Concurrency Control with Lock Preemption and Restoration

Gopi K. Attaluri Jacob Slonim* Per-Åke Larson
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3V5
E-mail: {gkattalu,jslonim,palarson}@bluebox.uwaterloo.ca

Abstract

Lock preemption is a common problem with two-phase locking. While a transaction is still active, its locks may be preempted to avoid a deadlock or to satisfy the lock request of a higher-priority transaction. In general, the victim of preemption is aborted. This paper suggests that the victim could be continued and its work up to preemption could be saved if the preempted lock is restored on its availability. Object-oriented database management systems based on client-server architecture, multidatabase systems, and relational database management systems using page-locks are a few cases where lock restoration can be beneficial. Lock restoration works well when the granularity of locking is larger than the granularity of database operations, namely, reads and writes.

This paper proposes a correctness criteria called value-serializability for isolation in histories generated by a lock restoring concurrency control scheme. A value-serializable history is shown to have the same behavior (in terms of values read and written) as a conflict-serializable history. The paper describes concurrency control scheme based on lock restoration, and proves that the scheme generates value-serializable histories in general and conflict-serializable histories for a certain class of preempted locks. Some important advantages and disadvantages of lock restoration are listed at the end.

Keywords: Transaction management, Concurrency control, Locking, Non-traditional database applications.

*Head of Research, Centre for Advanced Studies, IBM Toronto Lab, Toronto.

1 Introduction

Locking, in particular two-phase locking, is the commonly used concurrency control scheme in database management systems (DBMSs). A transaction locks a *data item*, the unit of locking, before reading/writing a part or the full data item. It releases the locks at termination (commit or abort). Read locks could be released earlier, after the transaction has obtained all the necessary locks.

Occasionally, the transaction manager forcibly takes away (*preempts*) locks before the transaction could release them. There are several reasons for preemption. Locks of one or more transactions may be preempted to break a deadlock of blocked lock requests. When a transaction crosses a timeout period designed to detect deadlocks or communication failures, its locks may be preempted. In a prioritized system (for example, VERSANT [1]), a low-priority lock may be preempted to satisfy a conflicting high-priority lock request. Finally, locks of unprepared, non-terminated transactions are preempted at site and system failures.

The victim of preemption, henceforth called the *victim*, can be handled in two different ways: (i) it can be aborted and restarted, or (ii) it can be continued if the preempted lock can be reallocated once it is available. In the case of site and system failures, the victim must be aborted because the volatile memory reflecting the victim's operations would have been erased, and locks could not be restored. The victim may be continued in other cases. Reallocation of preempted locks is non-trivial; both the preemption and reallocation must be transparent to the victim. Otherwise, the victim would violate the two-phase locking principle — it would lock, unlock and and again lock the same data item. If other transactions perform conflicting operations on the data

item between the preemption and reallocation, a conflict-non-serializable transaction history could result. This paper deals with transparent preemption and reallocation of locks. The process is referred to as *lock restoration* or simply *restoration*.

Commonly, the transaction manager aborts the victim rather than restore its preempted lock. This is perhaps for two reasons. Firstly, traditional transactions have been simple and short. It is simple to and more efficient to abort the victim and redo it than to try and continue it. Secondly, other transactions could lock the data item and read/write it between the preemption and restoration of the lock. If the victim has read (modified) the data before preemption, its read (modification) could be invalidated by other transactions. The validity of the victim's operation is difficult to determine when the lock becomes available.

This paper examines restoration as a solution to lock preemption. It paper discusses the relevance of lock restoration to current and future database systems and examines important theoretical and practical aspects of lock restoration. Section 2 argues that restoration can be useful in current and future transaction processing systems. It proposes a correctness criteria for histories realized by schemes using restoration. Section 3 describes a concurrency control scheme that supports restoration. The scheme is shown to satisfy the proposed correctness criteria. Section 4 discusses the pros and cons of restoration and identifies a few potential application areas. Finally, Section 5 provides the conclusions.

2 Motivation and Theory

This section looks at the issues in lock restoration and develops the theory for correctness (from isolation point of view) of histories generated when locks are restored. In the rest of the paper, correctness of a history refers to the correctness for transaction isolation (or simply isolation) in the history.

2.1 Why Preempt Locks

In case of a deadlock cycle, locks of one or more members must be preempted to resolve the deadlock. In case of a site failure, all active transactions are aborted and their locks are effectively preempted because it is possible to continue the transaction easily after the site recovery. In the above two cases, lock preemption is necessary. However, in a prioritized DBMS such as VERSANT lock preemption is desirable to avoid blocking a high priority transaction.

Lock preemption in a multidatabase system [2] is particularly interesting - regardless of the cause for preemption (site failure, communication failure, deadlock or timeout), restoration can always be attempted. A *multidatabase* system is a collection of autonomous, heterogeneous, preexisting local DBMSs. It supports global transactions to access data located in multiple local DBMSs. A global transaction is divided into multiple subtransactions, each subtransaction accessing the data at a local DBMS. A local DBMS, being autonomous, can unilaterally abort a subtransaction for any reason. For the global transaction, such an abort is equivalent to preempting all its locks at that DBMS.

2.2 Why Restore Locks

Traditionally, DBMSs viewed data items to be small and uniformly structured and transactions to be simple, non-interactive and short. This view is supported by traditional applications such as banking and airline ticketing. Current database research addresses the requirements for non-traditional applications such as Computer-Aided Design (CAD), Computer-Aided Engineering (CAE), video and audio image processing, and geographic information processing. These applications deal with large, non-uniform data items. Transactions are complex, interactive and possibly long. Current transaction processing systems include extended relational systems (with BLOBs, extended SQL, abstract data types, etc.) and object-oriented database systems (supporting classes, class hierarchy, complex objects, class inheritance and so on). Several extensions to the classical transaction model are proposed to deal with the complexity and longevity of transactions. Perhaps due to the simplicity of the classical transaction model or the lack of a widely accepted "extended transaction model", the classical transaction is still the commonly supported model. And, (two-phase) locking is commonly used for concurrency control.

Lock restoration could be beneficial in these systems for the following reasons:

- If the victim is aborted and redone, the work done by the victim before the preemption is wasted. With long transactions, that work

can be significant enough that one should try restoration instead.

- In object-oriented DBMSs (OODBMSs) based on client-server architecture, a disk page is often the unit of locking. One or more objects are stored on a page, and a transaction must lock the page to read/write an object. A preempted lock can be restored unless other transactions have modified the same object read by the victim. A page often contains multiple objects; therefore, often, the page lock may be preempted for other transactions accessing objects different from those accessed by the victim.

  The above argument also applies to non-OODBMSs when the granularity of concurrency control is larger than the granularity of read/write operations. An example is a relational DBMS that locks pages containing multiple tuples and allows a transaction to read/write tuples individually.

- In distributed DBMSs and multidatabase systems, a transaction is made up of multiple subtransactions that are executed at different sites. The costs of inter-site communication and inter-site transaction processing (more so for a multidatabase system) result in long transactions. If a subtransaction is aborted, rather than abort the entire transaction, one can "reinstate" the subtransaction by restoring its locks. If restoration is successful, the work of the entire transaction would be saved.

2.3 Terminology

This section illustrates the important issues in restoration through various examples. First, the definition of a few key terms is in order.

Definition 2.1 A *data item* is the unit of concurrency control (locking).

An example data item is a page, relation, tuple or field.

Definition 2.2 A *data object* is the unit of data accessed by a read (write) operation.

Tuples and fields are examples of data objects.

Definition 2.3 A *transaction* is a sequence of read or write operations on data objects ended with a commit or abort.

Definition 2.4 Two write operations, or a read and a write operation, are said to *conflict* with one another if they access overlapping data objects.

A conflict indicates that the execution order of the two operations determines behavior of the transactions and the database.

Definition 2.5 Given a set of transactions, a *history* is a partial order of their operations. History is a formal representation of the order of execution of operations by the DBMS. Two conflicting operations must be sequentially ordered in a history.

Definition 2.6 The *committed projection* of a history $\mathbf{H}$, denoted $C(\mathbf{H})$, contains only the operations of the committed transactions of $\mathbf{H}$.

The committed projection is important and determines the "correctness" of a history.

Definition 2.7 In a *serial history*, the operations of two transactions do not interleave. A history is *conflict-serializable* if its committed projection is *conflict-equivalent* to a serial history. That is, the committed projection has the same transactions as a serial history and orders the conflicting operations in the same order as the serial history.

Serialness provides the best form of correctness for isolation in the history. However, a serial history excludes concurrency, so conflict-serializability, which allows concurrency, is accepted as a valid correctness criteria. It is a well-known result that a history is conflict-serializable if and only if a directed graph representing the conflicts of committed transactions, known as serialization graph, is acyclic.

The above definitions of transactions and histories are a simplified adaptation from [3].

2.4 Issues in Lock Restoration

The following examples assume that the transaction manager uses two-phase locking with pages as data items and tuples as data objects. The focus is on managing the victim; we are not concerned with the reasons for preemption. The examples deal with three transactions T_1, T_2 and T_3 that access four tuples o_1, o_2, o_3 and o_4. The terms $r(x)$ and $w(x)$ denote read and write operations on tuple x. The subscript, if present, identifies the transaction the operation is part of:

$$\begin{array}{ll} T_1: & r(o_1)\ w(o_2)\ w(o_4) \\ T_2: & r(o_1)\ r(o_2)\ w(o_4) \\ T_3: & r(o_1)\ w(o_3)\ w(o_4) \end{array}$$

Tuples o_1 and o_4 are located on different pages. o_2 and o_3 are located on a page that is different from those of o_1 and o_4. Being located on the same page, tuples o_2 and o_3 cannot be concurrently locked/accessed for conflicting operations by two transactions.

Lock and unlock operations on pages are denoted by the terms lw, lr, uw and ur. $lw_1(o_1)$ means that the page containing o_1 is write-locked for T_1. $uw_1(o_1)$ means that a write lock of T_1 on the page containing o_1 has been released/preempted. Lock and unlock operations are not part of the history and are shown only to highlight lock preemption and restoration. Commit and abort operations, represented by $c(T)$ and $a(T)$, automatically release the locks owned by transaction T; unlock operations are not explicitly shown in their cases.

Example 2.1 Consider the history:

$$\begin{array}{ll} \mathbf{H}_1: & lr_2(o_1)\ r_2(o_1)\ lr_3(o_1)\ r_3(o_1) \\ & lr_2(o_2)\ r_2(o_2)\ ur_2(o_2)\ a_2 \\ & lw_3(o_3)\ w_3(o_3)\ lw_3(o_4)\ w_3(o_4)\ c_3 \end{array}$$

Because o_2 and o_3 are on the same page, T_2's page lock for o_2 is preempted[1] to accommodate T_3. T_2 is aborted because of the preemption, and $\mathbf{H}_1$ contains a single committed transaction T_3. Though tuples o_2 and o_3 are on the same page, their operations cannot conflict. When the page lock for o_2 is available after T_3's commit, it could be restored to T_2:

$$\begin{array}{ll} \mathbf{H}_1^{'}: & lr_2(o_1)\ r_2(o_1)\ lr_3(o_1)\ r_3(o_1) \\ & lr_2(o_2)\ r_2(o_2)\ ur_2(o_2) \\ & lw_3(o_3)\ w_3(o_3)\ lw_3(o_4)\ w_3(o_4)\ c_3 \\ & lr_2(o_2)\ lw_2(o_4)\ w_2(o_4)\ c_2 \end{array}$$

$\mathbf{H}_1^{'}$ has two committed transactions serialized in the order $T_3 \rightarrow T_2$.

Example 2.1 shows that restoration depends on what is accessed by the victim before preemption. Restoration can be guaranteed if the victim has modified but not read the tuple, as shown by Example 2.2.

Example 2.2 Consider the history:

[1] That is, the page lock is not released by T_2 but is forcibly taken away by the transaction manager.

$$\begin{array}{ll} \mathbf{H}_2: & lr_1(o_1)\ r_1(o_1)\ lr_2(o_1)\ r_2(o_1) \\ & lw_1(o_2)\ w_1(o_2)\ uw_1(o_2)\ a_1 \\ & lr_2(o_2)\ r_2(o_2)\ lw_2(o_4)\ w_2(o_4)\ c_2 \end{array}$$

T_1 is the victim here. To accommodate T_2, the modifications from $w_1(o_2)$ are discarded and the page lock for o_2 is preempted. T_1 is aborted and T_2 is the only committed transaction in the history.

Because o_2 is cleared of the modifications by $w_1(o_2)$, neither T_2 nor any other transaction accessing o_2 after preemption could have "read o_2 from T_1". The page lock for o_2 is available after T_2's commit and could be restored to T_1:

$$\begin{array}{ll} \mathbf{H}_2^{'}: & lr_1(o_1)\ r_1(o_1)\ lr_2(o_1)\ r_2(o_1) \\ & lw_1(o_2)\ w_1(o_2)\ uw_1(o_2) \\ & lr_2(o_2)\ r_2(o_2)\ lw_2(o_4)\ w_2(o_4)\ c_2 \\ & \underline{lw_1(o_2)\ w_1(o_2)}\ lw_1(o_4)\ w_1(o_4)\ c_1 \end{array}$$

$\mathbf{H}_2^{'}$ has two committed transactions serialized in the order $T_2 \rightarrow T_1$. The restored lock and the modifications from $w_2(o_2)$ are shown underlined. They are carried out by the transaction manager as part the restoration. T_1 is not aware of the restoration unless it issues an operation on o_2 between the preemption and restoration.

The following example shows that restoration may not be possible if the victim has read a tuple in the page before the page lock is preempted.

Example 2.3 Consider the history:

$$\begin{array}{ll} \mathbf{H}_3: & lr_2(o_1)\ r_2(o_1)\ lr_1(o_1)\ r_1(o_1) \\ & lr_3(o_1)\ r_3(o_1)\ lr_2(o_2)\ r_2(o_2)\ ur_2(o_2)\ a_2 \\ & lw_1(o_2)\ w_1(o_2)\ lw_1(o_4)\ w_1(o_4)\ c_1 \\ & lw_3(o_3)\ w_3(o_3)\ lw_3(o_4)\ w_3(o_4)\ c_3 \end{array}$$

Now, T_2's page lock for o_2 is preempted to accommodate first T_1 and then T_3. T_2 is aborted and the history has two committed transactions serialized in the order $T_1 \rightarrow T_3$.

The page lock for o_2 is available after T_2's commit, but restoration appears unsuccessful:

$$\begin{array}{ll} \mathbf{H}_3^{'}: & lr_2(o_1)\ r_2(o_1)\ lr_1(o_1)\ r_1(o_1) \\ & lr_3(o_1)\ r_3(o_1)\ lr_2(o_2)\ r_2(o_2)\ ur_2(o_2) \\ & lw_1(o_2)\ w_1(o_2)\ lw_1(o_4)\ w_1(o_4)\ c_1 \\ & lw_3(o_3)\ w_3(o_3)\ lw_3(o_4)\ w_3(o_4)\ c_3 \\ & \underline{lr_2(o_2)}\ lw_2(o_4)\ w_2(o_4)\ c_2 \end{array}$$

$\mathbf{H}_3^{'}$ is conflict-non-serializable as evidenced by a cycle of conflicts among committed transactions:

$$T_2 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2$$

Assume that the transaction manager records the committed value of the tuple read by the victim before preemption and compares it against the committed value of the tuple when the lock is restored. If the two values are identical (if the tuple has been modified by intervening transactions that committed, the last write operation from these transactions wrote the same value as read by the victim), restoration is possible as illustrated in Example 2.4.

Example 2.4 Reconsider history $\mathbf{H}_3^{'}$ of Example 2.3, where the page lock for o_2 is restored. If the value of o_2 before preemption and after restoration are the same, the value of o_2 read by $r_2(o_2)$ is the same as that from $w_1(o_2)$. Then, $r_2(o_2)$ is equivalent to have been executed after restoration; $\mathbf{H}_3^{'}$ can be "reduced" to a conflict-serializable history $\mathbf{H}_3^{''}$:

$$\begin{array}{ll} \mathbf{H}_3^{''}: & lr_2(o_1)\ r_2(o_1)\ lr_1(o_1)\ r_1(o_1) \\ & lr_3(o_1)\ r_3(o_1)\ lr_2(o_2)\ ur_2(o_2) \\ & lw_1(o_2)\ w_1(o_2)\ lw_1(o_4)\ w_1(o_4)\ c_1 \\ & lw_3(o_3)\ w_3(o_3)\ lw_3(o_4)\ w_3(o_4)\ c_3 \\ & \underline{lr_2(o_2)\ r_2(o_2)}\ lw_2(o_4)\ w_2(o_4)\ c_2 \end{array}$$

$\mathbf{H}_1^{''}$ is now conflict-serializable with three committed transactions in the order:

$$T_1 \rightarrow T_3 \rightarrow T_2$$

Based on the above examples, the issues in restoration are summarized below. Let o be a data item whose lock is preempted from and then restored to T_i. Assume that o_r and o_w are the parts of o read and written by T_i before lock preemption. o_r (o_w) is null if T has not read (written) any part of o. o_r is also null if T has only read the part of o that it has previously written to[2]

- *Neither o_r nor o_w has been accessed by a committed transaction between the preemption and restoration.*

  Clearly, no conflicts are introduced due to lock preemption and restoration; conflict-serializability of the history could not be violated.

- *Both o_r and o_w are null.*

  This is a special case of the above; the history must be conflict-serializable.

- *o_r is null and o_w is non-null. T's writes to o_w are discarded by the transaction manager before allocating the preempted lock to another transaction.*

  No other transaction could "read o_w from" T. So, no additional conflicts are introduced due to restoration; the conflict-serializability of the history could not be violated.

- *o_r is non-null. (If o_w is non-null, T's writes on it are discarded by the transaction manager before allocating the preempted lock to another transaction.)*

  Here, additional read-write conflicts are introduced by any conflicting writes on o_r between the preemption and restoration. The resulting history may not be conflict-serializable.

  If the value of o_r read by T is equivalent to the value of o_r at restoration, T can be deemed to have read o_r after the restoration, and the history can be "reduced" to an "equivalent" conflict-serializable history.

2.5 Correctness Theory for Restoration-based Histories

This Subsection formalizes the reduction mentioned above, and defines a correctness criteria for histories when locks are restored.

Definition 2.8 A history $\mathbf{H}_j$ is said to be a *value-reduction* of another history $\mathbf{H}_i$ if it satisfies the following conditions:

- The set of transactions is identical to that of $\mathbf{H}_i$.
- The ordering of operations is the same as that of $\mathbf{H}_i$, except that a read operation $r_i(x)$ is repositioned as follows:
  - The order of $r_i(x)$ with respect to other operations of T_i, the transaction it is part of, is unchanged from $\mathbf{H}_i$.
  - In the new position, $r_i(x)$ returns the same value as in $\mathbf{H}_i$. That is, the write operation(s) that $r_i(x)$ "reads from" provide the same value as in $\mathbf{H}_i$

Due to the repositioning of a read, the order of conflicting operations may be different in $\mathbf{H}_i$ and $\mathbf{H}_j$. The repositioned read cannot affect the values read and written by other transactions. Because it returns the same value in $\mathbf{H}_i$ and $\mathbf{H}_j$,

[2] In other words, o_r is null if T has not read o from some other transaction.

the corresponding transaction must also read and write the same values in $\mathbf{H}_i$ and $\mathbf{H}_j$. Therefore, each transaction must read and write the same values in $\mathbf{H}_i$ and $\mathbf{H}_j$.

If $\mathbf{H}_j$ is a value-reduction of $\mathbf{H}_i$, we say that $\mathbf{H}_i$ can be value-reduced to $\mathbf{H}_j$. Value-reduction is a symmetric relation — if $\mathbf{H}_i$ can be value-reduced to $\mathbf{H}_j$, the converse also holds. A history can have multiple value-reductions.

Example 2.5 Consider the operation $r_2(o_2)$ in the following histories involving transactions T_1, T_2 and T_3:

$$
\begin{array}{ll}
T_1: & w(o_1)\ w(o_2) \\
T_2: & r(o_2)\ w(o_3) \\
T_3: & r(o_3)\ w(o_2)
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{H}_1: & w_1(o_1)\ w_1(o_2)\ r_2(o_2)\ r_3(o_3) \\
 & w_3(o_2)\ w_2(o_3)\ c_1\ c_2\ c_3 \\
\mathbf{H}_2: & w_1(o_1)\ w_1(o_2)\ r_3(o_3)\ \underline{r_2(o_2)} \\
 & w_3(o_2)\ w_2(o_3)\ c_1\ c_2\ c_3 \\
\mathbf{H}_3: & w_1(o_1)\ w_1(o_2)\ r_3(o_3)\ w_3(o_2) \\
 & \underline{r_2(o_2)}\ w_2(o_3)\ c_1\ c_2\ c_3 \\
\mathbf{H}_4: & w_1(o_1)\ w_1(o_2)\ \underline{w_2(o_3)} \\
 & r_3(o_3)\ w_3(o_2)\ \underline{r_2(o_2)}\ c_1\ c_2\ c_3
\end{array}
$$

$\mathbf{H}_2$ is a trivial value-reduction of $\mathbf{H}_1$. The order of $r_2(o_2)$ respect to $r_3(o_3)$ has been changed. T_1, T_2 and T_3 behave identically in $\mathbf{H}_1$ and $\mathbf{H}_2$. $\mathbf{H}_3$ has changed the order of $r_2(o_2)$ with respect to both $r_2(o_2)$ and $w_3(o_2)$. $\mathbf{H}_3$ can be a value-reduction of $\mathbf{H}_1$ if and only if $w_1(o_2)$ and $w_3(o_2)$ write the same value to o_2. Finally, $\mathbf{H}_4$ has changed the order of $w_2(o_3)$ and $r_2(o_2)$ with respect to other operations. It cannot be a value-reduction of $\mathbf{H}_1$.

Definition 2.9 A history $\mathbf{H}_i$ is said to be *value-equivalent* to another history $\mathbf{H}_j$ if it can be obtained from $\mathbf{H}_j$ through zero or more value-reductions.

Value-equivalence is a reflexive, symmetric and transitive relationship among histories. From the property of value-reduction discussed above, it follows that if $\mathbf{H}_i$ is value-equivalent to $\mathbf{H}_j$, the transactions read and write the same values in both the histories.

Definition 2.10 A history $\mathbf{H}_i$ is *value-serializable* if its committed projection, $C(\mathbf{H}_i)$, is value-equivalent to some conflict-serializable history $\mathbf{H}_j$.

In the Example 2.5, $\mathbf{H}_1$ is a conflict-serializable history, while $\mathbf{H}_3$ is not. However, $\mathbf{H}_3$, whose committed projection is itself, is value-equivalent to $\mathbf{H}_1$, and hence value-serializable.

We propose value-serializability as the correctness criteria for isolation in histories resulting from lock restoration. If $\mathbf{H}_i$ is value-serializable, it is value-equivalent to a conflict-serializable history $\mathbf{H}_j$. So, transactions read and write the same values in $\mathbf{H}_i$ and $\mathbf{H}_j$. Because the values read and written by transactions are what characterizes a history in a read-write transaction model, a value-serializable history possesses the same validity as a conflict-serializable (or serial) history.

If value-serializability is substituted by conflict-serializability as the correctness criteria, histories such as $\mathbf{H}_3$ would be incorrect. In other words, fewer histories would be accepted as correct; therefore, transaction concurrency and throughput will be lower. Value-serializability offers another equally important advantage. It simplifies the restoration process to a comparison of the value of the data object at restoration. For conflict-serializability, one may need to monitor all write operations on the data object between the preemption and before restoration.

3 A Concurrency Control Scheme

This section describes a locking scheme for concurrency control. In this scheme, restoration is attempted as soon as a preempted lock is available. The victim continues normally if restoration is successful; it is aborted otherwise.

3.1 Transaction Manager Architecture

The transaction manager guarantees the ACID properties of transactions through its components such as buffer and lock managers. The data is stored in the database on the secondary storage. The buffer manager manages the transfer of data between the main memory buffers and the database. The lock manager is responsible for locking and unlocking data items on behalf of transactions. The transaction manager contains other components not relevant to our discussion. For example, the recovery manager component guarantees for the atomicity of transactions.

Figure 1: Designating Parts of a Data Item Based on Operations Performed

Before accessing (reading or writing) a data object, a transaction obtains the appropriate lock on the corresponding data item. The lock is held until the termination, unless preempted by the transaction manager.

We are concerned with the lock preemption and restoration aspects of transaction management and just recall the relevant principles of recovery below. Details such as lock implementation and recovery are beyond the scope of our discussion and can be found in [3].

- If the recovery scheme employs undo, the committed value of a data item (which is read and/or modified by current lock owner) is in the log and the uncommitted value (due to the uncommitted writes of current lock owner) is either in the buffer or the database.
- If recovery scheme employs redo, the committed value is in the log or the database and the uncommitted value is in the buffer.

3.2 Lock Processing

The transaction manager employs the strict two-phase locking scheme [3] and any of the well-known logging and recovery schemes [3]. Let T be a transaction whose mode m lock for data item d is to be preempted.

Based on the read and/or write operations performed by T, d is designated into three parts as shown in Figure 1. The parts are shown to be non-null and contiguous for illustration purposes. They can be null or discontiguous in general. The part where no operations have been performed is called the unaccessed part, d_u. The part that has been read before possibly being written is called the read-from-others part, d_r. d_w is the part that is written to or modified. d_u presents no concern in lock restoration. T has read the value of d_r from some other committed transaction(s). These reads must be remain valid at restoration time. d_w contains the uncommitted writes of T that must be hidden from other transactions. However, these writes must be applied to d_w and be visible to other transactions if and when T commits. The intersection of d_r and d_w represents the part that has been first read and then written.

T's lock on d is preempted as follows:

- If d_r is non-null, a private copy of d_r is created for T. It represents the value of d read from other transactions.
- If d_w is non-null, a private copy of d_w is created for T. It reflects all the (uncommitted) write operations of T. d is then returned to its committed value both in the database and the buffer, as if it has never not been modified by T.
- A special 'restoration' type of lock request is created, where T requires a mode m lock on d. The private copy (copies) information is attached to it.

The preempted lock can now be granted to other transactions. T remains blocked until the preempted lock is restored. If more than one lock is preempted, T remains blocked until all the preempted locks are restored. While blocked, T can be aborted either by the transaction manager or the user that submitted it. Private copies of data items belonging to T are destroyed at abort.

Restoration is attempted when the preempted lock is available (that is, d is no longer locked in a conflicting manner by other transactions):

- The restoration lock request is obtained. The private copies of d_r and d_w, if present, are retrieved. As per the following steps, the lock is restored if possible and T is aborted otherwise.

- If d_r is non-null, the value of d_r from the private copy is compared to the current, committed value in the database. If they differ, the read operations of T prior to preemption have been invalidated. The restoration is terminated and T is aborted. Otherwise, the operations remain valid and the lock can be restored.

- If d_w is non-null, the private copy of d_w is merged in to d. d now reflects all the uncommitted write operations by T.

- The private copies are destroyed. If validation is successful, the lock for d is restored to T.

A restored lock may be re-preempted. The above procedures are repeated for each re-preemption and re-restoration. A transaction cannot commit until all its preempted locks are restored. Thus, a transaction holds all its locks at commit time. The last restoration of a preempted lock is referred to as the final restoration. If a lock is preempted only once, there is only restoration, and that restoration will be called the final restoration.

3.3 Correctness Argument

A history $\mathbf{H}$ generated by the above scheme has to satisfy the correctness criteria, namely, value-serializability. This section proves that the history is conflict-serializable for a particular class of restored locks, where, for each preempted lock, the corresponding data item has not been read by the victim before preemption. In general (when restored locks are not limited in the above sense), the history is shown to be value-serializable.

If no locks are preempted, $\mathbf{H}$ is just a history generated by a two-phase locking scheme and hence conflict-serializable [4]. If locks are preempted, but the victim has not read or write the data item until the lock is finally restored, $\mathbf{H}$ is again conflict-serializable because the lock can be considered to have been acquired when it is finally restored to the victim, and the history becomes equal to one generated by two-phase locking. The remaining discussion concerns only those histories where victims read or write the data item before the final restoration.

If the victim has written the data item before the final restoration, the writes are applied to the database (and are made visible to other transactions) only after the final restoration.[3] So, in $\mathbf{H}$, those writes would appear after any conflicting operations performed by other transactions between the preemption and restoration.

Construct a history $\mathbf{H}'$ with the committed transactions of $\mathbf{H}$ based on the following lock schedule:

- Locks not preempted in $\mathbf{H}$ are obtained at the same time as for $\mathbf{H}$.

- Each preempted lock of $\mathbf{H}$ is locked in $\mathbf{H}'$ at the same time it is finally restored for $\mathbf{H}$.

This description leads to a two-phase locking schedule because the locks are held by the transactions in the same order in $\mathbf{H}$, and no locks are acquired by a transaction after releasing a lock.

$\mathbf{H}'$ is a conflict-serializable history because it has been obtained from a two-phase locking schedule. $\mathbf{H}'$ is conflict-equivalent to a serial history, and any other history conflict-equivalent to $\mathbf{H}'$ would be conflict-serializable as well. We show that (I) $C(\mathbf{H})$ is conflict-equivalent to $\mathbf{H}'$ if the victims could write but not read data items before the final restoration of the locks and (II) $C(\mathbf{H})$ is value-equivalent to $\mathbf{H}'$ if the victims could both read and write data items before the final restoration. We consider the two cases below.

Case I

In $C(\mathbf{H})$, the read/write operations are performed on a data item between the locking and release (commit) in the case of non-preempted locks and between the final restoration and release (commit) in the case of preempted locks. Because the order of lock (for non-preempted locks) and final restoration (for preempted locks) for $C(\mathbf{H})$ is the same as the corresponding lock operations for $\mathbf{H}'$, the order of conflicting operations must be the same in both $C(\mathbf{H})$ and $\mathbf{H}'$. Therefore, $C(\mathbf{H})$ is conflict-equivalent to $\mathbf{H}'$, and $\mathbf{H}$ is conflict-serializable.

[3] Recall that any modifications to the database are canceled when a lock is preempted and made available to other transactions.

Case II

In this case, a victim could read the data item before the final restoration of the lock.[4] Let T_i be a victim that has performed a read operation on data item d before the final restoration in $C(\mathbf{H})$. If no other transaction has performed a conflicting operation on d after the read and before the final restoration, the read operation will conflict with other operations in the same order in $C(\mathbf{H})$ and $\mathbf{H}'$.

However, if some other transaction T_j has performed a conflicting write operation on d after the read and before the final restoration, T_i's read occurs before T_j's write in $\mathbf{H}$, and after T_j's write in $\mathbf{H}'$. Hence, $C(\mathbf{H})$ cannot be conflict-equivalent to $\mathbf{H}'$.

We shall show that $C(\mathbf{H})$ is value-equivalent to $\mathbf{H}'$. We note that the order of operations within each transaction must be the same in both $C(\mathbf{H})$ and $\mathbf{H}'$ because the locking schedule for $\mathbf{H}'$ only affects the inter-transaction operation order. For $C(\mathbf{H})$ to be value-equivalent to $\mathbf{H}'$, we need only show that each read operation performed before the final restoration returns the same value if repositioned after the conflicting write operations of other transactions performed between the preemption and final restoration. Consider the read operation of T_i mentioned above. Let T_k be the transaction that had last written d before the lock was finally restored to T_i. We ought to show that the read operation of T_i, regardless of when it was performed by T_i, returns the same value if performed after T_k's write. According to our scheme, the read operation performed by T_i must be successfully validated at final restoration. Validation implies that the committed value of d at final restoration, reflecting the writes of T_k, is such that the read operation would return the same value that it did when executed by T_i. The argument can be repeated for any other read operation performed before the final restoration of the lock. Hence, $C(\mathbf{H})$ must be value-equivalent to $\mathbf{H}'$.

Example 3.1 $\mathbf{H}_3'$ of Example 2.3 is an example of the histories generated by our scheme. It is reproduced below:

$$
\begin{array}{ll}
\mathbf{H}_3': & lr_2(o_1)\ r_2(o_1)\ lr_1(o_1)\ r_1(o_1)\ lr_3(o_1)\ r_3(o_1) \\
& lr_2(o_2)\ r_2(o_2)\ ur_2(o_2) \\
& lw_1(o_2)\ w_1(o_2)\ lw_1(o_4)\ w_1(o_4)\ c_1 \\
& lw_3(o_3)\ w_3(o_3)\ lw_3(o_4)\ w_3(o_4)\ c_3 \\
& \underline{lr_2(o_2)}\ lw_2(o_4)\ w_2(o_4)\ c_2
\end{array}
$$

The lock and unlock operations are not part of the history; they are included to illustrate the timing of operations relative to the lock preemption and restoration.

$\mathbf{H}_3'$ contains only committed transactions, hence $C(\mathbf{H}_3') = \mathbf{H}_3'$.

The read lock of T_2 for the page containing o_2 has been preempted. Lock restoration is shown underlined. T_2 has read o_2 before preemption and the read operation is validated.[5]

The following is a conflict-serializable history obtained from the lock schedule of $C(\mathbf{H}_3')$, where the restoration is converted to a lock operation.

$$
\begin{array}{ll}
\mathbf{H}_3'': & lr_2(o_1)\ r_2(o_1)\ lr_1(o_1)\ r_1(o_1)\ lr_3(o_1)\ r_3(o_1) \\
& lw_1(o_2)\ w_1(o_2)\ lw_1(o_4)\ w_1(o_4)\ c_1 \\
& lw_3(o_3)\ w_3(o_3)\ lw_3(o_4)\ w_3(o_4)\ c_3 \\
& lr_1(o_1)\ r_1(o_1)\ lr_3(o_1)\ r_3(o_1) \\
& \underline{lr_2(o_2)\ r_2(o_2)}\ lw_2(o_4)\ w_2(o_4)\ c_2
\end{array}
$$

In $\mathbf{H}_3''$, $r_2(o_2)$ is positioned after $w_1(o_2)$, the conflicting operation of T_1 executed between the preemption and restoration.

4 Discussion

4.1 Pros and Cons

The following are some important advantages and disadvantages associated with lock restoration.

- Lock restoration can be implemented in a modular fashion such that overhead is incurred only if lock restoration is attempted. Restoration can be attempted selectively, when the victim is 'valuable'. The victim can be aborted at preemption in other cases.
- Even though value-serializability provides the same degree isolation in terms of reads and writes, conflict-serializability may be essential for certain applications. Our scheme can guarantee the conflict-serializability of histories. But, then, it cannot restore lock if the victim has read the data item before preemption.

[4] The victim could read the data item after the lock is acquired and before it is preempted. If the lock was re-preempted and re-restored multiple times, the read could be performed any time the lock is held - that is, between any adjacent restoration and preemption.

[5] That is, the transaction manager has verified that $r_2(o_2)$ returns the same value if executed after $w_1(o_2)$ as it did when performed before $w_1(o_2)$.

- Value-serializability is guaranteed by our scheme. It is more flexible than conflict-serializability and allows more locks to be restored. An important advantage of value-serializability is that the success test for restoration is a simple comparison of data item values.

- Lock restoration entails CPU, memory, and secondary storage overhead. CPU overhead is incurred to save the victim's lock and operation information when its lock is preempted. Similarly, CPU overhead is involved in 'recovering' the victim's operations at lock restoration. Memory and secondary storage overhead result from the need to preserve the victim's operation until its lock is restored.

- Lock restoration fails when the read operation of victim could not be validated. In such a case, locks would be held idly by the victim between preemption and this failed restoration. Were the victim aborted at preemption, its locks would be available to other transactions immediately.

- Read validation requires disk i/o if the data item, wholse lock has been preempted, is no longer in cache. But, this i/o is necessary whether the lock is restored or the transaction is aborted and redone.

4.2 Applications

Lock restoration appears suitable for environments involving long duration transactions and large-granularity locking. Restoration could preserve the work of a long running victim. Considering that restoration fails only when read operations are invalidated, large-granularity of locking, along with smaller granularity of operation access, improves the chances for restoration.

The following are some cases where lock restoration can be beneficial.

- Relational DBMSs using page locks but allowing tuple operations by transactions.

- Object-oriented DBMSs based on client-server architecture. Usually, these systems employ page-locking and object-based operations. And, commonly, multiple objects are located on a page.

- Global transaction manager in a multi-database system could benefit from using lock restoration. When a subtransaction of a global transaction aborts for some reason, the global transaction manager could try to resurrect that subtransaction by restoring its locks, rather than abort the entire global transaction. Value-serializability is particularly suitable here because read validation can be done at user application level.

4.3 Related Work and Extensions

We are not aware of any reported work on restoration. The closest related work is from IMS Fastpath [5]. Here, lock preemption is avoided to some extent by allocating write locks after a transaction has finished all its operations. Write operations of transaction can be buffered until data items are write locked for. This approach minimizes lock contention by reducing the duration of write locks. A disadvantage with this approach is that obtaining write locks at the end of operations will increase the duration of commit processing. Obtaining available write locks asynchronously with operations of the transaction would reduce this time. If necessary, the locks can be preempted and allocated to other transactions. Another disadvantage of buffering writes is that, if write locks cannot be obtained at commit time, the work by the transaction so far is wasted.

In our scheme, we take the pessimistic approach of blocking the victim until restoration. In an optimistic approach, the victim can continue its operations and block for commit (or prepare, if two-phase locking is involved). Read/write operations by the transaction can be applied against the private copies created at preemption. This approach is optimistic because if restoration fails, the work by the transaction between the preemption and the (failed) restoration is wasted.

Intent locks are supported by several systems. Restoration for preempted intent locks needs to be investigated. It appears that the current algorithm can be used to restore them by treating them as 'regular' locks.

5 Conclusions

Lock preemption is identified as a problem with two-phase locking schemes. The traditional approach to lock preemption is to abort the victim. This paper suggests that the work done thus far by the victim can be saved by restoring the preempted lock. Object-oriented DBMSs based on

client-server architecture, multidatabase systems, and relational DBMSs employing page-locking are some cases where lock restoration could be beneficial. Lock restoration works well when the granularity of locking is larger than the granularity of database operations (reads and writes).

This paper proposes a correctness criteria for isolation in histories generated by a concurrency control scheme that restores locks. The correctness criteria is called the value-serializability. A value-serializable history is shown to have the same behavior (read and write values) as a conflict-serializable. We have described a concurrency control scheme based on lock restoration and have proved that the scheme generates value-serializable histories in general, and conflict-serializable histories for a certain class of preempted locks. Finally, some important advantages and disadvantages of lock restoration have been listed.

Acknowledgements

The authors gratefully acknowledge the support from IBM Canada and the Natural Sciences and Engineering Research Council of Canada. They would like to thank the anonymous referees for their insightful comments and suggestions. They also thank Sreemani Tirumale for her comments on an earlier draft of this paper.

About the Authors

Gopi K. Attaluri: Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: gkattalu@neumann.uwaterloo.ca). Attaluri obtained his Master's degree in Computer Science from the Indian Institute of Technology, Kanpur, India in 1990. He worked in the library automation project at Kanpur during 1990-91. He has been a Ph.D. student at Waterloo since 1991. Attaluri's interests include transaction management, multidatabase systems and object-oriented database systems.

Jacob Slonim: IBM Software Solutions Division, Centre for Advanced Studies, Toronto Laboratory, IBM Canada Ltd., 844 Don Mills Road, North York, Ontario M3C 1V7, Canada (electronic mail: jslonim@vnet.ibm.com). Dr. Slonim is head of research at the Centre for Advanced Studies of the IBM Toronto Software Solutions Laboratory. He received his Ph.D. from Kansas State University in 1979. He is an adjunct professor at the University of Waterloo. He served on several international standard committees and has helped to organize numerous international conferences. In 1993, he became a member of the IBM Academy of Technology. Dr. Slonim was project leader for the CORDS project. His research interests include databases, distributed file systems, and software engineering.

Per-Åke Larson: Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1 (electronic mail: palarson@bluebox.uwaterloo.ca). Per-Åke (Paul) Larson received his Ph.D. from Åbo Swedish University, Finland in 1976. He joined the Department of Computer Science at the University of Waterloo in 1981 and was promoted to Full Professor in 1987. He served as chairman of the department during 1989-92. He was a principal investigator in the CORDS project. His research interests include multidatabase systems, query optimization and processing and parallel databases.

References

[1] S. Ahmed, A. Wong, D. Sriram, and R. Logcher. Object-oriented database management systems for engineering: A comparison. *Journal of Object-oriented programming*, 5(3):27–44, 1992.

[2] G. K. Attaluri, D. P. Bradshaw, N. Coburn, P.-Å. Larson, P. Martin, A. Silberschatz, J. Slonim, and Q. Zhu. The CORDS Multidatabase Project. *IBM Systems Journal*, 34(1), 1995.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publishing Company, 1987.

[4] K. E. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. On the notions of consistency and predicate locks in a relational database system. *CACM*, 19(11):624–34, 1976.

[5] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMW/VS FastPath. *Q. Bull. IEEE TC on Data Engineering*, 8(2):3–10, 1985.