10 VERSION MANAGEMENT IN THE STAR MLS DATABASE SYSTEM

Ramprasad Sripada and Thomas F. Keefe

Abstract: This paper describes version management in the Secure TransActional Resources - Database System (*-DBS) currently being developed at Penn State. This system employs concurrency control based on a secure multiversion timestamp ordering protocol. Efficient version management is critical to the performance of such a system. This paper describes a method of version management that requires no trust, adapts effectively to skewed access patterns, provides access to any version with at most one disk access and supports tuple level concurrency control. Based on our implementation, we report on the performance of this method.

10.1 INTRODUCTION

Multiversion databases are used as part of the design for Secure TransActional Resources-Database System (*-DBS) project currently being developed at Penn State. This paper addresses issues related to secure and efficient version management. The methods developed are implemented and the results and insights obtained are presented.

Secure version management methods often assign high-level transactions smaller timestamps than those of low-level transactions forcing them to access older versions. This bias can lead to performance penalties on high-level transaction. We attempt to improve performance of these transactions and at the same time improve performance for transactions that update or modify databases at their own security level. In this paper, we propose a dynamic on-page caching scheme and an in-memory version directory that reduces and

S. Jajodia (ed.), *Database Security XII*

improves the efficiency of I/O. The remainder of this section provides background in multilevel security and multiversioned databases.

10.1.1 Multilevel Security

A brief review of multilevel security (MLS) is presented here. An MLS policy consists of mandatory and discretionary portions. A mandatory security policy controls the flow of information based on the perceived trustworthiness of an individual while a discretionary security policy controls the flow of information based upon user identity. This paper considers mandatory security only. In systems enforcing multilevel security, objects represent elements of information and subjects represent active entities such as processes. Subjects and Objects are assigned security levels. The Bell-LaPadula model[3] provides a concrete method of enforcing mandatory access control policy. It defines allowable read and write accesses to data objects in the form of the *simple security* and **-property*[3]. The simple security requires that a subject be allowed to read an object only if the security level of the subject dominate that of the object. The *-property requires that a subject only be allowed to write objects with security levels dominating its own. In our work here, we restrict this further, and only a allow a subject to write objects at its own level. An implication of this is that transactions accessing a database at a lower security level appear to the lower database as a query.

10.1.2 Multiversion Databases

Versions are retained not for the sake of satisfying temporal queries but for concurrency purposes. This type of versioning is called transient versioning [5]. This means that at startup the database is single versioned. After recovery, the database is single-versioned once again. In a multiversioned system, transactions are assigned a timestamp value when they enter the system. Each version maintains both the timestamp of the transaction that created it and the maximum of the timestamps of all transactions that read it. These timestamp values are called the write and read timestamp respectively. When a query, wishes to access a record, it is provided with the version having the largest write timestamp less than or equal to its own.

Versions are created by update transactions. The update operation results in the creation of a new version of the tuple with the appropriate fields modified. The previous version is also maintained. Versions thus created can be chained together as shown in Figure 10.1. The primary version has the largest write timestamp within the version chain. As can be seen, besides data, a read and write timestamp and a pointer to the previous version are stored with each version. This is the overhead required to implement a multiversioned element. For effective version management, it is essential to limit the storage requirements as far as possible.

The remainder of this paper is organized as follows: First, related work is presented in Section 10.2. Section 10.3 presents a new type of on-page caching

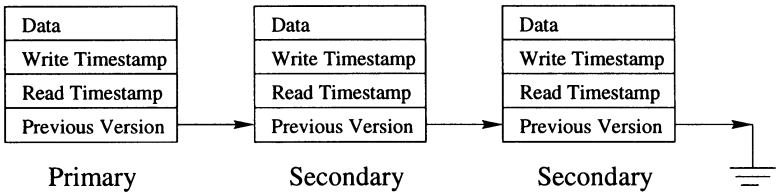

Figure 10.1 Version Chain

called dynamic on-page caching. Section 10.4 looks at efficient means for accessing a version and maintaining timestamp information necessary for concurrency control. Implementation issues are dealt with in Section 10.5. Experimental results obtained from the implementation are presented in the Section 10.6 followed by concluding remarks in Section 10.7.

10.2 RELATED WORK

Early multiversioned systems stored primary and secondary versions in two different database files [7]. The files containing the secondary versions is referred to as the version pool. However, in view of the inefficiencies arising from this storage arrangement, Bober and Carey suggested on-page caching [5]. In this approach, part of the version pool resides in the data pages of the main database itself.

On-page caching as suggested by [5] assigns a fixed portion of every data page to hold the cache. Consider for example, an update to a record. When this record is updated, the current primary version is copied into the cache before the new primary is created. If the cache is already full, garbage collection is attempted to remove versions in the cache that are no longer needed. Versions that would never be appropriate to transactions now or in the future can be deleted. If garbage collection is unsuccessful in freeing the required space, then a version in the on-page cache is chosen for replacement. This version is pushed to the version pool file thus creating space in the data page. Notice that a version is pushed when the on-page cache is full, not when the data page is full. One side effect of on-page caching is an improvement in data page utilization in the main database. This is because, more versions are made to reside in the main database itself using the space already available. The effect of on-page caches on performance is discussed in detail in [5].

Let us examine how a record is typically accessed using a B^+-tree. When a record has to be accessed by, say, a query operation, the location of the primary version is obtained from the leaf page in the B^+-tree. Then, starting from this primary version, the version chain is traversed until the appropriate version is found. Observe that retrieving each version in the chain can entail additional disk I/O. So, even after the leaf page is reached, retrieving the appropriate version might require several additional disk I/O operations. The access path to a version is determined by the storage organization of the database. On-page

caching tends to shorten the length of this path. For example, without on-page caching all secondary versions would reside in the version pool. However, even with on-page caching, overflow from on-page cache causes secondary versions to be pushed to the version pool. Alternative storage arrangements for faster access to a version were proposed in [6]. Three techniques were proposed and the performance of these techniques were evaluated using simulation. The method with the best overall performance was data page version selection (DP). In this approach all version information, including timestamp and version pointers are maintained along with the primary version in the same page in the data file. This ensures that any version can be accessed in at most two disk accesses. We propose to store this information in memory thereby reducing the number of disk accesses to one. In this regard, we assume a B^+-tree with clustering index.

A feasibility study of multiversioned databases enforcing MLS was reported in [15]. The focus was on mechanisms to provide efficient access to multiple versions of data. In this regard, the authors studied in detail the storage and access costs associated with multiversioning. An analytical performance model was developed to predict the penalty of retaining earlier versions for the sake of queries and the model was validated using measurements from an experimental prototype. However, the model did not address on-page caching. It was assumed that all secondary versions were maintained in the version pool. Both [15] and [7] assume versions at the granularity of pages. In this work, we adopt a tuple- level granularity and maintain version location information in memory.

10.3 DYNAMIC ON-PAGE CACHING

In our scheme, no space is dedicated to an on-page cache. The size of on-page cache is allowed to grow dynamically to accommodate the workload requirements. Versions are pushed to the version pool only when the data page is full as opposed to the on-page cache as recommended in [5]. However, we still retain a version pool that would be used when a data page becomes full. Whenever a page becomes full, we check if any of the versions can be collected. If not, then the oldest version is selected for replacement. We adopt a write-one policy, i.e., only one version is written to version pool each time. Note that writing one version at a time to a version pool does not lead to an I/O for every replacement as buffering can be used. For recovery purposes, however, we utilize a separate log file. This would store the updates to the databases before the transaction commits. This makes it unnecessary to flush the versions pool writes to disk before committing a transaction.

Dynamic on-page caching allows the benefits of on-page caching described above to be more fully utilized. Results in [5] suggest that queries execute faster as the size of the on-page cache is increased. By allowing this size to be determined dynamically, we can accommodate secondary versions more efficiently. This method also adapts well to nonuniform access patterns. For pages that see little or no update activity, the portion of the page that otherwise would be set aside for the cache is available for primary versions. However, when a page

is updated frequently it becomes a hotspot [11]. In this case, with a fixed cache size, cache overflow would occur frequently and the performance benefits of on-page caching are reduced. To address this problem, the size of the dynamic on-page cache is controlled dynamically based on the update frequency of the page.

One important side effect of a dynamic on-page caching scheme is the capability to tailor the cache size to meet the needs of a known work load. For example, when the workload is dominated by update operations, then the cache size will be adjusted to accommodate a sufficient number of secondary versions. When the workload is dominated by sequential scan queries, a smaller cache size will result. This will tend to preserve locality among the primary versions and allow these types of queries to complete faster. Databases inevitably experience non-uniform access patterns resulting in the creation of hot spots in certain regions of the database. An inability to adapt the cache size as required will tend to reduce the throughput of the database system. Fixed size on-page caches behave as if there is no cache at all once they become full. This is because, every update causes some version to be pushed to the version pool. Hotspots can provide sufficient update activity to fill an on-page cache and lead to reduced effectiveness. Dynamic modification of on-page cache size can adapt to such non-uniform access patterns. Below, we present a strategy for controlling on-page cache size to address this problem.

Hotspots are characterized by rapid version creation. We propose to use the following measure to characterize the intensity of update activity.

$$hot\_rate = \frac{num\_vers}{curr\_timestamp - ver\_timestamp}$$

In the definition, *num_vers* is the number of versions created, *curr_timestamp* is the current timestamp, and *ver_timestamp* is the time at which *num_vers* was last set to zero. Thus, this measure approximates the update rate for the page. We mitigate the effect of a hot spot by splitting the corresponding B^+-tree index page early. That means, a page that meets our criterion would be split even before it is full. This splitting is triggered when *hot_rate* reaches some predefined limit.

Splitting a page early causes updates to complete faster. This is because the updates are now distributed to the two pages that resulted from the split. This leads to less contention and thus higher throughput. Further, immediately after a page split, the amount of free space in the page increases to 50%. So, more versions can be accommodated. The positive effect of on-page caching on utilization of disk blocks is offset by splitting the page early. However, as we expect only a relatively small portion of the database will meet our criteria for a hot spot, this reduced utilization will only apply to a small portion of the database.

10.4 VERSION DIRECTORY

We propose storing the timestamp information and version pointers in memory. A similar idea for storing timestamp information in a single versioned system is proposed in [4]. After the tuple identifier for a key value is located using the index, we can use the in-memory structure to determine where the appropriate version resides with no additional I/O operations. In this scheme, at startup, the database is single versioned and all tuples have a default timestamp. At this moment, no timestamp information is required. Since all tuples have the same default timestamp, it need not be stored with each tuple. As updates and inserts occur, the version directory is used to store information about the version chains that are being formed. So, the version directory only needs to store information about version chains that do not have a default timestamp on the primary version. Thus, we assume that if information about a tuple is not available in the version directory, then it has a default timestamp. Note that the size of the version table is proportional to the number of active transactions and not the size of the database. This is because we only need to maintain information about versions that have been updated recently. From time to time, the default timestamp can be reset to a higher value. This allows us to collect some of the memory tied up in the table. When the default timestamp is changed, all versions with smaller timestamps can be removed from the version directory. We ensure that no active transaction exists with a timestamp below the default timestamp. Any such transactions are aborted. For more details refer to [14].

Storing version information in memory improves the performance of transactions at dominating security levels. We use a secure timestamp generator based on the protocol described in [10]. As explained earlier, due to the *-property [3], a database can only be queried by transactions at dominating security levels. Combined with our timestamp generation method this forces high-level transactions to access older versions. If the appropriate version for these queries resides in the version pool then it would require multiple disk I/O for retrieving that version. Using a version directory we can avoid this bias against high-level transactions and ensure that all versions can be accessed with at most one disk access.

We can think of the version directory as a more efficient method of storing and caching, in memory, the timestamp and version chain information. To see the advantage of this approach, consider the following example. Assume that tuples require, on average, 200 bytes of storage and that timestamps requires 16 bytes each. The two timestamps associated with each tuple amounts to an overhead of about 15%. Thus, each page in the buffer pool is only 85% effective. This is especially troubling when we realize that the majority of the timestamps are old enough to be replaced by a single default value. Thus by maintaining only those timestamps that are actually necessary, we reduce this overhead considerably. This leads to higher effective I/O rates, and a better use of memory.

A hash table is used for storing the version information. Hashing is done in such a way that all tuples lying in the same page would have their information

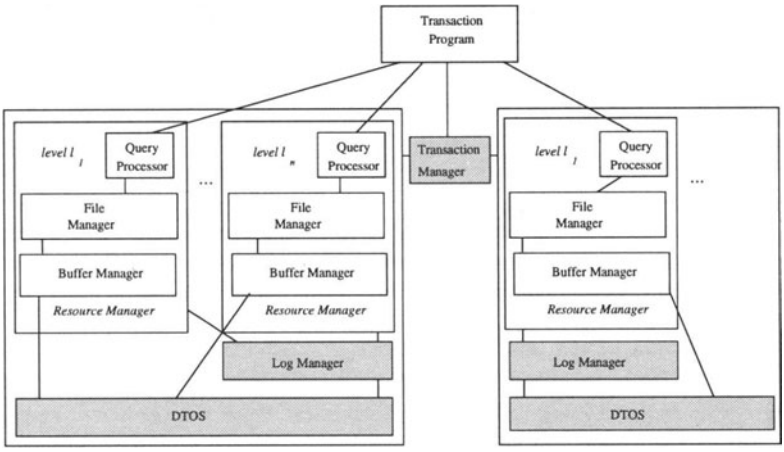

Figure 10.2 Architecture of Star-DBS

stored in the same hash chain. During garbage collection, versions in a page can be collected by looking at one hash chain.

10.5 IMPLEMENTATION ASPECTS

The design discussed above was implemented as part of the *-DBS project. The prototype is hosted on Distributed Trusted Operating System (DTOS) [12]. DTOS is an experimental prototype operating system developed at Secure Computing Corporation. It provides mechanisms to implement multilevel security on the CMU Mach Microkernel [1] [8]) and provides policy-based control over all Mach services.

Figure 10.2 shows the architecure of the Star-DBS prototype. The trusted components are shown shaded. The prototype adopts a client/server architecture. A transaction executing at a client begins by contacting a transaction manager (TM) which assigns it a timestamp. For details on the protocol governing secure timestamp generation refer to [10]. The TM provides timestamps to transactions at all security levels. The transaction then proceeds to make service requests to one or more resource managers (RM). When the transaction is complete, it contacts the transaction manager again to request that its work be committed.

Each RM is implemented as an untrusted subject performing operations on behalf of clients at a single level. The RM implements a restricted SQL-like RPC level interface (i.e., no nested queries, no aggregates, no sortby, and no support for groups). The RM makes pin/unpin requests to the buffer manager (BM) [2]. The buffer manager controls the movement of data between the persistent and volatile portions of the database for all security levels. It also coordinates logging with page flushes to enforce the write ahead logging (WAL) protocol[9]. Each RM is multithreaded allowing it to service requests from mul-

tiple clients concurrently. The log manager [13] writes uninterpreted undo/redo records to the log on behalf of RMs, writes commit and abort records on behalf of the TM and controls the flushing of log records to disk.

The designs described in this paper were implemented on the DTOS operating system and consists of approximately 5000 lines written in C. The role of this implementation is to act as a file manager within the architecture shown in Figure 10.2.

One of the important implementation challenges was the version table. The version directory is organized as a hash table. Each security level maintains an independent version directory for the versions residing at its security level. This version information is accessed by all transactions at the same security level as well as by transactions at dominating security levels. Thus, this involves realizing a logically single version directory with independent version directories at each security level. In our prototype, mandatory access control is enforced by a trusted component of the buffer manager. We utilize this component to allow high-level transactions to access version directories at lower levels. Each RM creates a file that holds the version directory for that level. Transactions at higher level thus retrieve the buffer containing the entry they wish to access. In case, a subject at the lower security level tries to pin this page in write mode, the page is copied to another buffer [2]. Retrieval and traversal through the hash list are done transparently through an interface implemented within the RM. This interface abstracts away the security related issues and provides functionality allowing all standard operations on a hash table. So, RMs need not explicitly do anything special to retrieve an entry in the version directory, even if it resides in another security level.

Our RM is implemented as an untrusted subject. A single version directory for all the RMs at different security levels would have required a trusted implementation. An untrusted implementation eliminates the need for formal security evaluation of the component and allows simpler prototyping.

10.6 EXPERIMENTAL RESULTS

The implementation provided a means to test the feasibility and performance of the ideas we developed. We were interested in the effect of not storing version information in stable storage on performance.

The first test conducted was to observe how the size of the on-page cache would vary if no limit was placed on its size. In particular we wanted to observe the variance of on-page cache size. A wide variance in on-page cache size across the database suggests that dynamic control will be effective. For this purpose, a database was populated with tuples whose key values were generated randomly with uniform distribution from the set 1, ..., 100,000. Then, tuples were chosen following a uniform distribution from this set for update. Selection is done with replacement so that one tuple may be updated several times during an experiment. On average, for every ten tuples in the database, one update operation was applied. This means the average size of a version chain is 1.1 versions. This value was motivated by results of a performance study described

in [5]. Page size for the database was 4096 bytes. Tuple size was chosen to allow 30 tuples per page. Tuples were inserted until the database consisted of about 100 data pages. At the end of all insert and update operations, the number of primary and secondary versions in each data page was measured.

The distribution of on-page cache size is given by the histogram shown in Figure 10.3. On the x-axis is shown the size of the on-page cache as a percentage of the page size. The percentage of pages that have a particular cache size is shown on the y-axis. We repeated the experiment five times. For each experiment we compute the mean, i.e., $\overline{x_1}$, $\overline{x_2}$, ..., $\overline{x_5}$. We then calculate the mean and standard deviation for this collection of five samples. Assuming a normal distribution, we calculated 90% confidence intervals for the sample mean using the formula:

$$\left(\overline{X} - \frac{1.64\sigma}{\sqrt{n}}, \overline{X} + \frac{1.64\sigma}{\sqrt{n}}\right)$$

In the expression $\overline{X}$ represents the mean of the five sample means, σ represents their standard deviation and n represents the number of measurements (i.e., five). The mean cache size is 8.816% with a 90% confidence interval of (8.415, 9.218).

As can be seen the size of on-page cache in each page varies widely. This significant variation in the size of on-page cache makes it extremely difficult to predefine a particular size for the on-page cache. Also, a significant portion of the database is populated with pages which have no secondary versions at all. This is indicated by the number of data pages with zero on-page cache size. This shows that a significant number of elements experienced no updates at all and thus validates our assumptions that storing timestamps in the data page is not efficient.

Another test was devised to observe the savings in disk I/O due to the version table. The scheme we compare our savings against is Data Page scheme (DP) [6]. In this method, all of the version information is stored with the primary version. So, the number of disk I/Os required to retrieve the primary version would be one, and overall, the number of accesses needed to retrieve any version in the version chain would not be more than two. A database was created as discussed above. However, to remove the effects of dynamic on-page caching on the version table we set the maximum size of the on-page cache at 10% of the data page size. The same set of key values were used in both cases. Then, a query is run to execute a table scan over the tuples in the database. The disk I/O required when the version directory was used is measured and the disk I/O with DP is also measured. The mean savings obtained are 7.556% with a 90% confidence interval of (6.920, 8.192). With dynamic on-page caching enabled even more versions would reside in the data page itself and this would help to increase the savings in disk I/O.

To examine the combined effect of dynamic on-page caching and the version table, tests were conducted using a non-uniform access pattern. We characterize the amount of nonuniformity or access skew as x%, implying that x% of access

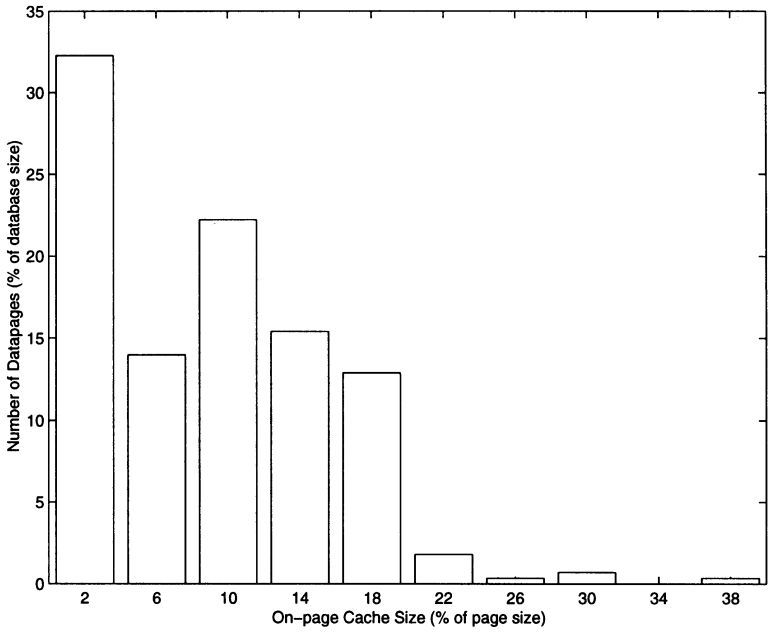

Figure 10.3 Distribution of On-page Cache Size

requests are directed to $100-x\%$ of the data elements in the database [11]. The database is divided into two parts, the first constitutes $x\%$ of the data items and the second represents the reminder $(100 - x\%)$. With probability $\frac{100-x}{100}$ a transaction accesses the first part. An element of this set is chosen based on a uniform distribution. Thus, for a 70% Skew, 70% of the accesses are to 30% of the data elements. Our tests ranged from a uniform distribution (a skew of 50%) to a 90% skew.

So, for each level of skew considered, we measured the disk I/O that was saved by a query scanning the entire relation. Again, a database was created as described above. Only the updates to the database were skewed. Then a query was run in isolation when all the update and insert activity in the database was complete. The timestamp of this query is between the timestamps selected for updates and inserts. This means, if no update was applied to a version then the primary version is the appropriate version for this query. If an update operation was applied, then the secondary version is the appropriate version. The results obtained are shown in Figure 10.4. The difference between the disk I/O required for a query with a fixed on-page cache size of 10% using DP, and the disk I/O for our design is expressed as a percentage plotted on the y-axis. For each access skew, the test was repeated five times and the average and variance are shown on the plot. The error bars represent 90% confidence intervals.

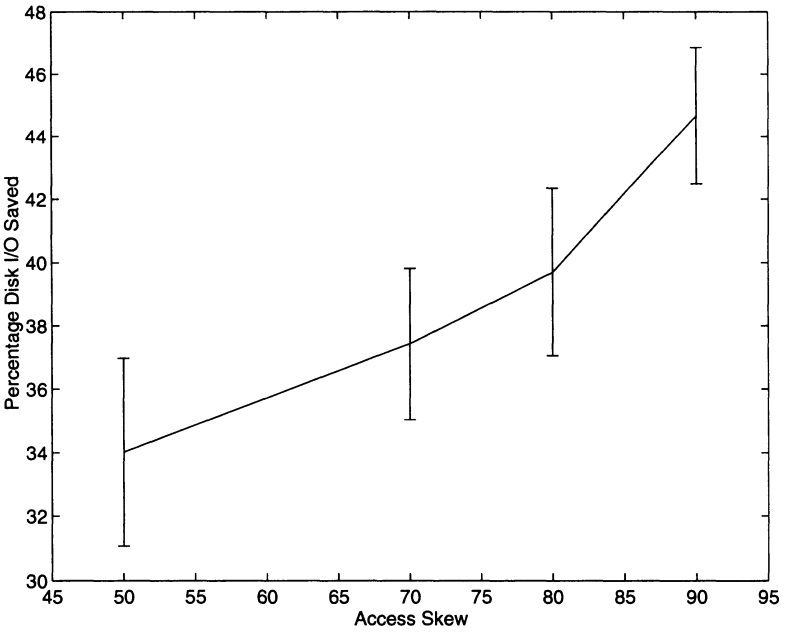

Figure 10.4 Disk I/O Reduction as a function of Access Skew

As can be seen, as the amount of skew increases, the disk I/O saved also increases. This is because, due to the formation of hotspots, the pages are split earlier. This leads to more versions being held within the data page and hence a saving in disk I/O. Also, savings accrue due to the version directory that reduces disk I/O required when the appropriate version for the query resides in the version pool. As noted earlier, in a hot spot, the performance of fixed size on-page caches degrades. This effect becomes more pronounced as access skew is increased. This is because more versions are pushed to version pool. As disk I/O for retrieving versions in version pool is reduced with our scheme, the corresponding savings in disk I/O increases.

10.7 CONCLUDING REMARKS

We have presented a design for version management in a multilevel secure database system and described a prototype based on this design. In addressing the issues relating to storage of versions we found that a dynamic on-page caching scheme can effectively adapt to non-uniform access patterns. The version directory improves performance by reducing the overhead of maintaining version information. However, the version directory is constructed from a set of independent version directories each associated with a RM at that security level. This was done with support from DTOS and the buffer manager.

The combined effects of dynamic on-page caching and the version table show a reduction in I/O of between 32 and 47% over the DP method of [6].

References

[1] Mike Accetta, William Bolosky Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. *Proceedings of the Summer 1986 USENIX Conference*, Summer 1986.

[2] Ashwin Baskaran. Buffer management for a multilevel secure dbms. Master's thesis, Dept. of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, December 1996.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and multics interpretations. Technical Report MTR-2997, Mitre Corp., March 1976.

[4] P.A. Bernstein, V. Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[5] Paul Bober and Michael Carey. On mixing queries and transactions via multiversion locking. *Proceedings of the Eight IEEE Data Engineering Conference*, 1992.

[6] Paul Bober and Michael Carey. Indexing alternatives for multiversion locking. Technical Report 1184, Dept. of Computer Science, University of Wisconsin-Madison, November 1993.

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

[8] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. *Proceedings of the USENIX Conference*, Summer 1990.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[10] T. F. Keefe and W. T. Tsai. Multiversion transaction scheduler for centralized multilevel secure database systems. Technical Report TR-93-116, Department of Computer Science and Engineering, The Pennsylvania State University, January 1993.

[11] Vijay Kumar, editor. *Performance of Concurrency Control mechanisms in Centralized Database Dystems.* Prentice Hall, Englewood Cliffs, NJ, 1996.

[12] Spencer E. Minear. Providing policy control over object operations in a mach based system. *Proceedings of USENIX Conference*, pages 1–15, April 1995.

[13] V. R. Pesati, T. F. Keefe, and S. Pal. The design and implementation of a multilevel secure log manager. *Proceedings of the IEEE Symposium on Security and Privacy*, page 55 64, May 1997.

[14] Ramprasad Sripada. The design of a multiversion database file manager for multilevel secure systems. Master's thesis, Deptartment of Computer Science and Engineering, The Pennsylvania State University, August 1997.

[15] A. C. Warner and T. F. Keefe. Version pool management in a multilevel secure multiversion transaction manager. *Proceedings of IEEE Symposium on Research in Security and Privacy*, May 1995.