reader-first* $\leq$ *update-first* $\leq$ *last* (modulo *max-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 [14] for recovery purposes [11]. 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 on Chan's completed transaction lists [11], slightly simplifying the implementation while preserving the desired semantics.

Of the three multiversion algorithms studied in this paper, two were designed to be used with this version management scheme [7, 11]. Reed proposed a scheme where versions with timestamps newer than some threshold value are kept, and older versions are discarded [24]. 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 and 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, unlike Reed's proposal, it allows objects to be updated in place, which is usually thought to be preferable 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 making one uniform assumption would facilitate a fairer comparison of the algorithms, and using the CCA scheme allows 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 that 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 [1, 6, 8].

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 on the basis of its terminal of origin, and its class determines the manner in which its readset and writeset are assigned. Transaction classes, readsets, and writesets are generated using the workload parameters shown in Table I.

The parameter *mpl* determines the level of multiprogramming for the workload. 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 names 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, the class of the terminal is used to decide the class of the transaction. To provide a steady mix of transactions of each class, terminal classes are assigned when the simulation begins by statically designating $\lfloor small\text{-}frac * num\text{-}terms \rfloor$ terminals as generators of small transactions and the remainder as generators of large transactions. If the class of a newly initiated 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-size-dist*. It may be fixed or uniform. If it is fixed, the readset size is simply *small-mean*. If it is uniform, the readset size is chosen from a uniform distribution with range $small\text{-}mean \pm \lfloor small\text{-}mean/2 \rfloor$. 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

Table I. Workload Parameters for Simulation

	Workload parameters
mpl	Multiprogramming level
db-size	Size of database
small-frac	Fraction of small transactions in mix
small-mean	Mean size for small transactions
small-xact-type	Type for small transactions
small-size-dist	Size distribution for small transactions
small-write-prob	Pr(write X | read X) for small transactions
large-mean	Mean size for large transactions
large-xact-type	Type for large transactions
large-size-dist	Size distribution for large transactions
large-write-prob	Pr(write X | read X) for large transactions

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 that 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. 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 reenters the concurrency control queue and makes the request. It is assumed for modeling convenience that transactions first perform all of their read accesses and then perform any write accesses.

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 with a mean of one average transaction response time; this adaptive restart delay is invoked regardless of the concurrency control algorithm employed, as it was found in a previous study that the adaptive delay seems to have a stabilizing effect on any concurrency control algorithm when the conflict probability becomes large [1]. Following its restart delay, the transaction then begins making all of its concurrency control requests and object accesses over again. Eventually, the transaction may complete and the concurrency control algorithm may choose

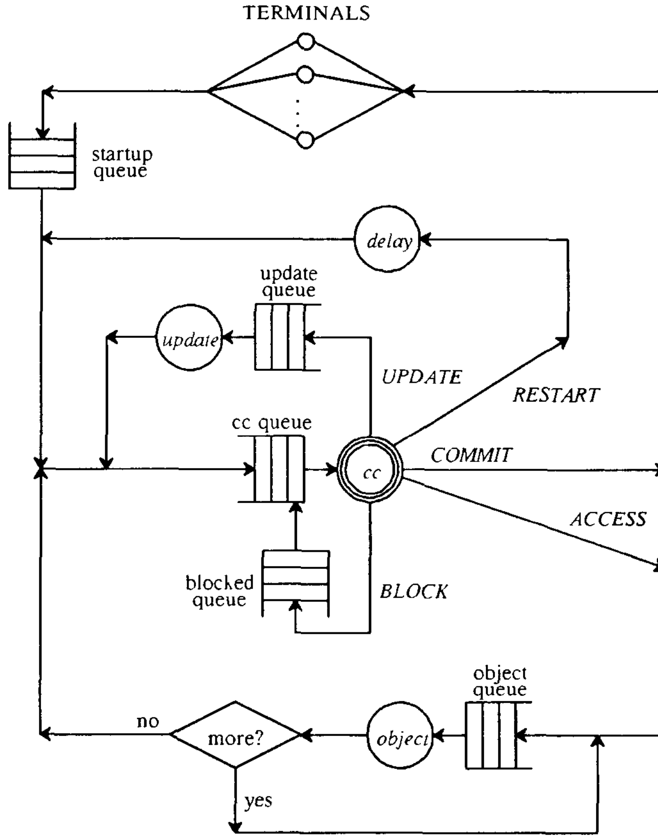

Fig. 2. Logical queuing model.

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 I/O (disk) and CPU 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 can involve I/O processing followed by CPU processing. The amounts of I/O and CPU used 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 (plus the restart delay path). Requests in the CPU queue are serviced FCFS (first-come, first-served), except that concurrency control requests have priority over all other service requests. The service discipline for the I/O queue is also FCFS. These service disciplines

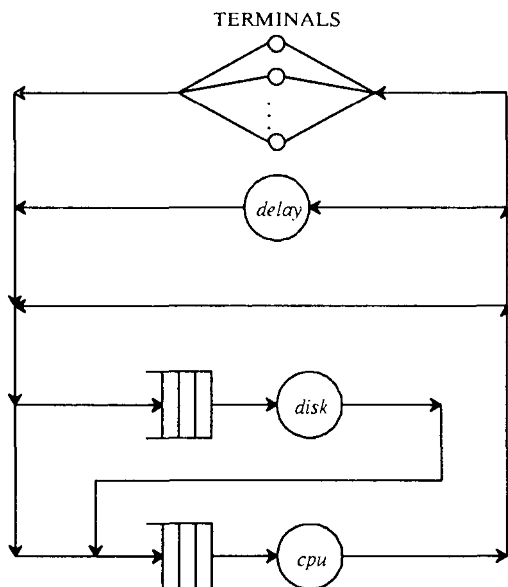

Fig. 3. Physical queuing model.

Table II. System Parameters for Simulation

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-cpu	Basic unit of concurrency control CPU time

were chosen to approximately model the characteristics that a real database system implementation would have. Note that, since a transaction never requests CPU time for processing more than one page in a cycle through the CPU queue, CPU scheduling is approximately equivalent to a round-robin CPU scheduling policy where the quantum exceeds the page processing time.

The parameters determining the service times (I/O and CPU) for the various logical resources in the model are given in Table II. 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 is assumed to require the same resources as reading an object,[3] 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 parameter *cc-cpu* is the amount of CPU

[3] We refer here to the cost of writing the object itself; recovery cost issues are discussed separately in Section 3.4.

time associated with a concurrency control request. All of 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 1 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 [5], 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 any CPU costs associated with concurrency control processing.

3.4 Concurrency Control and Recovery Costs

In order to simulate the single-version and multiversion concurrency control algorithms of interest, it is necessary to make some assumptions about their costs. In this section we briefly describe how the simulation cost parameters are used to model the costs for each of the multiversion algorithms.

It is assumed that transactions are issued startup timestamps (for all three algorithms) and registered as being either read-only or update transactions (for MV2PL 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 MV2PL 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 a charge of *cc-cpu* per action [8].

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

of 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.

Note that we do not distinguish between random and sequential I/O costs in our model of the object accessing cost. We chose not to do so for several reasons. First, while sequential transactions incur lower I/O costs than random transactions in a single-user environment, this is much less true in a multiuser environment where the disk receives interleaved requests for accesses to different cylinders on the disk. Second, there is recent evidence that, in a multiuser, multidisk environment, it may actually be better to spread sequential data pages out across the disks (i.e., to "decluster" the data) in order to improve performance [20]. Finally, distinguishing between random and sequential I/O would require us to adopt a much more detailed model of the disk, its associated scheduler, and the physical layout of data on disk, and we doubt that this added complexity would significantly affect the overall conclusions of the paper.

In addition to the costs for concurrency control processing and following version chains, the multiversion algorithms incur version maintenance costs. 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. We ignore this source of costs throughout most of the paper, as we believe that this cost is analogous to the recovery costs (e.g., logging costs) that arise in single-version algorithms, and thus that recovery costs are comparable for all of the algorithms considered here. In terms of UNDO and REDO logging [14], copying before-images to the version pool eliminates the need for UNDO logging for transaction recovery; however, REDO logging is still necessary to protect against media failure (and also against normal crashes if updated pages are not forced to disk at commit time) [8, 11]. Thus, to see that it is indeed reasonable to ignore the before-image copying cost here, we must compare it with the cost of UNDO logging in single-version algorithms.

For UNDO logging, the before-image of each changed object is written to the log, which requires a sequential page write (assuming page-level logging). The cost of before-image copying can be reduced to a similar level by keeping the version pool on dedicated stable storage devices like log disks [10]; version pool disks can be managed in nearly the same way as log disks are managed, as a circular buffer [15]. Although it is true that a version pool read will move the version pool disk arm, thus lessening the sequentiality of version pool writes, similar arm movement can occur with log disks; for example, a background log archiving process will compete with log writes, as will the log reads needed to undo the effects of restarted transactions. (We see later that multiversion algorithms can reduce the frequency of transaction restarts, making the latter effect more pronounced for a log disk than for a version pool disk.) Similarly, the amount of information recorded in the log is not significantly different from

that recorded in the version pool—although we assume page-level operations in this study, both the log and the version pool mechanism are capable of operating at the record level as well [10, 15]. The bottom line is that similar recovery costs appear to be incurred by both single-version and multiversion concurrency control algorithms, and a truly realistic comparison of these costs would require a very detailed model of the recovery implementations used by the respective algorithms. We thus factor out these recovery costs (i.e., log writes and version pool writes) throughout most of the paper. An alternative version maintenance cost assumption is examined in Section 4.4, and we will see that it is indeed important that the version pool write cost be made equivalent to the cost of a log write in order to retain all of the benefits associated with multiversion algorithms.

3.5 Statistical Analysis

In the simulation experiments reported here, one of the primary performance metrics used is the transaction throughput rate. Mean throughput results and 90 percent confidence intervals for these results were obtained from the simulations using a variant of the *batch means* [30] approach to simulation output analysis. The approach used is due to Wolff (personal communication, 1983); 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 nonadjacent 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 the confidence interval data for brevity, presenting just mean throughput figures. However, we only point out performance differences that 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 [6, Appendix 3].

4. EXPERIMENTS AND RESULTS

In this section we present the results of four 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 larger read-only transactions. The mean size of the read-only transactions in the mix is varied in this first experiment. 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 workloads consisting only of update transactions, varying the multiprogramming level, to see how the algorithms behave over a range of update conflict probabilities. Finally, Experiment 4 investigates the importance of efficient version pool management by repeating Experiment 1 under a cost model where version pool writes are assumed to be more expensive than log writes. In all four experiments, we examine the performance of both the multiversion algorithms and their single-version counterparts.

Table III contains the settings for parameters that are fixed throughout all four experiments. The database size used for the experiments is 500 pages (or

Table III. Fixed Parameters

Fixed parameter settings	
db-size	500 pages
startup-cpu	10 ms
startup-io	35 ms
cc-cpu	1 ms
obj-cpu	10 ms
obj-io	35 ms

2 megabytes, assuming 4-kilobyte pages). This rather small size was chosen to allow the simulation of transaction sizes that represent a significant fraction of the database without requiring prohibitively long simulation times. 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 (assuming that all concurrency control information is kept in tables in main memory). The cost associated with accessing a page is a 35-millisecond disk access and 10 milliseconds of CPU time to process the page. Other parameters are varied from simulation run to simulation run and are provided in the descriptions of the individual experiments. (Also, the reader is reminded that the meaning and use of each parameter are described in detail in the previous section.)

We must mention here that our parameter settings are not intended to duplicate those of real applications, and the same is true of the workloads used for our experiments. Our intention is to investigate how the algorithms compare with one another under various conditions, both in terms of performance (e.g., throughputs and response times) and storage costs. The particular conditions that we address in this paper are varied read-only transaction sizes, varied mixes of read-only and update transactions, a varied degree of conflicts among update transactions, and two different version pool write costs. Since our simulation model has many parameters, there are necessarily a number of parameters that we could have varied but did not. For example, we could have varied the granularity of the database, the write probability or the readset size for update transactions, or the various I/O and CPU cost parameters. For the purposes of this study, however, such variations were not of interest. Also, studies with such variations have been described elsewhere [1, 6, 8]. Although different parameter settings would lead to different absolute performance results, we believe that the experiments presented here do a good job of illustrating the key performance and storage trade-offs.

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 as a fraction of the overall database size. We study the relative performance of each of the multiversion algorithms, first as compared with their single-version counterparts, and then as compared with each other. The performance metrics used are the per-class throughputs and response times.

Table IV. Workload Parameters, Experiment 1

Workload parameters	
mpl	10
small-frac	0.8
small-mean	2
small-write prob	1.0
small-xact-type	Random
small-size-type	Fixed
large-mean	5, 10, 25, 50, 100
large-xact-type	Sequential
large-size-type	Uniform
large-write-prob	0.0

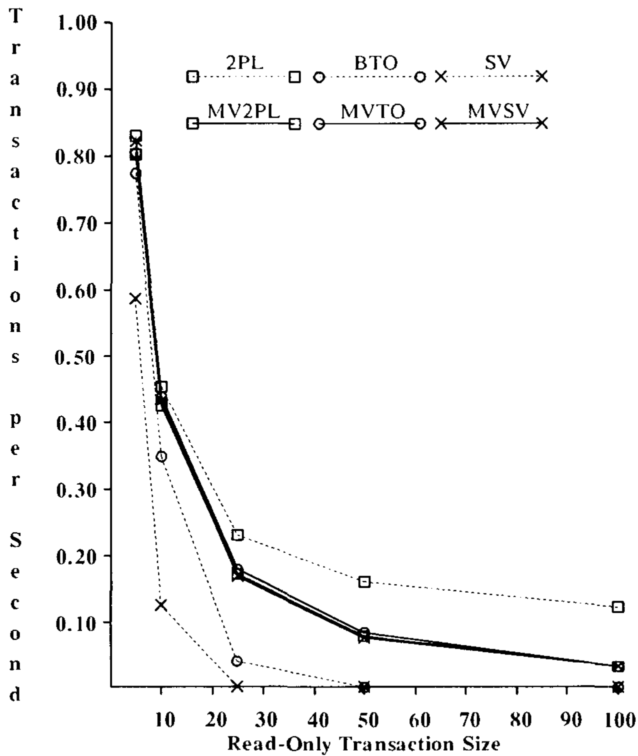

Fig. 4. Read-only transaction throughput.

(Overall throughput and response times are not examined here, as they have little or no meaning with a multiclass workload. For example, the way to maximize the overall throughput and minimize the overall response time would be simply to block large read-only transactions forever and run only small update transactions.) Also examined are the size of the version pool and the number of disk accesses required to satisfy read requests from transactions.

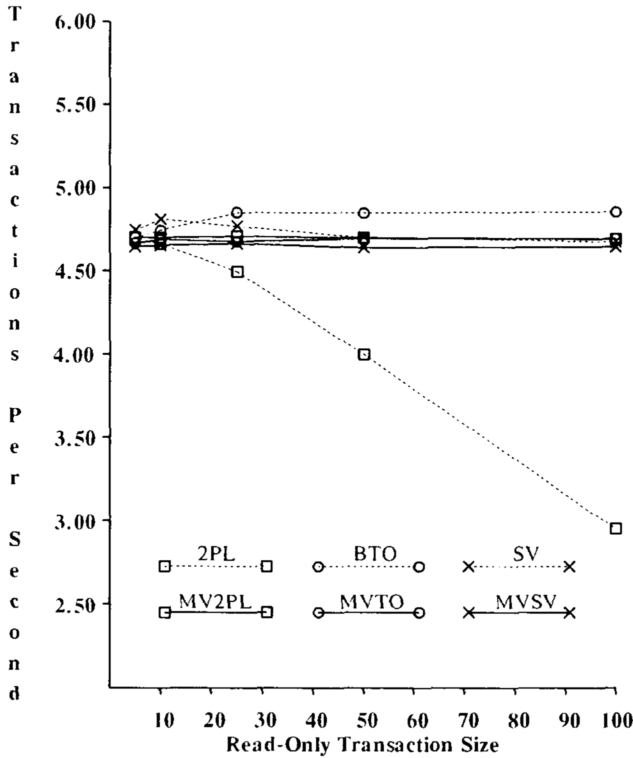

Fig. 5. Update transaction throughput.

The workload parameters for Experiment 1 are shown in Table IV. The level of multiprogramming is set to ten, so ten transactions will be in the system at all times (although some of these ten may be blocked or undergoing a restart delay). Eighty percent of these transactions are updaters, reading and then updating two randomly chosen pages, and the other 20 percent are read-only transactions. Each read-only transaction reads a number of sequential pages, and the mean size of these transactions is varied from 5 pages up to 100 pages over the set of simulations associated with this experiment. The readset size distribution for read-only transactions is uniform, ranging ±50 percent from the mean size (as described in Section 3.1). 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 what is gained by having multiple versions available to eliminate this latter source of conflicts.

Figure 4 shows the throughput results for the read-only transactions for the three multiversion concurrency control algorithms and their single-version counterparts, and Figure 5 shows the throughput results for the update transactions in the workload. Two things are evident from these figures. First, the three multiversion algorithms provide almost identical throughput, both for the

read-only transactions and the update transactions. 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 multiversion algorithms eliminate this source of conflicts, allowing read-only transactions to execute using older versions of objects and requiring update transactions to compete only among themselves for access to the objects in the database. Second, the three single-version algorithms provide quite different performance tradeoffs between the two transaction classes. 2PL provides good throughput for the large read-only transactions, but it also provides the worst throughput for the update transactions. BTO and SV, on the other hand, provide better update transaction throughput at the expense of the large read-only transactions—the large transaction throughput drops rapidly as the size of the read-only transactions is increased.

In order to understand why the single-version algorithms perform as they do, it is necessary to consider how each of them treats the two classes of transactions. Both SV and BTO are biased against large read-only transactions because of their conflict resolution mechanisms. SV restarts a transaction at the end of its execution if any of the objects in its readset have been updated during its lifetime; for large transactions, such an update is very likely, so the large read-only transactions in the mix have little hope of ever being able to commit. For read-only transaction sizes of 25 (5 percent of the database) or more, these transactions are simply being restarted over and over, and their throughput rate is zero. Similarly, BTO restarts a transaction any time it attempts to read an object with a timestamp newer than its startup timestamp, meaning that the object has been updated by a transaction that started running after this transaction did. Again, this becomes very likely as the read-only transaction size is increased, and read-only transactions are "starved out" by the update transactions when the read-only transaction size is 10 percent or more of the database. In contrast, 2PL has the opposite problem. Read-only transactions set locks on objects that they read, perhaps occasionally waiting briefly while an update transaction completes, and then hold these locks for the remainder of their execution time. Thus, read-only transactions can hold locks on a significant portion of the database for quite a long time when their size is large. Update transactions that wish to update locked objects must wait a long time in order to lock and update these objects (half the read-only transaction execution time, roughly).[4] This is evident in Figure 5, where 2PL's update transaction throughput decreases significantly when the size of the read-only transactions in the mix exceeds 5 percent of the database size.

Figure 6 presents the response time results for the read-only transactions. For the most part, the trends in this figure reflect the throughput trends observed in Figure 4. It is again evident that, as compared with the multiversion algorithms, 2PL favors large read-only transactions, while BTO and SV are biased against them. In fact, the response time curves for BTO and SV end early because no large read-only transactions at all were able to complete successfully during the simulations beyond the points shown. Figure 6 also shows a slightly better response time for large read-only transactions under MVTO as compared with

[4] Also, since update transactions tend to be younger in comparison with read-only transactions, they will be chosen as the victims when deadlocks arise involving read-only and update transactions.

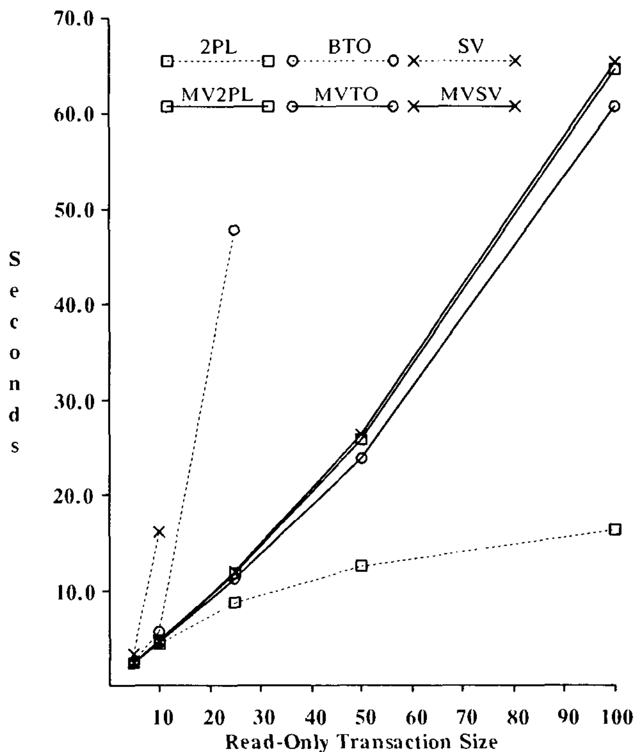

Fig. 6. Read-only transaction response time.

the other multiversion algorithms. This trend is explained by Figure 7, which gives the average number of disk accesses required to satisfy an individual read request from a read-only transaction. Figure 7 shows that the average number of versions read by a large read-only transaction is slightly lower under MVTO. The reason for this is that, in the MV2PL and MVSV algorithms, versions are stamped with the commit timestamp of their creator. Read-only transactions read the most recent committed version of an object, a version that was committed before the read transaction started running. In MVTO, however, versions are instead stamped with the startup timestamp of their creator, and read-only transactions read the most recent version that has a timestamp less than their own startup timestamp. Note that this version does not have to have been created by a transaction that committed quite as long ago (i.e., prior to this one's startup)—it may have been created by a transaction that started running right before this one. As a result, large read-only transactions end up reading slightly newer data under MVTO than under MV2PL or MVSV, and thus the version chains that they follow to find the desired data are slightly shorter on the average.

Figure 8 shows the response time results for the update transactions. Again, the trends in this figure closely reflect the throughput trends observed in Figure 5. In Figure 8, BTO is seen to provide better performance than SV for the update transactions when the size of the read-only transactions in the mix is large; this trend is visible in Figure 5 as well. This is explained by Figure 9, which

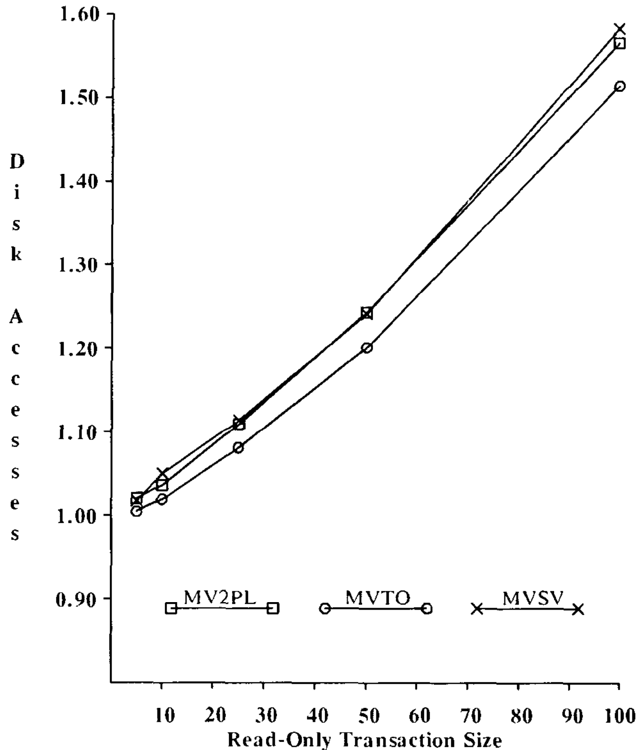

Fig. 7. Read-only transaction version accesses.

shows the fraction of the available disk time (since the disk is the bottleneck resource with our parameter settings) that is spent processing requests from transactions that were later restarted. It can be seen in the figure that SV is the most wasteful of the algorithms, as it uses the largest amount of the overall disk capacity for processing requests that were subsequently rendered useless. This is because SV restarts transactions at the end of their execution, whereas at least BTO, the second most wasteful algorithm, tends to restart transactions earlier when restarts are called for. Wasting more resources leads to lower throughput and higher response times in most situations, as shown in [1, 8].

Having studied the performance of the multiversion algorithms for this experiment, we now consider their storage overheads. Figure 10 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. The three multiversion algorithms have similar storage overheads, 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 is why the relative size of the version pool increases with read-only transaction size. As shown in the figure, the version

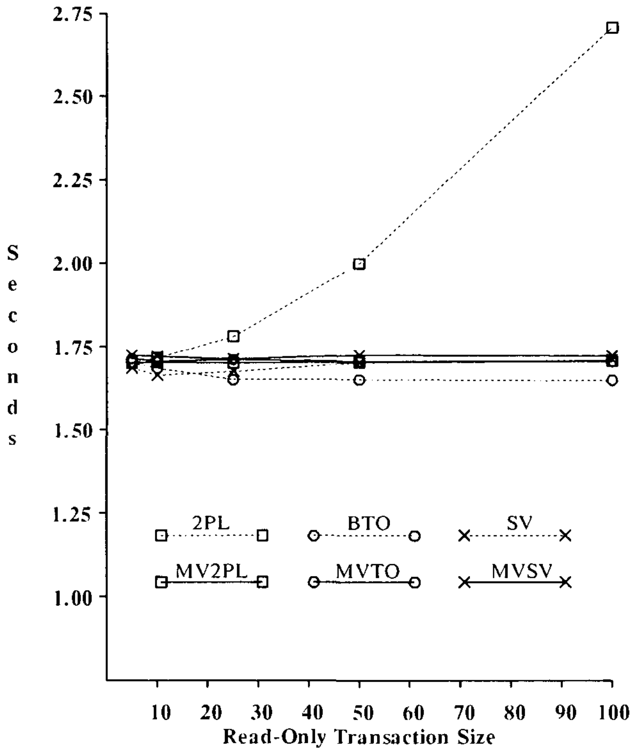

Fig. 8. Update transaction response time.

pool is quite small until the read-only transaction size reaches 25 (5 percent of the database size), at which point the version pool size is about 20 percent of the database. When the read-only transaction size is 10 percent of the database size, the version pool size is approximately 40 percent of the size of the database. Finally, when each read-only transaction reads about 20 percent of the database, the version pool becomes as large as the database itself, as a large number of update transactions can complete during the execution of a single read-only transaction at this point.

Figure 11 shows the average number of version accesses (i.e., disk reads) required for accessing a data object, including reads by both the update transactions and the read-only transactions. The trends are similar to those of Figure 7, which shows the same metric with the update transaction reads excluded. The average number of disk accesses (over all reads) is less than 1.07 or so as long as read-only transactions read no more than 10 percent of the database, and it is still less than 1.15 when read-only transactions read as much as 20 percent of the database. Of course, as one would expect, the average number of disk accesses is higher if only the read-only transactions are considered (as was the case in Figure 7).

Tables V–VII each contain three results pertaining to the version access behavior of transactions for the MV2PL algorithm for read-only transaction

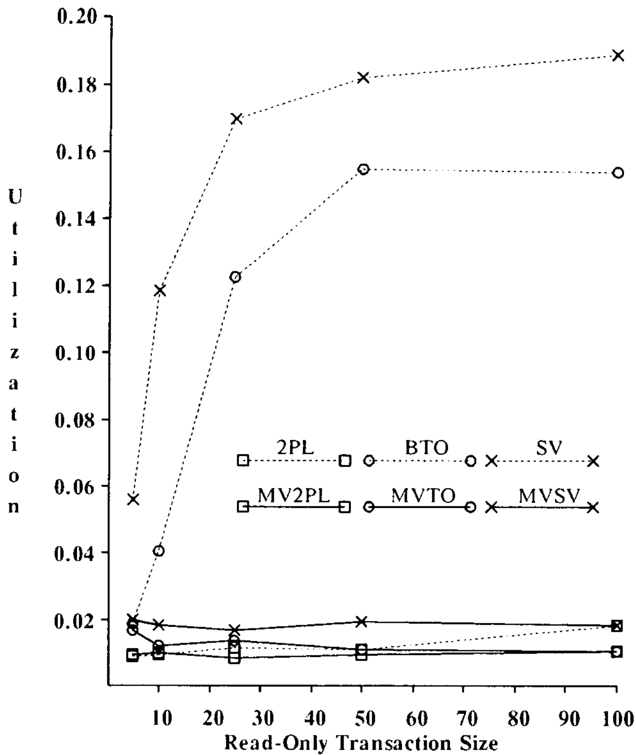

Fig. 9. Wasted disk utilization.

sizes of 25, 50, and 100 (respectively). The results for MVTO and MVSV were similar, so MV2PL 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 request. The next two columns in each table pertain to the number of disk accesses per read for the update and read-only transactions, respectively. The last column pertains to the number of disk accesses per read overall (i.e., regardless of the requesting transaction's class).

In all three tables, as expected, read requests for the update transactions were always satisfied in one disk access. In addition, Table V shows that, with a mean read-only transaction size of 25, or 5 percent of the database, almost 90 percent of the read requests for the read-only transactions were satisfied in a single disk access. Ninety-nine percent of the read requests from these transactions were satisfied in only one or two disk accesses, and less than 1 percent needed three or four disk accesses. No read request ever required more than four disk accesses at this read-only transaction size setting. The last column in the table shows that, overall, about 97 percent of all read requests were satisfied in only one disk access. Table VI shows the same collection of statistics for the case in which the mean size of the read-only transactions is 50, or 10 percent of the overall database size. The results are similar to those in Table V, though the percentages for more than one disk access are a bit higher in this case. Nearly 80 percent of all read

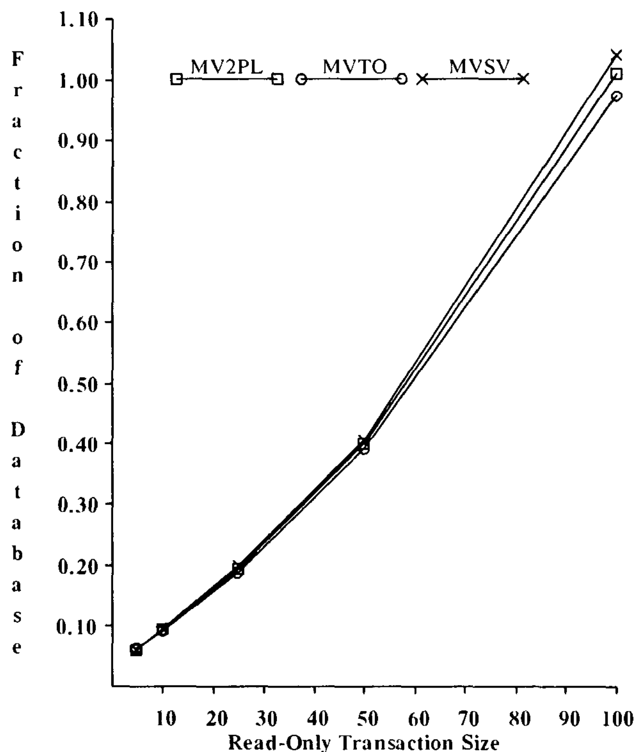

Fig. 10. Relative version pool size.

requests from the read-only transactions were satisfied in a single disk access, and 97 percent of the requests required just one or two disk accesses. Just over 3 percent of the read requests necessitated three or four disk accesses in this case, with none requiring more than that. Ninety-four percent of all read requests, overall, required only one disk access.

Table VII shows the version access statistics for the most extreme case examined, where the mean read-only transaction size is 100, or 20 percent of the database size. Even with this extremely large read-only transaction size, over 90 percent of all reads were satisfied in just one disk access. However, it is evident here that read-only transactions did encounter a greater percentage of reads requiring multiple disk accesses. Only about 62 percent of all read requests from the read-only transactions could be processed in one disk access, with the remainder of the first 87 percent of the requests requiring two accesses. Another 9 percent of the read requests required three disk accesses, and another 3 percent required four accesses. The remaining 1–2 percent of the requests required between five and eight disk accesses in this case. It is clear from these results that, even in extreme cases, the vast majority of reads do not need to use a version other than the most recently committed version. Note, however, that the performance results discussed earlier do indicate that having old versions available for those accesses that need them is definitely beneficial.

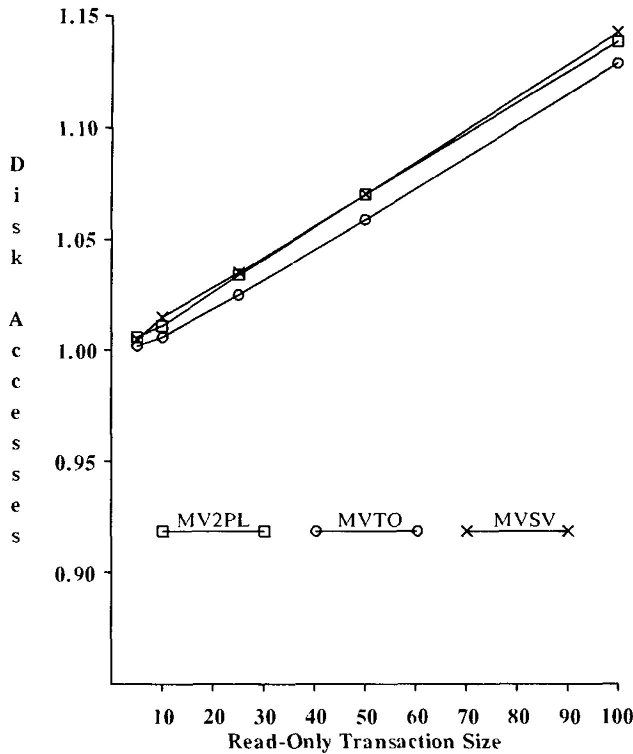

Fig. 11. Average version accesses.

Table V. Versions Accessed, Read-Only Transaction Size = 25, MV2PL

Disk accesses	Percent of update transaction reads	Percent of read-only transaction reads	Percent of all reads
1	100.00	89.89	96.90
2	0.00	9.33	2.86
3	0.00	0.73	0.23
4	0.00	0.05	0.01

Considering the results of Experiment 1 as a whole, several conclusions can be drawn at this point. First, it was seen that multiversion concurrency control algorithms are definitely beneficial from a performance standpoint—all three of the algorithms examined here outperformed their single-version counterparts in some respect. The MV2PL algorithm remedied a problem that 2PL was seen to have: For medium to large read-only transaction sizes, the response time for the update transactions degraded quickly as the read-only transaction size was increased. Similarly, MVSV and MVTO remedied the (dual) problem that SV and BTO were found to have: As the read-only transaction size was increased, the large read-only transactions quickly began to be starved out (i.e., restarted over and over again) because of updates made by the update transactions in the

Table VI. Versions Accessed, Read-Only Transaction Size = 50, MV2PL

Disk accesses	Percent of update transaction reads	Percent of read-only transaction reads	Percent of all reads
1	100.00	79.31	94.05
2	0.00	17.46	5.02
3	0.00	2.83	0.81
4	0.00	0.40	0.12

Table VII. Versions Accessed, Read-Only Transaction Size = 100, MV2PL

Disk accesses	Percent of update transaction reads	Percent of read-only transaction reads	Percent of all reads
1	100.00	61.80	90.63
2	0.00	25.36	6.22
3	0.00	8.79	2.15
4	0.00	2.81	0.69
5	0.00	1.04	0.25
6	0.00	0.15	0.04
7	0.00	0.03	0.01
8	0.00	0.02	0.01

workload. Our results also confirm results reported for BTO versus MVTO in [18], as we found that multiple versions do not help as much when the read-only transactions are small. In addition, we found that the same is true for 2PL versus MV2PL (and also for SV versus MVSV to some extent). Finally, in analyzing the storage-related aspects of the algorithms, it was found that, for read-only transactions that read less than 5 percent of the database on the average, the size of the version pool was less than 20 percent of the size of the database; only when read-only transactions read more than 10 percent of the entire database did the size of the version pool exceed 40 percent of the database size. It was also found that, even in extreme cases, the vast majority of read requests could be satisfied in a single disk access.

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, the size of read-only transactions 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 how the performance benefits associated with multiversion algorithms vary with the transaction mix. Also investigated is the way in which the storage cost (i.e., the size of the version pool) varies with the mix.

Table VIII gives the parameter settings used in this experiment. The level of multiprogramming is set to ten, as in Experiment 1. The update transactions in the mix again read and update two objects. The mean size of read-only transactions is 50 in this case, meaning that each read-only transaction reads between 25 and 75 sequential pages. As mentioned above, the mix of transactions in the

Table VIII. Workload Parameters, Experiment 2

Workload parameters	
mpl	10
small-frac	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	1.0
large-mean	50
large-xact-type	Sequential
large-size-type	Uniform
large-write-prob	0.0

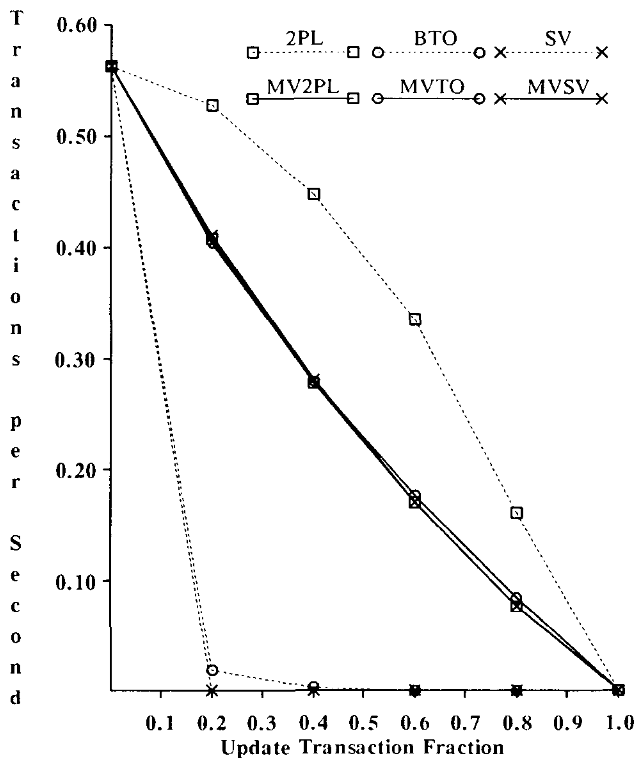

Fig. 12. Read-only transaction throughput.

workload is varied in this experiment. In particular, workloads with 0, 20, 40, 60, 80, and 100 percent update transactions are studied, with the remainder of the workload consisting of read-only transactions.

Figure 12 shows the throughput results for the read-only transactions for this experiment, and Figure 13 shows the throughput results for the update transactions in the workload. Figures 14 and 15 give the corresponding response-time results. Figure 14 does not show results for the SV and BTO algorithms, except

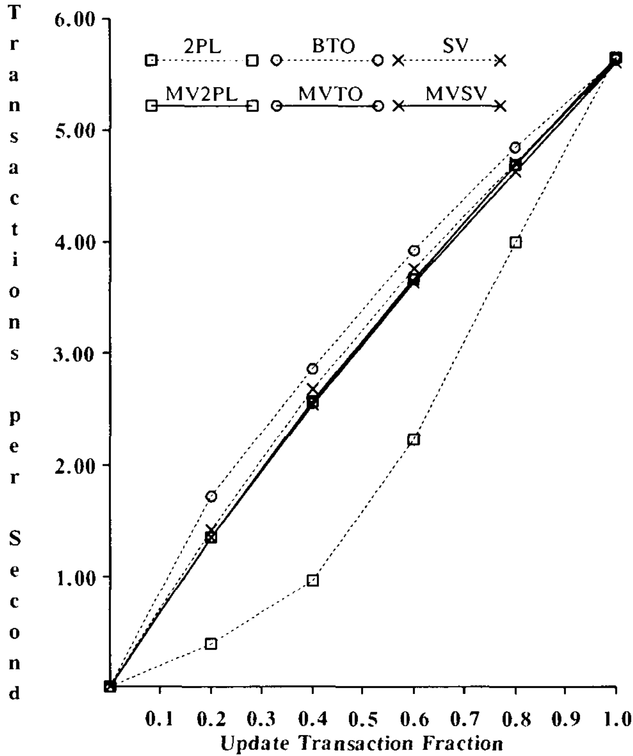

Fig. 13. Update transaction throughput.

when the mix has no update transactions; no read-only transactions were able to complete beyond this point under SV, and those that did complete under BTO (at the 20 and 40 percent settings) had an average response time more than ten times worse than that of the multiversion algorithms (which would throw the scale of the graph way off). In both sets of figures, the multiversion algorithms perform nearly identically—as in Experiment 1, the workload here is such that the multiversion algorithms eliminate most concurrency control conflicts. For these algorithms, the read-only transaction throughput decreases linearly as the fraction of update transactions in the workload is increased. The update transaction throughput also varies linearly, but in the opposite manner. This is expected since, without conflicts, the throughput of each class is simply proportional to the number of terminals submitting transactions of that class. The linear increase in the response times for read-only transactions under the multiversion algorithms, shown in Figure 14, is due to the fact that the average number of disk accesses required to satisfy their read requests increases with the fraction of update transactions in the mix (as we shall see shortly). This factor is another contributor to the throughput decrease for read-only transactions in Figure 12. The slight performance advantage that MVTO has in Figures 12 and 14 is due to the fact that read-only transactions read slightly newer data in MVTO, as explained in Experiment 1, so read-only transactions incur somewhat fewer extra disk accesses when MVTO is used.

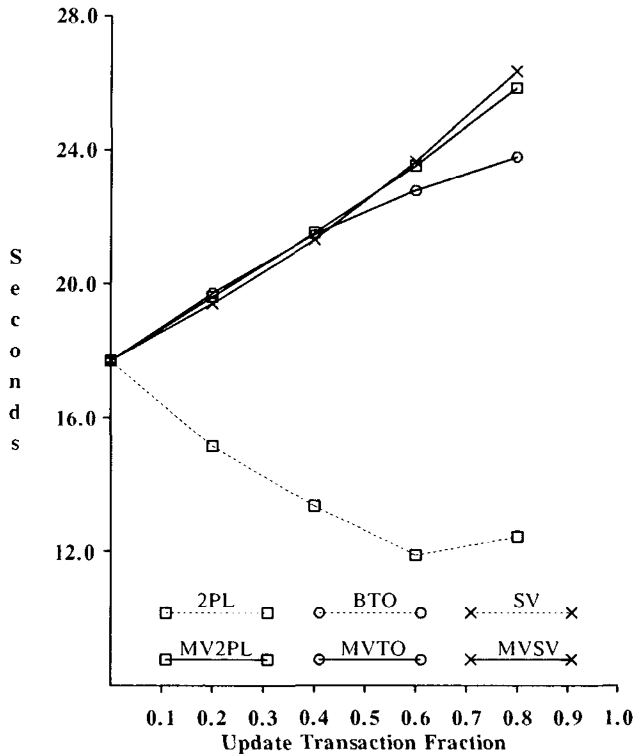

Fig. 14. Read-only transaction response time.

Turning to the single-version algorithm performance results, it is evident from Figure 12 that it takes only a small fraction of update transactions to cause the read-only transactions to starve in the BTO and SV algorithms (i.e., to be repeatedly restarted), causing read-only transaction performance to degrade rapidly as the fraction of update transactions is increased. Both BTO and SV provide slightly better performance for the update transactions than the multi-version algorithms because of this, as shown in Figures 13 and 15. This is because a larger fraction of the system's CPU and I/O resources are available for processing update transactions while read-only transactions are undergoing restart delays. BTO provides slightly better performance than SV does for the update transactions for the reasons discussed in Experiment 1. As for 2PL, Figures 12–15 again show that 2PL provides better performance for the large read-only transactions than the other algorithms, but worse performance for the update transactions, which is consistent with what we saw in Experiment 1. Because update transactions are usually blocked waiting for locks held by read-only transactions, more of the system's resources are available for serving the read-only transactions. The update transactions suffer the most under 2PL when the majority of transactions are read-only transactions, as evidenced by Figure 15. All three single-version algorithms perform the same as their multi-version counterparts at the two extremes, where the mix consists of either all read-only transactions or all update transactions; this is because there are no

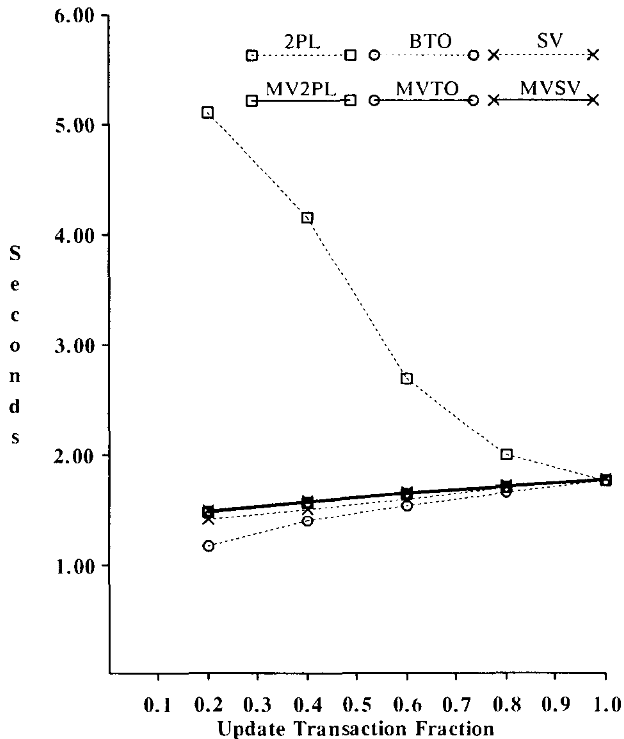

Fig. 15. Update transaction response time.

conflicts when the mix is devoid of updaters, and there are virtually no conflicts at the other extreme with the parameter settings used here.

Figures 16 and 17 show the storage-related results obtained in this experiment. The results are basically what one would expect, given the storage results of Experiment 1 and the performance results just examined. The relative size of the version pool is greatest with 80 percent updaters and 20 percent readers, which is where the largest number of updates take place during the lifetime of the read-only transactions. The average number of disk accesses needed to satisfy read requests from read-only transactions is also the greatest for this mix of transactions. Figure 17 shows the disk access behavior for both the read-only transactions alone and the overall mix (including update transactions). The overall average actually peaks at the mix of 60 percent updaters and 40 percent readers; this is because the update transactions have their requests satisfied in a single disk access, and the 80 percent of the mix that are update transactions pull the overall average value down toward 1.0 at the 80 percent setting. The slight version chain length advantage that MVTO has over MV2PL and MVSV is also shown by Figure 17.

4.3 Experiment 3: Update Conflicts

This experiment examines the behavior of the algorithms under a workload consisting almost entirely of update transactions. Whereas Experiments 1 and 2 examined the performance of each of the multiple-version algorithms under

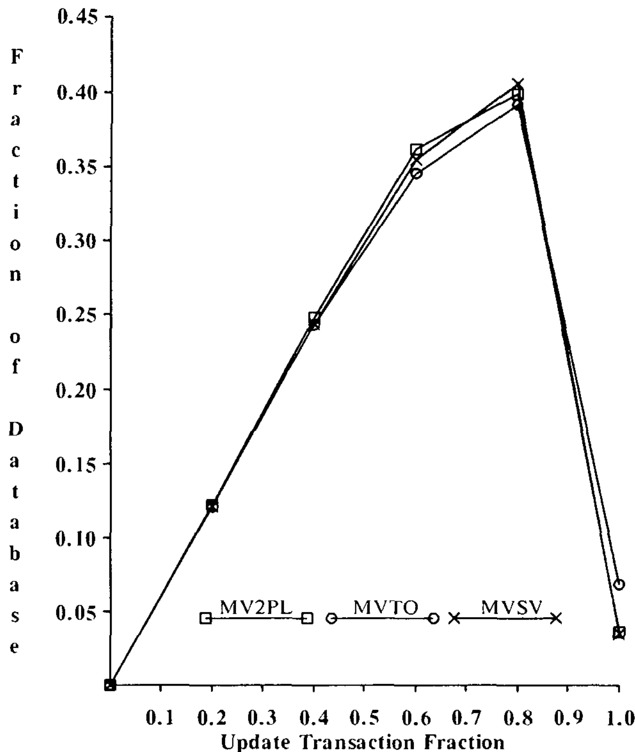

Fig. 16. Relative version pool size.

conditions that were favorable for the algorithms, we examine their performance under less favorable conditions in this experiment. The size of update transactions is somewhat larger here, and the multiprogramming level is varied in order to see how each of the algorithms performs under varying conflict probabilities (as in [1]).

The workload parameters for Experiment 3 are given in Table IX. The transactions in this experiment read a random number of pages and then update each page with 50 percent probability. The number of pages read is selected from a uniform distribution with a mean of 6 pages, so transaction readsets range between 3 and 9 pages in size. These pages are selected at random from among all of the pages in the database. The multiprogramming level is varied from 1 to 50, as indicated in the table. At the extreme, then, conflicts between update transactions become highly probable—the average transaction reads 6 pages and updates 3, and there are 50 such transactions in the system at a time, all with a database size of only 500 pages.

Figures 18 and 19 give the throughput and response time results for Experiment 3. These results indicate that the single- and multiple-version counterparts of each algorithm provide extremely similar performance, which is expected since the mix consists almost entirely of update transactions. (Since transactions read an average of six items and update each one with 50 percent probability, the

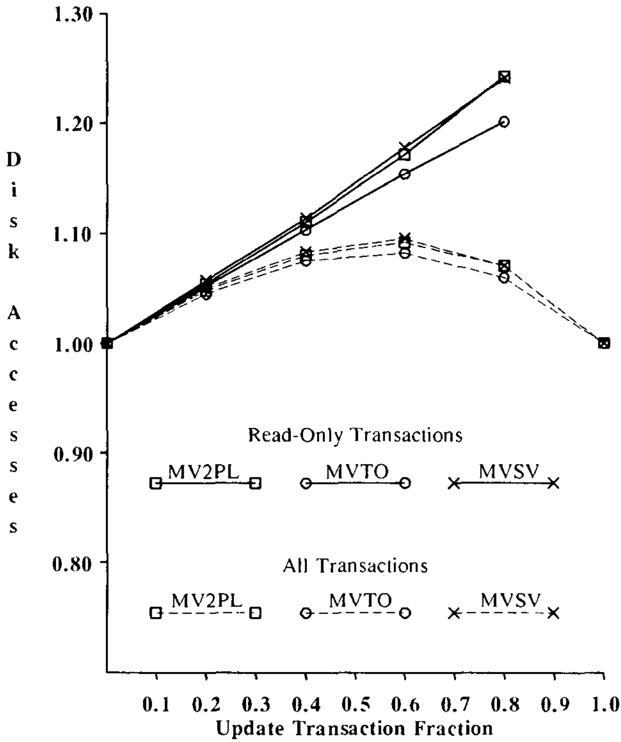

Fig. 17. Version accesses.

Table IX. Workload Parameters, Experiment 3

Workload parameters	
mpl	1, 5, 10, 20, 50
small-frac	1.0
small-mean	6
small-xact-type	Random
small-size-type	Uniform
small-write-prob	0.5

likelihood of a newly arrived transaction not updating any of the items that it reads is very small.) All three of the multiversion algorithms and their single-version counterparts perform similarly at low multiprogramming levels, where there are too few concurrent transactions to lead to many conflicts, but their performance diverges at higher multiprogramming levels. MV2PL (and 2PL) perform significantly better than the other algorithms when conflicts become likely, with MVTO (and BTO) performing just slightly better than MVSV (and SV) under these circumstances. These results are due to the large fraction of the disk resources that the timestamp-based and optimistic algorithms waste due to restarts. This is evident in Figure 20, which shows the fraction of the disk

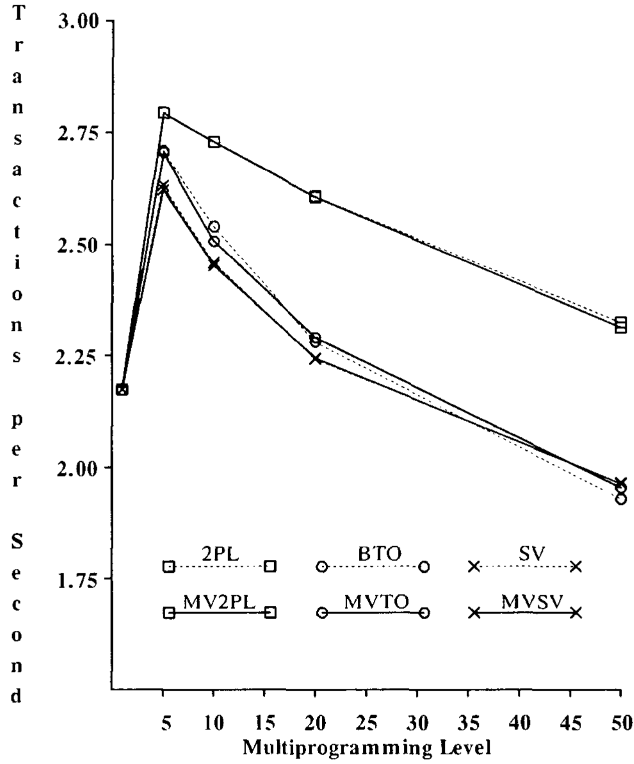

Fig. 18. *Transaction throughput.*

resources wasted by transactions that were later restarted. The locking algorithms attempt to block transactions instead of restarting them to resolve conflicts (unless deadlocks occur), so they waste fewer resources and therefore get more useful work done. As a result, the locking algorithms provide higher throughput and lower response times. Similar results were reported for blocking-oriented versus restart-oriented algorithms in [1, 8]. Finally, the shape of the curves in Figure 18 is easily explained; starting at the lowest multiprogramming level, increasing the multiprogramming level first increases throughput because it decreases the idle time for the disk, but then throughput decreases owing to conflicts that lead to transactions being restarted and resources being wasted.

Figure 21 shows how the relative version pool size behaves as a function of the multiprogramming level in this experiment. Note that, with no read-only class of transactions in the mix, the version pool simply serves as an UNDO log, and its size is determined by the number of objects that are currently (or were recently) undergoing updates. In particular, since the version pool is managed as a circular buffer, it holds the before-images of all objects that were updated since (and including) the first update made by the oldest active transaction instance in the system. Thus, Figure 21 basically shows the size of the UNDO log for the algorithms. Table X gives statistics that shows the lifetime of the average instance of a transaction. The lifetime is defined as the time from startup until

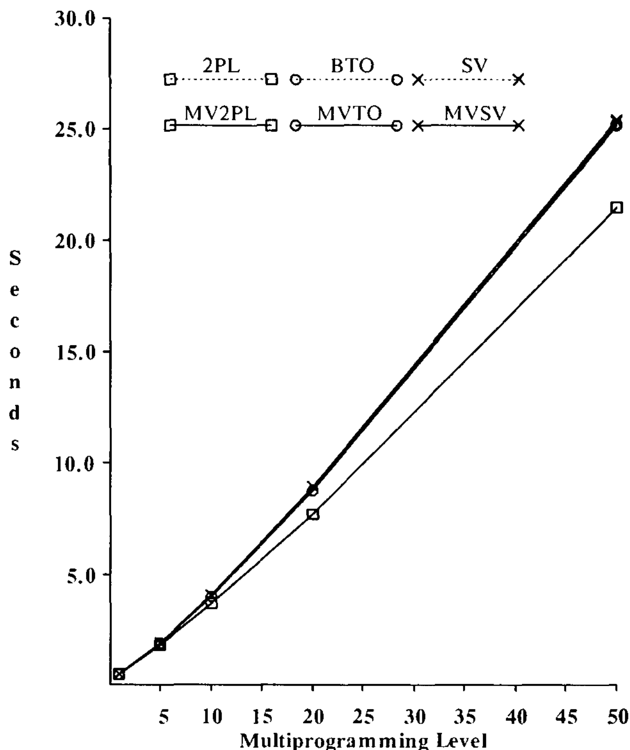

Fig. 19. Transaction response time.

either a commit or a restart for an instance of a transaction; this is different from the transaction's response time, which is the sum of the lifetimes of all of its instances, including all restarted instances, plus the restart delays incurred in between. As is evident from comparing Figure 21 and Table X, there seems to be a correlation between the average transaction instance lifetime and the average version pool size, which makes sense since the lifetime of the oldest transaction at a given point in time will influence the number of before-images in the version pool. Note, however, that this correlation is not perfect—over much of the multiprogramming level range, MVTO has a slightly larger relative version pool size than MV2PL despite its shorter average lifetime values.

4.4 Experiment 4: Version Pool Write Cost

This last experiment examines the behavior of the algorithms under a different set of assumptions about the version pool implementation for the multiversion concurrency control algorithms. As described in Section 3.4, we have assumed thus far that the cost of writing the before-image of an object to the version pool is comparable with the cost of writing the before-image of an object to the log. The former cost is the UNDO-related recovery cost for the multiversion algorithms, and the latter cost is the UNDO-related recovery cost for the single-version algorithms. In this section we ask the question, "What if the version pool

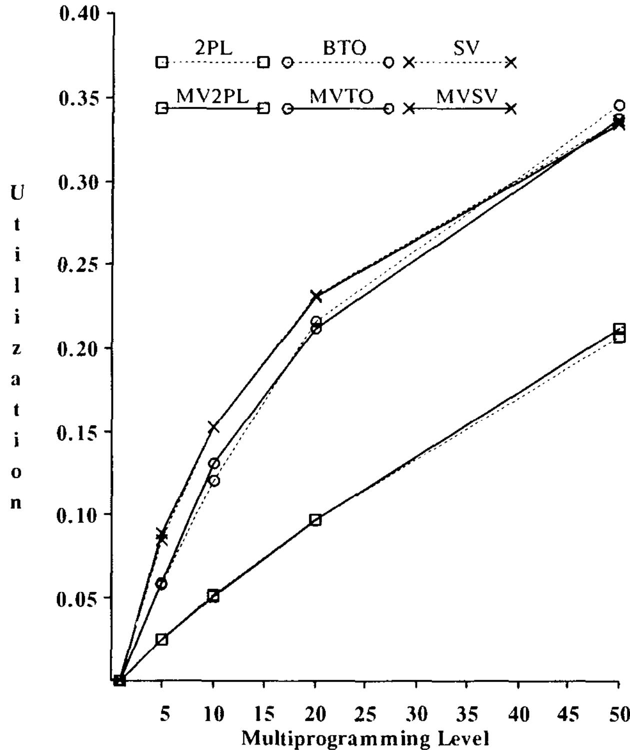

Fig. 20. Wasted disk utilization.

is not stored on one or more dedicated disks, and thus version pool writes are equivalent in cost to normal disk writes?" We investigate the answer by assuming that the before-image copying cost for the multiversion algorithms is equal to the difference between the cost of a random disk write, which we have assumed to be 35 milliseconds (in Table III), and the cost of a sequential log disk write, which we assume to be about 40 percent of the cost of a random write, or 14 milliseconds. This provides a rough way to account for the incremental cost that the multiversion algorithms would incur as compared with the single-version algorithms if the version pool were not managed in a loglike fashion; a more realistic approach would involve integrating a detailed queuing model of the recovery subsystem into our simulation model for all of the algorithms. We opted for the simpler, rougher cost model because the purpose of this experiment is simply to illustrate the importance of an efficient version pool implementation, and also because a more detailed model would add quite a bit of complexity to our simulator. Given this modified cost model, we repeated Experiment 1 (whose workload parameters were given in Table IV).

Figures 22–25 give the throughput and response time results that were obtained by rerunning Experiment 1 under the modified cost model. The results for the update transactions are very different here. As shown in Figures 23 and 25, the performance of the update transactions is strongly affected by the increased

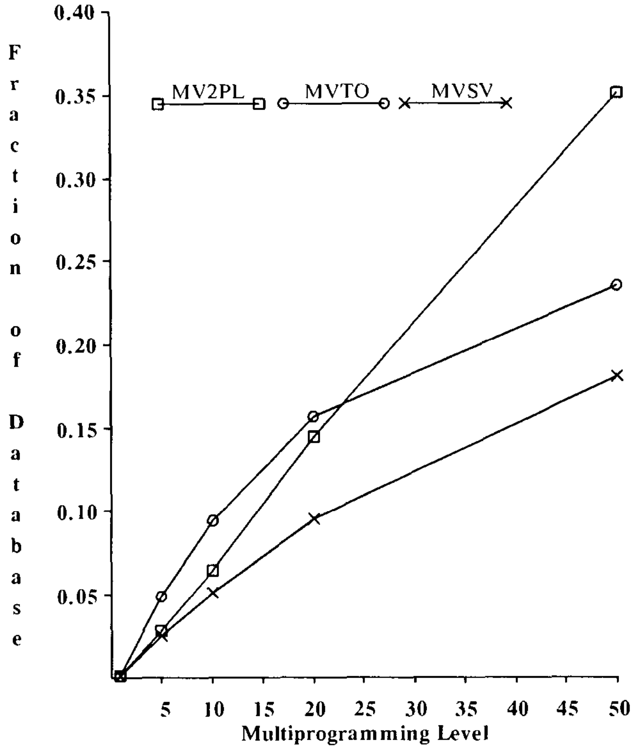

Fig. 21. Relative version pool size.

Table X. Transaction Instance Lifetimes (seconds)

MPL	MV2PL mean (std. dev.)	MVTO mean (std. dev.)	MVSV mean (std. dev.)
1	0.460 (0.149)	0.460 (0.149)	0.460 (0.149)
5	1.677 (0.513)	1.612 (0.463)	1.518 (0.470)
10	3.193 (1.025)	2.805 (0.791)	2.610 (0.759)
20	5.730 (2.227)	4.556 (1.239)	4.193 (1.229)
50	10.530 (5.637)	6.981 (1.836)	6.928 (1.945)

before-image copying cost—their performance is about 20 percent worse than it was under the old cost model, making the multiversion algorithms less attractive here. For SV and BTO versus MVSV and MVTO, the multiversion algorithms are still needed to avoid starvation of read-only transactions; however, under the cost model in this experiment, about 20 percent of the update transaction performance must be sacrificed in order to help out the read-only transactions. For 2PL versus MV2PL, 2PL is now the more attractive algorithm until the read-only transaction size approaches 20 percent of the overall database size; before this point, the before-image copying cost associated with MV2PL outweighs the advantages of reduced waiting time for the update transactions. As for how read-only transaction performance is affected by the modified cost model,

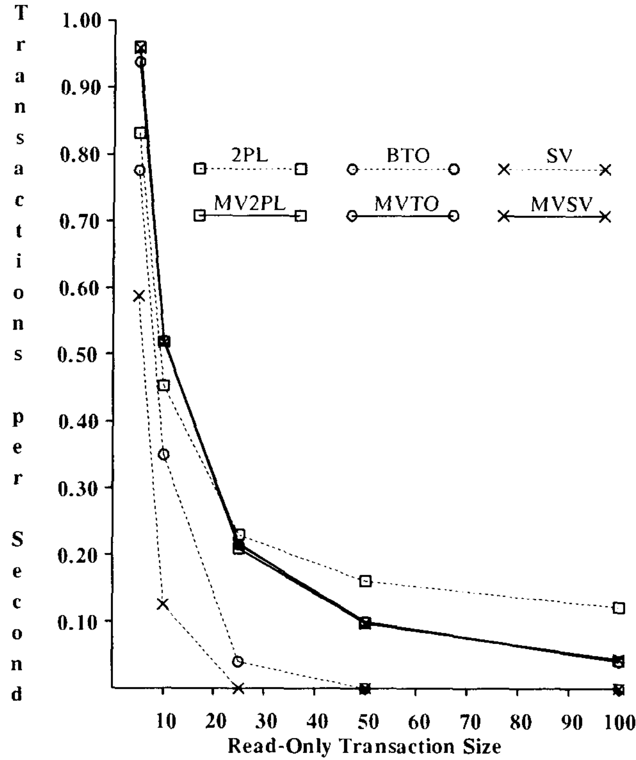

Fig. 22. Read-only transaction throughput.

the read-only transactions actually have somewhat better performance under the modified model. They do not update objects, so they do not suffer the cost of before-image copying; instead, they gain the advantage that, because the response time for the update transactions is larger here, fewer can run during their lifetime. As a result, fewer updates occur during the lifetime of read-only transactions, version chains are shorter on the average, and thus they end up incurring fewer additional disk accesses. It is clear from these results that it is important that the cost of before-image copying be minimized so that the results of Experiment 1, and not these results, are more indicative of the true performance of the multiversion algorithms (as argued in Section 3.4).

The storage-related results that were obtained in Experiment 4 are similar to those presented for Experiment 1, so we do not show them here. The only changes were that the size of the version pool and the average version chain lengths in this experiment came out significantly smaller than those of Experiment 1 for the reasons described in the previous paragraph. For example, the average number of versions accessed to process a read-only transaction's read request dropped from nearly 1.6 in Experiment 1 to about 1.35, or by over 15 percent, for a read-only transaction size of 20 percent of the database. The relative version pool size at this extreme dropped from about 1.0 to a little under 0.6. The overall shapes of the version and storage-related curves remained the same, however.

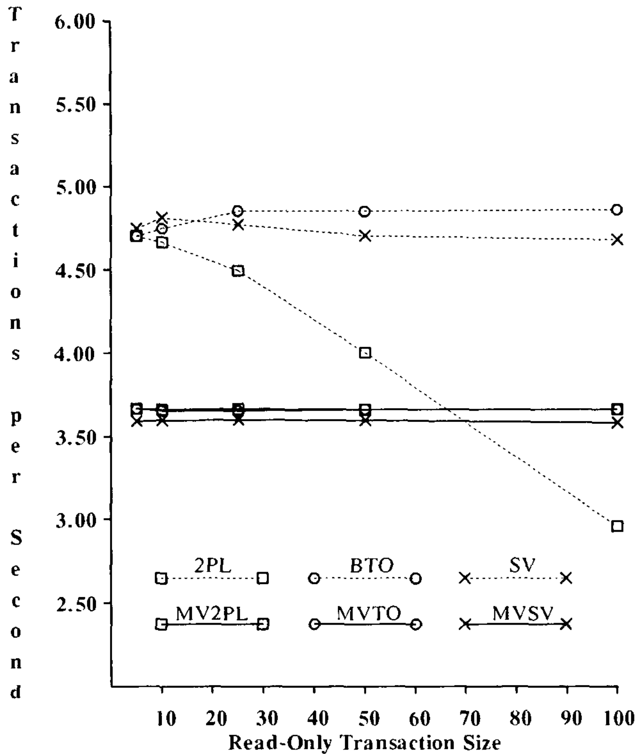

Fig. 23. Update transaction throughput.

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 multiversion locking algorithm, and a multiversion variant of Kung and Robinson's serial validation algorithm [16]. 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 offer performance advantages over their single-version counterparts under such a workload. Since the probability of conflicts was nearly zero for the multiversion algorithms with this workload, the three multiversion algorithms themselves performed almost identically. It was seen that MVSV and MVTO significantly outperformed SV and BTO with respect to large read-only transactions, as both SV and BTO tended to starve this transaction class in favor of the small update transactions in the mix. For MV2PL, the overall throughput and response-time results indicated a different trade-off than those for 2PL. 2PL was found to favor large read-only

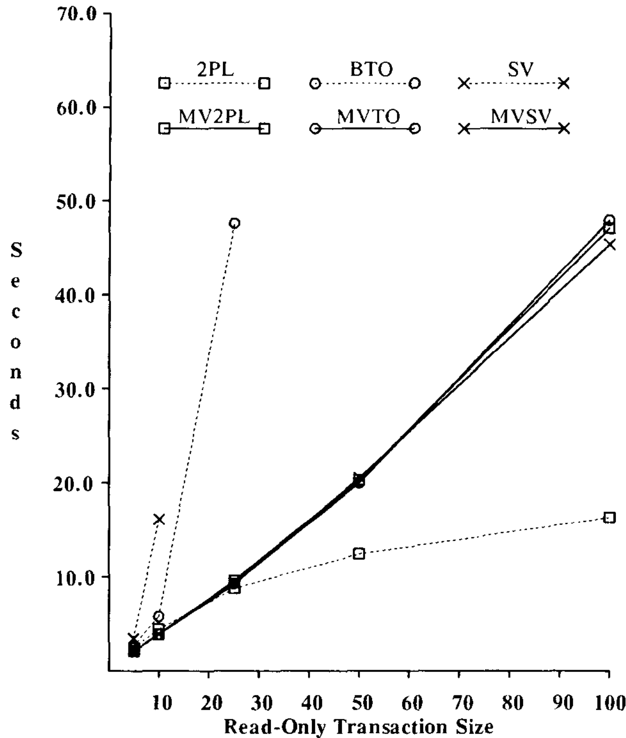

Fig. 24. Read-only transaction response time.

transactions over small update transactions, leading to poor update transaction performance. MV2PL remedied this problem, providing better response time than 2PL for small update transactions at some expense in the performance of large read-only transactions (i.e., it did not favor them over updaters). This is a significant advantage for some applications, such as on-line transaction processing.

In terms of their storage characteristics, all three multiversion algorithms were again fairly similar in Experiment 1. For the parameter settings used here, it was found that the size of the version pool was less than 20 percent of the size of the database when read-only transactions read an average of less than 5 percent of the overall database; only when read-only transactions read more than 10 percent of the entire database did the size of the version pool exceed 40 percent of the database size. These results concur with Reed's claims of reasonable overhead for his (somewhat different) storage management scheme [24]. It was also found that, even in extreme cases, the vast majority of read requests could be satisfied in a single disk access. When read-only transactions read an average of 10 percent of the database, it was seen that almost 95 percent of all read requests were still able to be handled in only one disk access. Considering only the read-only transaction reads in this situation, it was still found that about 80 percent of their requests were handled in a single disk access. Finally, it was seen that, despite the cost of accesses to old versions, the performance benefits of the

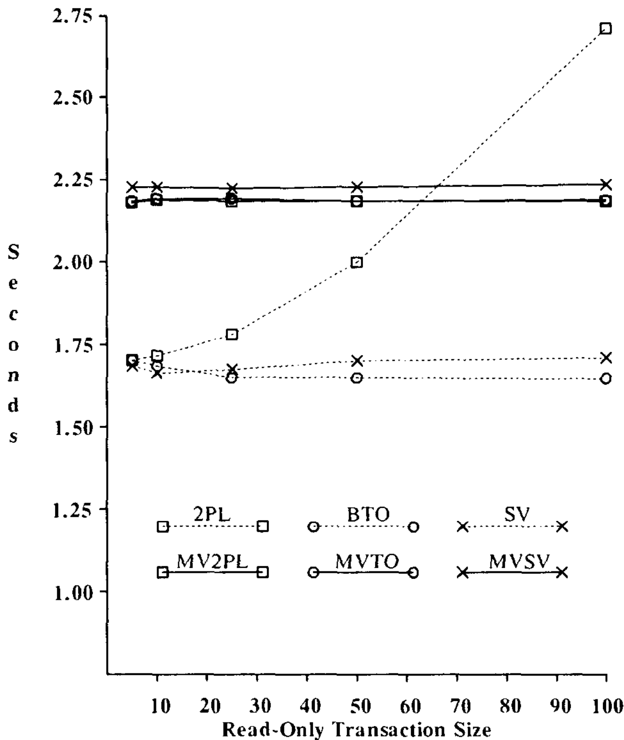

Fig. 25. Update transaction response time.

multiversion algorithms clearly outweighed the costs. While the exact storage overhead figures are certainly dependent on workload particulars, such as the multiprogramming level of the database system and the size and update characteristics of the update transactions in the workload, the basic trends are fairly clear from these results. In addition, our workload—with ten transactions *always* present in the system, and with the read-only transactions reading a significant portion of the database—is probably a stressful workload by practical standards. Thus, we would expect the storage overheads and average version chain lengths in actual systems to be smaller than those observed here.

Three other experiments were also performed. Experiment 2 examined the performance and storage characteristics of the algorithms as a function of the fraction of read-only transactions in the mix. It was found that only a small fraction of update transactions are required for BTO and SV to begin starving out long read-only transactions, and therefore for the advantages of MVTO and MVSV to show up. For the locking algorithms, small update transactions were found to have a much better response time under MV2PL than under 2PL when the workload was comprised mainly of large read-only transactions. It was also found that the relative size of the version pool was the greatest with 80 percent update transactions and 20 percent read-only transactions, where the most updates occurred during the lifetimes of the read-only transactions. Experiment 3 examined the performance of the algorithms for a mix consisting solely of

update transactions, and the multiprogramming level of the system was varied in this experiment. The multiversion algorithms provided the same performance as their single-version counterparts in this experiment, as expected, and 2PL and MV2PL were found to outperform the other algorithms. Finally, Experiment 4 examined the performance of the algorithms under a modified model of the version pool write cost, but with assumptions otherwise similar to those of Experiment 1. The performance of the small update transactions suffered under the multiversion algorithms due to the added overhead associated with copying the before-images of updated objects to the version pool, making it clear that the version pool should be treated in a manner similar to that of a log in conventional systems.

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 nature of these improvements depends on the multiversion algorithm in question. The added costs that arise due to following version chains for read requests are not all that 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 that might be required to satisfy read requests from ongoing transactions is not that large—in our experiments, the average size of the version pool only began to be fairly significant when the read-only transactions read more than 5–10 percent of the entire database or when the number of update transactions in the system was very large (as was the case in Experiment 3 at the highest multiprogramming levels). Although the actual performance and storage overhead figures will vary depending on workload and implementation details, we believe that our results provide a good picture of the costs and benefits associated with the multiversion approach to concurrency control.

ACKNOWLEDGMENTS

The authors wish to acknowledge helpful discussions that we had with Mary Vernon regarding approaches to modeling multiclass transaction workloads. Comments from the referees on earlier versions of this paper helped us to improve both the clarity and the technical content of the presentation. The NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many CPU-hours that were required for this study.

REFERENCES

(Note: References [25]–[27] are not mentioned in the text.)

1. AGRAWAL, R., CAREY, M., AND LIVNY, M. Models for studying concurrency control performance: Alternatives and implications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Austin, Tex., May 28–30). 1985.
2. BAYER, R., HELLER, H., AND REISER, A. Parallelism and recovery in database systems. *ACM Trans. Database Syst. 5*, 2 (June 1980), 139–156.
3. BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 2 (June 1981), 185–221.
4. BERNSTEIN, P. A., AND GOODMAN, N. Multiversion concurrency control—Theory and algorithms. *ACM Trans. Database Syst. 8*, 4 (Dec. 1983), 465–483.

5. BRYANT, R. SIMPAS—A simulation language based on PASCAL. Tech. Rep. 390, Computer Sciences Dept., Univ. of Wisconsin-Madison, June 1980.
6. CAREY, M. Modeling and evaluation of database concurrency control algorithms. Ph.D. dissertation, Computer Science Div. (EECS), Univ. of California, Berkeley, Aug. 1983.
7. CAREY, M. Multiple versions and the performance of optimistic concurrency control. Tech. Rep. 517, Computer Sciences Dept., Univ. of Wisconsin-Madison, Oct. 1983.
8. CAREY, M., AND STONEBRAKER, M. The performance of concurrency control algorithms for database management systems. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug.). VLDB Foundation, 1984.
9. CHAN, A., AND GRAY, R. Implementing distributed read-only transactions. *IEEE Trans. Softw. Eng. SE-11*, 2 (Feb. 1985).
10. CHAN, A., DAYAL, U., AND HSU, M. Providing database management capabilities for mission critical applications. Paper presented at the *International Workshop on High-Performance Transaction Processing Systems* (Asilomar, Calif., Sept.). IEEE, New York, 1985.
11. CHAN, A., FOX, S., LIN, W., NORI, A., AND RIES, D. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Orlando, Fla., June 2–4). ACM, New York, 1982.
12. DATE, C. *An Introduction to Database Systems* (Vol. II). Addison-Wesley, Reading, Mass., 1982.
13. DUBOURDIEU, D. Implementation of distributed transactions. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*. 1982.
14. GRAY, J. Notes on database operating systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, Eds. Springer-Verlag, New York, 1979.
15. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the system R database manager. *ACM Comput. Surv. 13*, 2 (June 1981), 223–242.
16. KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (June 1981), 213–226.
17. LAI, M., AND WILKINSON, W. Distributed transaction management in JASMIN. In *Proceedings of the 10th International Conference on Very Large Data Bases* (Singapore, Aug.). VLDB Foundation, 1984.
18. LIN, W., AND NOLTE, J. Basic timestamp, multiple version timestamp, and two-phase locking. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Italy). VLDB Foundation, 1983.
19. LIN, W., AND NOLTE, J. Performance of distributed concurrency control. In *Distributed Database Control and Allocation*. Final Tech. Rep., vol. 2, Computer Corporation of America, Cambridge, Mass., 1983.
20. LIVNY, M., KHOSHAFIAN, S., AND BORAL, H. Multi-disk management algorithms. Paper presented at the *International Workshop on High-Performance Transaction Processing Systems* (Asilomar, Calif., Sept.). 1985. (Also MCC Tech. Rep., Microelectronics and Computer Technology Corporation, Austin, Tex., Dec. 1985.)
21. PAPADIMITRIOU, C., AND KANELAKIS, P. On concurrency control by multiple versions. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 89–99.
22. PEINL, P., AND REUTER, A. Empirical comparison of database concurrency control schemes. In *Proceedings of the 9th International Conference on Very Large Data Bases* (Florence, Italy). VLDB Foundation, 1983.
23. REED, D. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., 1978.
24. REED, D. P. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Sys. 1*, 1 (Feb. 1983), 3–23.
25. RIES, D. The effects of concurrency control on database management system performance. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, 1979.
26. RIES, D. R., AND STONEBRAKER, M. Effects of locking granularity on database management system. *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 233–246.
27. RIES, D. R., AND STONEBRAKER, M. R. Locking granularity revisited. *ACM Trans. Database Syst. 4*, 2 (June 1979), 210–227.

28. ROBINSON, J. Design of concurrency controls for transaction processing systems. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1982.
29. ROOME, W. A content-addressable intelligent store. *Bell Syst. Tech. J. 61*, 9 (Nov. 1982).
30. SARGENT, R. Statistical analysis of simulation output data. In *Proceedings of the 4th Annual Symposium on the Simulation of Computer Systems.* National Bureau of Standards, Boulder, Colo., 1976.
31. STEARNS, R., AND ROSENKRANTZ, D. Distributed database concurrency controls using before-values. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Ann Arbor, Mich., Apr. 29–May 1). ACM, New York, 1981.
32. ULLMAN, J. *Principles of Database Systems*, 2nd ed. Computer Science Press, Rockville, Md., 1983.

Received August 1984; revised February 1986 and June 1986; accepted June 1986