Abstract

In this paper, we formalize and prove the correctness of a nested transaction version of the concurrency control algorithm using a linear hash structure. Nested transactions allow increased parallel execution of transactions, and handle transaction aborts in our system. We present our nested transaction model in a linear hash structure environment using a well-known I/O automaton model. We have modeled both the buckets and the transactions as I/O automata. In our algorithm, the locks have been considered at both key and vertex level. These locks have been implemented in a nested transaction environment using Moss's two phase locking algorithm and the locking protocols of the linear hash structure algorithm with a lock coupling technique. We have proved that our linear hash structure algorithm in a nested transaction environment is 'serially correct'. We have discussed briefly the client–server architecture for the implementation of our system.

Keywords: Nested transaction; Linear hash structure; I/O automaton model; Lock-coupling; Serially correct; Client–server

1. Introduction

The theory of nested transactions [23,27] allows the benefits of atomicity to be available within a transaction. In a nested transaction model [27], a subtransaction may contain operations to be performed concurrently, or operations that may be aborted independent of their invoking transaction. Such operations are considered subtransactions (children) of the original transaction. This parent–child relationship defines a transaction tree. Such hierarchical transactions are termed as nested transactions [27]. Moss presented a concurrency control algorithm using two phase locking for a nested transaction environment [27]. A transaction can acquire a lock on the data
object on request if no other transaction holds a conflicting lock on the data object. Also, all the retainers of the conflicting locks are ancestors of the requested transaction. A transaction can terminate only after all its children are terminated (committed or aborted). Failure of subtransactions may result in invocation of alternate subtransactions that could replace the failed ones to accomplish the successful completion of the whole transaction. A child transaction has access to the data locked by its parent. It is atomic with respect to its parent and its siblings. It is serializable with its siblings. It becomes permanent only if its parent becomes permanent. If a parent aborts, all its descendants’ effects are to be undone. Therefore, a child’s scope is restricted to its parent only. Hence, this model is also termed a closed nested transaction model. When a transaction commits, and if it is not a top level transaction, its locks are inherited by the parent (so that it retains the inherited locks). If it aborts, its locks are discarded. If it is a top-level transaction, its locks are discarded on its termination.

In recent years, attention is being given to the design of concurrency control algorithms which take advantage of the knowledge of particular data structures and the semantics of operations such as insert, delete, find, etc. to improve availability and expedite accesses. Data structures which have been analyzed with above in mind are B-trees and its variants [1,26,34], extendible hashing [3,10,17], linear hashing [4,16,19], exponential hashing [24], and dynamic hashing [5,20,29]. Most of the concurrency control algorithms for data structures assume that a transaction consists of a single decisive operation [14,15] such as read, write, insert, delete, etc. The problems associated with guaranteeing serializability become more complicated when one considers transactions consisting of decisive as well as nondecisive actions (like search). Recently, some work reported in [9,17] support such multiaction transactions. In [29], a concurrency control and recovery algorithm using a linear hash structure with separators [20] is presented but nested transaction model is not considered.

A concurrency control algorithm for accessing B-trees in the nested transaction model [27] using I/O automaton model [23,25] has been proposed in [9] where each transaction performs more than one decisive operation. The notion of ‘strongly-serially correct’ behavior has been defined in [9] and has been used as the correctness criterion. The strongly correct behavior says that a schedule \( \alpha \) is correct if there is a serial schedule \( \beta \) such that no transaction can tell the difference between \( \alpha \) and \( \beta \), and each data object, storing the value of a key, is updated by \( \alpha \) and \( \beta \) in exactly the same sequence. This definition requires the same schedule \( \beta \) for all the transactions \( T \) such that \( \alpha | T \) (projection of \( T \) on \( \alpha \)) = \( \beta | T \) (projection of \( T \) on \( \beta \)). However, the work in [9] has some drawbacks. First, the vertices and keys both are considered to be data objects but the lock management on vertices is not implemented. Lock management on keys is said to be performed by the respective vertex manager but no lock is acquired on the vertex when a key is not found. Each vertex and key need to be locked to ensure that change of data at a vertex is done in an atomic step to ensure serializability. Therefore, it is not clear how serializability is being guaranteed. Secondly, the compression process [34] to merge two adjacent vertices as a result of any deletion of keys, is not discussed. The paper discusses no system implementation issues.

There are various other complex distributed algorithms [2] and areas of transaction processing systems that would benefit from a more rigorous analysis within the framework of the nested transaction model. The level of detail in these algorithms also makes careful reasoning very difficult. A formal correctness of Moss’s two phase locking algorithm for the nested transactions has appeared in [6] using I/O automaton model. In [7], the read-update locking algorithm [35] has
been generalized and a new commutative locking algorithm has been introduced to handle nested transactions. Fekete et al. [8] have presented a serialization graph construction for the nested transactions based on I/O automaton model. Gifford’s basic quorum consensus algorithm for the data replication [11] is generalized by Goldman in [12] to accommodate nested transactions and transaction aborts. Recently, the multi-granularity algorithm given in [13] has been extended to nested transaction systems in [21]. Some more related work by same authors have appeared in [18,22]. In [32], a nested transaction concurrency control algorithm was discussed using I/O automaton model. In [30], we have discussed a linear hash structure algorithm in a multi-level transaction environment [36] to take the advantage of commutative properties of operations at each level of nesting to increase the concurrency.

In this paper, we focus our attention to linear hash structures, which are interesting because of their simplicity and ability to handle large amount of data with increased concurrency. Some real-world applications of linear hashing include database indexing, symbol table look-ups (Fortran variable names) and file and directory hashing in high performance systems. Here we model nested transactions in the environment of linear hash structures [4] using I/O automaton model. Not only there are interesting technical problems that have to be handled in ways other than those used for B-trees, we show that the drawbacks mentioned earlier in the work of [9] have natural and effective solutions. Here, we represent linear hash operations as nested transactions, which are modeled as I/O automata. Buckets in linear hash structures are also modeled as I/O automata. These transactions and buckets automata are described formally using some pre- and post-conditions. The formal description is used to construct a complete correctness proof based on standard assertion techniques and on natural correctness conditions and takes advantage of modularity that arises from describing the algorithm as nested transactions. We have not only verified the theoretical correctness of our model but we have also implemented our system to conform that specifications given for I/O automata can be adopted for system implementation [28]. We have also discussed the performance evaluation in terms of throughput and degree of concurrency in [28]. However, here we have only discussed an overview of the implementation and more details are out of scope of this paper. A preliminary version of this paper has appeared in [33].

Our design of the nested transaction model for the linear hash structure can be adapted as building blocks in the development of the nested transaction models using data structures such as extendible hashing [3,10,17] and other dynamic data structures. This is possible as most of the data structures support operators like search, insert, delete, merge and split and they differ only in methodology. Thus, our nested transaction model and the proof methodology can be easily reconfigured for other data structures.

The rest of the paper is organized as follows: In Section 2, we discuss some preliminaries required to understand later discussions. Section 3 presents an overview of the linear hash structures in a nested transaction environment. In Section 4, we model each transaction as an I/O automaton and discuss the pre- and post-conditions. Section 5 presents modeling of each bucket as an I/O automaton. In Section 6, we discuss link manager automata. We discuss in Section 7 some basic properties about the states of transactions deducible from the bucket manager automaton. We briefly discuss direct data locking objects in Section 8. We discuss correctness of our model in Section 9. We outline the brief implementation details in Section 10. We conclude in last section.
2. Preliminaries

2.1. Overview of linear hash algorithm

In the linear hash structure model [4], there are primary buckets where each bucket holds some constant number of keys. When a primary bucket is full, its overflow records are chained to it. Existence of a family of hash functions $h_1, h_2, h_3, \ldots, h_i, \ldots$ with $h_i : k \rightarrow \{0, \ldots, N - 1\}$ such that for any key value $k$, either $h_i(k) = h_{i-1}(k)$ or $h_i(k) = h_{i-1}(k) + 2^{-1} \times N$, where $N$ is the number of buckets assumed. The hash functions change as the hash structure grows or shrinks.

The operations defined on a linear hash structure are find (search), insert, delete, split and merge. A variable called level is used to determine the appropriate hash function to be used. A variable called next is used to point to the bucket to be split. Level and next together are referred to as root variables. Each bucket keeps an additional field local level that specifies the hash function appropriate to that bucket. The private variable 'lev' keeps the value of level at the time root variables are read. To access a bucket, the process checks whether lev value matches that bucket's local-level, and if not, it increments lev value and recalculates the address $h_{lev}(key)$ until a match is found. This operation is called rehashing. A process in its search phase behaves as follows: The root variables are read and their values determine the hash function to be used initially. The bucket and hash function are updated as follows:

\[
\begin{align*}
\text{lev} &= \text{level} \\
\text{bucket no.} &= h_{lev}(\text{key}) \\
\text{if} \quad &
\begin{cases} 
\text{bucket no.} < \text{next} \text{ then} \\
\text{lev} &= \text{lev} + 1 \\
\text{bucket no.} &= h_{lev}(\text{key}) 
\end{cases} \\
\text{where} \quad &h_{lev}(\text{key}) = \text{key mod } (2^{\text{lev}} \times N)
\end{align*}
\]

The basic strategy of a search process is to permit the use of potentially obsolete information to access a bucket initially, and then, if it turns out to be the wrong bucket, to follow a remedial path through the related buckets. Once the target bucket is located, an insert operation writes the new key or changes the value of the existing key whereas a delete operation sets the value of the key equal to nil.

The insertion of keys may result in a split of a bucket. The split is performed in a cyclic manner using the simple policy that when any bucket overflows, the next bucket in the cycle is to be split. The hash function changes as the bucket splits and therefore, the new hash function assigns a new bucket address to some of the keys previously placed using the old hash function. This new hash function is applied to one bucket at a time in a linear fashion. A split operation moves some keys from the original bucket to a new primary bucket appended at the current end of the hash file. A split operation increases the address space of the primary buckets to reduce the accumulation of overflow chains. Formally, the split operation is performed on the bucket that is pointed by the pointer next. The pointer next is initially set to bucket 0. The next pointer travels from 0 to $2^{l-1} \times N$ with hash function $h_l$. After the split, the next is updated as follows:
next ← (next + 1) MOD (N * 2 * * level)
If next = 0 then level ← level + 1

A merge operation is triggered when an individual primary bucket is emptied by a delete operation. The merge operation readjusts the variable level and moves back the variable next as follows:

If next = 0 then level ← level - 1
next ← next - 1 MOD(N * 2 * * level)

The find operation can be performed concurrently with the other processes executing find, insert, delete and split operations. A process executing insert and delete respectively may operate in parallel if they are on different buckets. A split may be performed in parallel with insert and delete that are not accessing the particular chain being split. The interaction between a merge and a process doing find, insert, or delete operation is more complicated. These processes may not access the two buckets being merged and may not read the values of level and next while merging process is using them. At most one restructuring operation merge or split is executable at a time.

The locking scheme employed in [4] is as follows: The primary bucket and all its overflow buckets are locked as a unit. The find algorithm uses lock-coupled [4,15] read-lock on the root and holds the lock until a read-lock is placed on the bucket. Lock-coupling provides the particular flow of locking in which next component is locked before releasing the lock on the current component. The insert and delete processes put read-lock on the root and selective-lock on the buckets with lock-coupling. The split operations use selective-locks on the root as well as on the buckets. Exclusive-locks are used on the root and both the buckets involved in merge when the old overflow buckets are deallocated. If a rehashing is required, a lock is placed on the subsequent bucket before the lock is released on the current bucket. The readlock on the root held by the searching process prevents a merge from decreasing the size of the address space (by updating level and next) during the initial bucket access. The lock compatibility matrix is given in Fig. 1. The correctness of locking algorithm ensures that the concurrent execution of transactions is equivalent to the serial execution of the same transactions.

2.2. User transaction

We define the user transactions in a system to be all the nonaccess transactions. User transactions are user-visible in the sense that all the subtransactions are invoked explicitly by the user within the user program.

<table>
<thead>
<tr>
<th>Lock Request</th>
<th>Existing Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read-lock</td>
</tr>
<tr>
<td>Read-lock</td>
<td>yes</td>
</tr>
<tr>
<td>Selective-lock</td>
<td>yes</td>
</tr>
<tr>
<td>Exclusive-lock</td>
<td>no</td>
</tr>
</tbody>
</table>

Fig. 1. Lock compatibility matrix.
When the user explicitly may not invoke subtransactions but the system may provide a set of subtransactions then these subtransactions are user-invisible. For example, in any search-access operation, the system may provide search as a subtransaction before accessing the data object. Similarly, we call an object user-visible if it is explicitly specified in the user-visible transactions. In our analysis, we have considered nesting of only user-invisible transactions. One can generalize this to the nesting of the user-visible transactions as well.

2.3. I/O automaton model

An I/O automaton $A$ as defined by Lynch et al. [23] is a 5-tuple $(\text{states}(A), \text{start}(A), \text{out}(A), \text{in}(A), \text{steps}(A))$. States($A$) is the set of states of $A$, start($A$) is the set of start states and a subset of states($A$). Out($A$) and in($A$) are the sets of output and input operations respectively. Steps($A$) is the transition relation of $A$, which is a set of triples of the form $(s', \pi, s)$ where $s'$ and $s \in$ states($A$), $\pi \in$ in($A$) $\cup$ out($A$), i.e., the automaton changes its state from $s'$ to $s$ on operation $\pi$. An element of the transition is called a step of $A$.

The finite alternating sequence $s_0, s_1, s_1, \ldots, s_n, s_r$ of the states and operations of $A$ is called an execution of $A$. A schedule of $A$ is the subsequence of an execution of $A$ consisting only of the operations of $A$.

A set of I/O automaton may be composed to create a system $S$ such that the sets of output operations are disjoint. Thus, a state of the composed I/O automaton is a tuple of states, one for each component and the start states are tuples consisting of start states of the components. Let $\alpha$ be a schedule of a system with a component $A$, then $\alpha \upharpoonright A$ is the subsequence of $\alpha$ containing exactly the operations of $A$. Clearly, $\alpha \upharpoonright A$ is a schedule of $A$. The reverse holds by the Composition Lemma [23] which is formally stated as follows: Let $\sigma'$ be a schedule of a system $S$ and let $\sigma = \sigma' \pi$ be an output operation of the component $A$. If $\sigma \upharpoonright A$ is a schedule of $A$, then $\sigma$ is a schedule of $S$.

2.4. Nested transaction system and correctness

A nested transaction system is modeled by a 4-tuple $(\tau, \text{parent}, O, V)$, where $\tau$ is a set of transaction names organized into a tree by the mapping parent: $\tau \rightarrow \tau$, where $\tau_0$ acts as the root. The set $O$ denotes the set of objects; it partitions the set of accesses, where each partition block contains accesses to the particular objects. $V$ is the set of return values. We can relate two nested transaction systems as follows: a nested transaction system $P = (\tau_P, \text{parent}_P, O_P, V_P)$ is called a structural extension of the nested transaction system $Q = (\tau_Q, \text{parent}_Q, O_Q, V_Q)$ if $\tau_P \supseteq \tau_Q$, $O_P \supseteq O_Q$, respectively, $V_P = V_Q$, and parent$_P$, restricted to $\tau_Q$, = parent$_Q$.

A nested transaction processing system is said to be modeled using the I/O automata when each components of the system, namely each nonaccess transactions, objects and the scheduler, is modeled as an automaton. Each of these automata is specified with the help of some pre- and post-conditions. These pre- and post-conditions are used to prove the properties describing the behavior of the system. Formulation of the nested transaction systems as I/O automata permits precise correctness conditions to be satisfied by the algorithm. These correctness conditions can be stated at the transaction interface that does not contain explicit information about object representation.

The correctness of a transaction processing system is defined in terms of a serial execution of the 'same' system. That is, it requires an execution of the same system to exist in which transactions
run one at a time without interleaving of steps of different transactions. Correctness is defined by first giving a separate specification of permissible serial executions as seen by a user of the system, and then defining how executions of a transaction processing system must relate to this specification. The permissible serial execution for a transaction processing system is defined by introducing the notion of a scheduler, which executes transactions serially. Such systems, called serial systems, are not constrained by the issues of concurrency control, recovery and transaction aborts.

Formally, a schedule \( \alpha \) of a system is serially correct for a transaction \( T \) if its projection on \( T \), \( \alpha \mid T \), is identical to \( \beta \mid T \) for some serial schedule \( \beta \) [25]. In other words, \( T \) sees same things in \( \alpha \) that it would see in some serial schedule. \( \alpha \) is serially correct if it is serially correct for every nonorphan, nonaccess transaction.

The principal notion of correctness for a transaction processing system is that of serial correctness of the root transaction \( T_0 \) of all finite schedules. This says that ‘outside world’ cannot distinguish between the given system and the serial system. A fairly strong and possible interesting correctness condition is the serial correctness of all the nonaccess transactions. In this case, neither the outside world nor any other individual user transaction can distinguish between the given system and the serial system. Note that the definition of serial correctness, relative to all the nonaccess transactions, does not require that all transactions see schedules that are a part of the same execution of the serial system; rather each could see schedules arising in a different serial execution.

The serial correctness for all nonorphan transactions implies serial correctness for \( T_0 \) because the serial scheduler does not have the action ABORT(\( T_0 \)) so \( T_0 \) cannot be an orphan. Note that each correctness condition discussed here can be applied to many different kinds of transaction processing systems. All that is needed is that the system be modeled as an I/O automaton with the appropriate named actions.

2.4.1. Transaction automata

Each transaction is modeled as an I/O automaton which is specified with the help of the following operations [6]:

**Input operations:**
- CREATE(\( T \))
- REPORT-COMMIT(\( T', v \)), where \( T' \in \text{children}(T) \) and \( v \) is the return value
- REPORT-ABORT(\( T' \)) where \( T' \in \text{children}(T) \)

**Output operations:**
- REQUEST-CREATE(\( T' \)) where \( T' \in \text{children}(T) \)
- REQUEST-COMMIT(\( T, v \)) where \( v \) is the return value

The CREATE operation wakes up the transaction. The REQUEST-CREATE is a request by \( T \) to create a particular child transaction. The REPORT-COMMIT operation reports to \( T \) the successful completion of one of its children and returns a value depending upon the operation performed. The REPORT-ABORT operation reports to \( T \) the unsuccessful completion of one of its children, without returning any value. The REQUEST-COMMIT is a request by \( T \) to commit.

2.4.2. Generic object automata

The generic object automata [6] serve as the specifications of the concurrent behavior of the operations on the data objects. The operation for each object are the CREATE, REQUEST-
COMMIT operations for all the corresponding access transactions. The CREATE operation is an invocation of an access to the object, while the REQUEST-COMMIT is a return of value in response to such an invocation. It has two additional input operations INFORM-COMMIT and INFORM-ABORT for every transaction, which inform about the fate of the transaction. It also has an internal operation OBTAIN-LOCKTYPE($T$) for each $T$, an access to $b$. Thus, an object $b$ (bucket) is modeled as an automaton with the following operations:

**Input operation:**
- CREATE($T$)
- INFORM-COMMIT-AT-($b$) OF $T, T \neq T_0, b \in H$; $H$ is a linear hash structure
- INFORM-ABORT-AT-($b$) OF $T, b \in H$

**Output operation:**
- REQUEST-COMMIT($T, v$) where $T \in \text{accesses}(b)$

**Internal operation:**
- OBTAIN-LOCKTYPE($T$) for each $T$, an access to $b$

2.4.3. Generic scheduler automaton

The generic scheduler [23] is modeled as an I/O automaton. It passes request for the creation of subtransactions to the appropriate recipient, makes the decision about the completion of the children and reports back to their parents, and informs objects of the fate of transactions. The operations are as follows:

**Input operation:**
- REQUEST-CREATE($T$)
- REQUEST-COMMIT($T, v$)

**Output operations:**
- CREATE($T$)
- COMMIT($T$)
- ABORT($T$)
- REPORT-COMMIT($T, v$)
- REPORT-ABORT($T$)
- INFORM-COMMIT-AT-($b$) OF $T, b \in H$
- INFORM-ABORT-AT-($b$) OF $T, b \in H$

The REQUEST-CREATE and REQUEST-COMMIT inputs are identified with the corresponding output operations CREATE, REPORT-COMMIT and REPORT-ABORT of the transactions and the object automata. The COMMIT and ABORT operations are internal, marking the point of time where the decision on the fate of the transaction is irreversible. The COMMIT($T$) and ABORT($T$) are called return operations. The INFORM-COMMIT and INFORM-ABORT informs the data object automaton about the fate of transactions, for details see [6].

3. Linear hash structure algorithm and nested transactions

In this section, we briefly describe the modeling of the linear hash structure algorithm of [4] in the nested transaction environment as shown in Fig. 2.
In the nested transaction tree in Fig. 2, next to the level of user-visible transactions, we have a level of transaction managers (TMs). The find operation is performed logically by an equivalent read-TM whereas the insert and delete are performed by the write-TMs. The search-read and search-write (for insert or delete) access subtransactions are situated at the leaf level the level below TMs. Search-write accesses physically modify or delete the keys present in a bucket whereas the search-read accesses read the keys present in a bucket. For split and merge, a split-TM and a merge-TM are provided. These operations are delinked from the insert and delete operations. During an insert operation, an overflow means a split is required whereas a delete operation may account for an underflow which signals the need for merging. If the root transaction $T$ intercepts an overflow message, it triggers a split-TM whereas if it intercepts an underflow message, it invokes a merge-TM. These TMs invoke the access subtransactions of the type split-write and merge-write to physically accomplish the split and merge operations, respectively. Release access subtransactions are created to release the locks on the penultimate buckets and are explained later in this section.

A bucket manager (BM) manages each bucket and its entire overflow buckets. A BM provides the lock management on the keys present in a bucket and in its overflow buckets. Since a key $k$ in
a linear hash structure is a resilient object [23]. Different transactions may see different versions of the same key. However, there is only one version of each linear hash structure. Therefore, at any time, the set of all keys contained in a linear hash structure is partitioned among the buckets, and each key in a bucket has a set of versions associated with it. When keys move from a bucket to another as a result of the split or merge, all its versions move with it.

In our model, the transactions situated at one level above the leaf level also hold locks on the root variable being provided by the scheduler. The value of the root variable would be returned to the scheduler when a lock is granted to a transaction. Lock management on the root variable and on the buckets is done using Moss's two phase locking algorithm [27] and the locking algorithm using the lock-coupling protocols given in [4].

According to the implementation technique of [23], each access subtransaction can lock a single data object (a bucket including all its overflow buckets in our case) only and after its commit, it has to release the lock. However, the compression process in the linear hash structure algorithm needs two buckets to be locked at a time before accessing any of the buckets. Therefore, the implementation of the compression process is difficult with the existing technique of representing each object as an automaton. To implement a merge operation, a link manager (LM) automaton for the two merging buckets is introduced which provides locks on the buckets simultaneously. A LM automaton is created dynamically by combining the two BM automata to be merged. That is, initially the system does not contain any LM automata. Since a merge operation may not be initiated at all, and therefore, the system requires no preexisting LM automata. The components in a state of an LM automaton are derived from the corresponding components of the two linked BM automata. The difference between a BM and an LM automata is that the former allows a subtransaction to access a single bucket while the later allows a subtransaction to modify the two related buckets simultaneously.

The lock-coupling technique is also difficult to implement using I/O automaton model where each object automaton has only two actions; one requesting the action (similar to the invocation of an operation) and the other returning a result (representing the response to the invocation). There is no intermediate returned action before the final response to the request. Therefore, it is not possible for a subtransaction to inform its parent about acquiring the lock on the bucket before its commit. In case the parent gets the information early, it can initiate the process to release the lock acquired on the previously accessed bucket as required by the lock-coupling technique of [4]. Therefore, to implement the lock-coupling technique in a nested transaction environment, a minor modification is incorporated. We release (drop) the lock held on the penultimate bucket (one before the last accessed bucket) instead of the previously accessed bucket after a lock is acquired on the next searched bucket with the help of another subtransaction, called 'release access' shown in Fig. 2. This is also because in a nested transaction model, a subtransaction passes its lock to its parent subtransaction on commit. In our case, we want to drop the lock (i.e., not to be retained by its parent) held by the search-access subtransaction on the penultimate bucket once the key $k$ is not found in that bucket.

---

1 A resilient object has no capabilities for managing concurrency; rather it assumes that the concurrency control is handled externally by the bucket managers. Also, a resilient object has additional capabilities to undo operations of transactions that it discovers have aborted.
For the correctness, we use the definition of 'serial correctness' given in [12], which does not require the same serial schedule satisfying \( x | T = \beta | T \) for every nonaccess transaction \( T \) in contrast to 'strongly-serial correctness' defined in [9].

On the system implementation, when a user requests for an operation to be performed, the agent initiates the corresponding TM. TM can further initiate the various subtransactions, which are passed to the appropriate bucket containing the data needed. TM receives the data from the subtransaction and processes the collected data, and may initiate some more subtransactions if it is required. The scheduler as shown in Fig. 2 determines the order in which the subtransactions are to be executed. The scheduler receives two types of request; a request to create a transaction (REQUEST-CREATE) or to commit a transaction (REQUEST-COMMIT). In return, the scheduler responds by creating, committing, or aborting the transaction (Fig. 2). The scheduler will control operations of the subtransactions, makes a decision about the completion of the subtransaction and reports back to their parent (a report-commit or report-abort), and informs objects the fate of the transactions. Transactions situated at one level above the leaf level hold lock on the root variable (i.e., level and next) managed by the scheduler. More details on the system implementation can be found in [31].

4. Linear hash structures, nested transaction system and I/O automata

The linear hash system (LH) described before can be framed as the nested transaction system \( B = (\tau_B, \text{parent}_B, O_B, V_B) \) where \( \tau_B \) is the set of transaction automata described above, \( \text{parent}_B : \tau_B \rightarrow \tau_B \) is a mapping which describes the parent-child relationship. \( O_B \) is the set of buckets and \( V_B \) is the set of returned key values.

Each state of an automaton is defined in terms of a subset of the following components:
1. Created is a boolean variable used to initiate a CREATE operation.
2. \( p \) is boolean variable which controls the initiation of the release access subtransactions.
3. \( i \) is a boolean variable to control the initiation of release, search-read or search-write access subtransactions.

All the boolean variables are initialized to false.
4. Data holds the return value. Initially, data is undefined.
5. Phase can be of the type 'idle', 'searching', 'finished', 'merge' or 'split'.
6. Nextbucket keeps track of the hash function that gives the address of the nextbucket in the next bucket chain to be searched during the search operation for the desired key \( k \) starting from the first bucket chain of the linear hash structure.
7. \( rv \) contains the root variables next and level.
8. \( lev \) keeps the level value.
9. Count is a control variable. Initially, it is zero.
10. root(\( T \)) is the initial hash function associated with the transaction \( T \).
11. key(\( T \)) is the key associated with the transaction \( T \).
12. read-lockholders is the set of all the transactions holding read-locks on the root variable \( rv \). Initially, read-lockholders = \( \{T_0\} \).
13. sel-lockholders is the set of all the transactions holding selective-locks on the root variable \( rv \). Initially, sel-lockholders = \( \{T_0\} \).
14. excl-lockholders is the set of all the transactions holding exclusive-locks on the root variable $rv$. Initially, excl-lockholders = \{T_0\}.

15. putchain gives the address of the new bucket in case of split. Initially, putchain is undefined.

16. release-bucket_1 and release-bucket_2 give the addresses of the previously accessed bucket and the penultimate accessed bucket, respectively. Initially, they are undefined.

17. merge-1 and merge-2 give the addresses of the two buckets to be merged. Initially, both are undefined.

18. read-committed and write-committed are the sets of committed search-read and search-write access subtransactions, respectively. Initially, both the sets are empty.

19. $r(1)$ returns a value of the type ‘found’, ‘not found’, ‘underflow’, ‘overflow’, and ‘merge’, and $r(2)$ either returns the address of the next bucket to be searched or it returns the value associated with the key. Initially, $r(1)$ and $r(2)$ are undefined.

$H$ will denote the linear hash structure.

**Read-TM**

The purpose of a read-TM is to initiate a read access to read one of the keys $k \in H$. The read-TM invokes a read access to a bucket. There are bucket managers for each bucket $b_1, \ldots, b_m \in H$. The read-TM has a lev variable initially assigned the value level, read from the root variable $rv$. Let $b_w$ be the bucket whose local-level matches with the lev value of the read-TM, i.e., the bucket $b_w$ holds the key $k$ if key $k \in H$. Accesses to buckets which led to $b_w$ are non-decisive operations and the read access to $b_w$ is a decisive operation.

A state of the read-TM uses the components data, phase, nextbucket, created, $p, rv$, lev and count to define pre- and post-conditions. The scheduler maintains and provides read-lock on the root variables, and also provides an initial hash function.

**CREATE($T$) with root($T$) = ‘initial hash function’, key($T$) = $k$, and $rv$ = (level, next)**

The first precondition asserts that $T$ is not created before. The second precondition says that the read-lock on the variable $rv$ must be with the ancestors of $T$. Since $rv$ is passed as a parameter by the scheduler, it is not associated with the state of the automaton. The first postcondition informs that $T$ is created. The second condition indicates that search phase has started for the key $k$. The variable key associated with the transaction is assigned the key $k$ to be searched. The condition data = nil informs that the initial data value is nil. In the following two conditions, a read-lock is granted to $T$ on the root variable $rv$ and the root variables level and next are assigned to $rv$. The next condition asserts that the variable lev associated with $T$ maintains an initial level value. The variable $h_1$ keeps the initial hash function that gives the address of the first bucket to be searched. Finally, if the nextbucket value is less than the next value, lev value of the transaction is incremented till the $h_j$ value is greater than or equal to next and a new address is assigned to nextbucket recalculated by the new hash function (This condition corresponds to rehashing technique discussed in Section 2.1).

**Precondition:**
 $s'.created = false$
 $\text{read-lockholders}(rv) \subseteq \text{ancestors}(T)$

**Postcondition:**
 $s.created = true$
 $s.phase = \text{‘searching’}$
key(T') = k
s.data = nil
read-lockholders(rv) = read-lockholders(rv) ∪ \{T'\}
rv = (level,next)
lev(T') = level
\{∃ a sequence h_1, h_2, \ldots h_j \exists
h_1 = root(T)
h_i = h_{lev}(T) + i(k) \quad ∀i \leq j
h_i < next ∀i < j
h_j \geq next
s.nextbucket = h_j\}

REQUEST-CREATE \(T') with type(T') = search-read and key(T') = k\)

The first precondition is that \(T'\) must be an access to the bucket whose address is given by the variable nextbucket. The other preconditions are as before except the last which says that the previous release access (if any) has been committed. The automaton has no postconditions and therefore, all the variables in the states \(s'\) and \(s\) will be same.

Precondition:  
 \(T' \in\) access \((s'.nextbucket)\)
\(s'.phase = \text{'searching'}\)
\(s'.created = \text{true}\)
key(T') = k
lev(T') = lev(T)
\(s'.i = \text{false}\)

REPORT-COMMIT\((T', v) with type(T') = search-read and v = (r(1), r(2))\)

The only precondition asserts that search process is not completed. The first postcondition says that lock held by the read-TM \(T\) on \(rv\) is released. The postcondition \(r(1) = \text{‘found’}\) says that data have the required returned value. If count = 1, i.e., if the key is found in the second searched bucket then the variable \(p\) is set to true so that the process to release the lock retained by \(T\) on the previously searched bucket is to be initiated. Otherwise, if \(r(1) = \text{‘not found’}\) then the variable called release-bucket-2 keeps the address of the penultimate bucket searched whereas release-bucket-1 keeps the address of the last searched bucket. In the following two conditions, \(r(2)\) assigns the address of the nextbucket to be searched to the variable nextbucket and lev value of \(T\) is incremented by one. In the next condition, if count \(\geq 2\) then \(s.p = \text{true}\) and \(s.i = \text{true}\), i.e., the process to release the lock acquired on the penultimate searched bucket is to be initiated. Next \(T\) is added to the set read-committed. The variables \(r(1)\) and \(r(2)\) are returned by the bucket manager automaton BM(b).

Precondition:  
\(s'.phase = \text{'searching'}\)

Postcondition:  
If \(T \in\) read-lockholders(rv) then read-lockholders(rv) \(=\) read-lockholders(rv) \(\cup\) \{parent(T)\}
If \(r(1) = \text{‘found’}\) then
\(s\).phase = 'finished',
\(s\).data = \(r\)(2)
If \(s\).count = 1 then \(s\).p = true
If \(r\)(1) = 'not found' then
\{\(s\).release-bucket_2 = \(s\)'release-bucket_1
\(s\).release-bucket_1 = \(s\)'nextbucket
\(s\).nextbucket = \(r\)(2)
lev(\(T\)) = lev(\(T\)) + 1
\(s\).count = \(s\).count + 1
If \(s\).count \(\geq\) 2 then \{\(s\).p = true, \(s\).i = true\}{/}
\(s\).read-committed = \(s\)'read-committed \(\cup\) \{\(T\)\}

REQUEST-CREATE(\(T\')) with type (\(T\')) = release access

This operation is to release the lock acquired on the penultimate accessed bucket. This operation has two sets of preconditions. However, only one set of preconditions is satisfied at a given time. The first two preconditions in both are as explained before. The precondition \(s\)'p = true in both the conditions says that some transaction \(T\') holds the lock on the target bucket, whose lock is to be released. The following two conditions in the first set check that there is a penultimate accessed bucket and the key is not found so far. The similar two conditions in the second set check that either there is no penultimate searched bucket or the key searched is found. The following precondition in both says that \(T\') must be an access to the penultimate accessed bucket. The last condition in both says that the transaction, which accessed the penultimate bucket, must have been committed.

Precondition: \(s\)'created = true
\(s\)'phase = 'searching' or 'finished'
\(s\)'p = true
\(s\)'i = true
\(s\)'release-bucket-2 \(\neq\) nil and \(r\)(1) \(\neq\) 'found'
\(T\') \(\in\) access (\(s\)'release-bucket-2)
last \(T\") \(\in\) access (\(s\)'release-bucket-2) \(\in\) \(s\)'read-committed

Precondition: \(s\)'created = true
\(s\)'phase = 'searching' or 'finished'
\(s\)'p = true
\(s\)'i = true
\(s\)'release-bucket-2 = nil or \(r\)(1) = 'found'
\(T\') \(\in\) access (\(s\)'release-bucket-1)
last \(T\') \(\in\) access (\(s\)'release-bucket-1) \(\in\) \(s\)'read-committed

REPORT-COMMIT(\(T\'), \(v\)) with type (\(T\')) = release access

Postcondition: \(s\).i = false
If \(r\)(1) = 'found' then \(s\).p = false
REPORT-ABORT\(T')\)

Postcondition: no change

REQUEST-COMMIT\((T, v)\)

Precondition: \(s'.phase = \text{'finished'}\)
\(s'.p = \text{false}\)
\(v = s'.data\)

Postcondition: \(s.phase = \text{'idle'}\)

Write-TM

The purpose of a write-TM is to store a new key and its value or to change the value of the existing keys in \(H\). Each write-TM has a key\((T)\) and the data\((T)\) as parameters. To delete a key, we set data\((T) = \text{nil}\). A write-TM invokes a search-write access where key\((T)\) is stored or updated in the bucket \(b\) in a bucket chain. All the variables used to define pre- and post-conditions are as in the case of read-TM. The details are in Appendix A.

Split-TM

A split-TM is used to perform a split operation on receiving an overflow message. The root transaction \(T\) invokes a split-TM. The split operation results in a splitting of a bucket into two new buckets. The bucket at the new address is written using the hash function before the new version replaces the target bucket. Its local-level indicates that the bucket has split and the new bucket has been incorporated into the hash structure. All the components used to define pre- and post-conditions are as before. The details are in Appendix B.

Merge-TM

When the root transaction \(T\) intercepts an underflow message, it triggers a merge-TM. The purpose of a merge-TM is to merge the two buckets. The variable next is updated after the merge operation. All the state variables used to define pre- and post-conditions are as defined earlier. The details are in Appendix C.

5. BM automaton

In system \(B\), we have a bucket manager for each bucket \(b \in H\). Each BM also maintains the keys and their associated values. We have locks on the buckets as well as on the keys. Therefore, both the buckets and the keys are the objects in system \(B\). Each bucket manager is modeled as an automaton and it manages keys in the buckets. We consider keys as the user-visible objects and buckets as user-invisible objects. Each bucket manager automaton provides the resilient lock management [23] for the keys in the bucket.

The set of all the keys in a linear hash structure \(H\) is partitioned among the primary bucket and its overflow buckets. Each key in a bucket has a set of versions associated with it. When a key moves from a bucket to another as a result of splitting or merging, all its versions move with it.
The accesses to a bucket $b$ are of the type search-write, search-read, split-write and release access. These accesses interact with an automaton via the scheduler. A state variable called created contains accesses to $b$ that has been initiated. The variable run contains accesses to the bucket $b$ on whose behalf $BM(b)$ has output REQUEST-COMMITs. The variable lev is the local level value of $b$, keyset stores the keys that are in $b$, map is a variable which maps transactions to contents($b$), and mapsize maps the transactions to the number of non-nil keys in $b$. $r(1)$ and $r(2)$ are the returned values. All other variables are as defined before. $BM(b)$ has the following operations:

**CREATE($T$) with key($T$) = $k$ and lev($T$) = level**

Precondition: $T \in$ accesses ($b$)

Postcondition: $s$.created $=$ $s'$.created $\cup \{T\}$

**OBTAIN-LOCKTYPE($T$) where type($T$) = search-read**

The first precondition is that $T$ is created but is not completed. The second precondition asserts that the excl-lock on $b$ must be with the ancestors of $b$. The postcondition says that a read-lock on $b$ is granted to $b$.

Precondition: $T \in s'$.created $-$ $s'$.run

$s'$.excl-lockholders ($b$) $\subseteq$ ancestors ($T$)

Postcondition: $s$.read-lockholders ($b$) $=$ $s'$.read-lockholders ($b$) $\cup \{T\}$

**OBTAIN-LOCKTYPE($T$) where type($T$) = search-write**

Precondition: $T \in s'$.created $-$ $s'$.run

$s'$.sel-lockholders ($b$) $\cup$ $s'$.excl-lockholders ($b$) $\subseteq$ ancestors ($T$)

Postcondition: $s$.sel-lockholders ($b$) $=$ $s'$.sel-lockholders ($b$) $\cup \{T\}$

**OBTAIN-LOCKTYPE($T$) where type($T$) = split-write**

Precondition: $T \in s'$.created $-$ $s'$.run

$s'$.sel-lockholders ($b$) $\cup$ $s'$.excl-lockholders ($b$) $\subseteq$ ancestors($T$)

Postcondition: $s$.sel-lockholders ($b$) $=$ $s'$.sel-lockholders($b$) $\cup \{T\}$

**REQUEST-COMMIT($T, v$) with type($T$) = search-read, key($T$) = $k$, and $v$ = ($r(1), r(2)$)**

The second precondition says that key $k$ to be read should belong to the keyset in state $s'$ and $T$ must have obtained a lock on bucket $b$. Then the write-lock on key $k$ must be with the ancestors of $T$. Next condition asserts that if lev variable in state $s'$ of the bucket manager for bucket $b$ matches the lev value of transaction $T$ then key $k$ is found in $b$. $r(1)$ and $r(2)$ are the returned variables with the values $r(1)$ = found and $r(2)$ is mapped to the value of key $k$ in state $s'$ read by the transaction $T$ whose least common ancestor holds a write-lock on $k$ among $s'$.write-lockholders. The two postconditions assert that a read-lock is granted to $T$ on the key $k$ and the variable run is updated in state $s$. 
Precondition: \[ T \in s'.created \land s'.run \]
\[ k \in s'.keyset \]
\[ T \in s'.read-lockholders (b) \]
\[ s'.write-lockholders (k) \subseteq \text{ancestor} (T) \]
If \[ s'.lev = \text{lev}(T) \] then
\[ \{r(1) = \text{'found'}\} \]
\[ r(2) = s'.map ( \text{least} (s'.write-lockholders, k)) \] \]
Postcondition: \[ s.\text{read-lockholders}(k) = s'.\text{read-lockholders}(k) \cup \{T\} \]
\[ s.\text{run} = s'.\text{run} \cup \{T\} \]

REQUEST-COMMIT\((T, v)\) with type \((T) = \text{search-read}\) and key\((T) = k\), and \(v = (r(1), r(2))\)

Precondition: \[ T \in s'.created \land s'.run \]
\[ T \in s'.\text{read-lockholders} (b) \]
If \[ s'.\text{lev} \neq \text{lev} (T), \text{i.e., k} \notin s'.\text{keyset} \] then \(r(1) = \text{'not found'}\)
\[ r(2) = h_{\text{ev}}(T) + 1 \]
Postcondition: \[ s.\text{run} = s'.\text{run} \cup \{T\} \]

REQUEST-COMMIT\((T, v)\) with type\((T) = \text{search-write}\), key\((T) = k\), data\((T) = d\) and \(v = (r(1), r(2))\)

The fifth precondition says that if size of the bucket in state \(s'\) is maxsize then the new key \(k\) is accommodated in \(b\) and the variable \(r(2)\) returns the message ‘found and overflow’. Otherwise, \(r(1)\) returns ‘found’ and \(r2\) is set to nil. The first two postconditions say that if key \(k\) is found, maxsize is incremented by one and the write-lock is granted to \(T\). In the following condition, the transaction \(T\) is mapped in the state \(s\) to the least common ancestor of \(T\) in the transaction tree among \(s'.\text{write-lockholders}\). Next the value of the key \(k\) is updated and \(T\) is added to the set run.

Precondition: \[ b \text{ is the bucket such that } k \in s'.\text{keyset} \]
\[ T \in s'.\text{created} \land s'.\text{run} \]
\[ T \in s'.\text{sel-lockholders} (b) \]
\[ s'.\text{write-lockholders} (k) \cup s'.\text{read-lockholders} (k) \subseteq \text{ancestors} (T) \]
If \[ s'.\text{mapsSize} (T) = \text{maxsize} \] then \(r(1) = \text{'found and overflow'}\) else \(r(1) = \text{'found'}\)
\[ r(2) = \text{nil} \]
Postcondition: \[ \text{If } s.\text{map} (T,k) = \text{nil} \text{ then } s.\text{mapsSize} (T) = s'.\text{mapsSize} (T) + 1 \]
\[ s.\text{write-lockholders} (k) = s'.\text{write-lockholders} (k) \cup \{T\} \]
\[ s.\text{map} (T) = s'.\text{map} (\text{least} (s'.\text{write-lockholders}, k)) \]
\[ s.\text{map} (T,k) = d \]
\[ s.\text{run} = s'.\text{run} \cup \{T\} \]

REQUEST-COMMIT\((T, v)\) with type\((T) = \text{search-write}\), key \((T) = k\), data\((T) = \text{nil}\) and \(v = (r(1), r(2))\)
Precondition:  
- $b$ is the bucket where $k \in s'.keyset$
- $T \in s'.created \implies \neg s'.run$
- $T \in s'.sel-lockholders (b)$
- $s'.write-lockholders (k) \cup s'.read-lockholders (k) \subseteq \text{ancestors (} T \text{)}$

Postcondition:  
- $s'.write-lockholders (k) = s'.write-lockholders(k) \cup \{T\}$
- $s.mapsize(T) = s'.mapsize(T) - 1$
- If $s.mapsize (T) = \text{nil}$ then $r(1) = \text{'found and underflow'}$ else $r(1) = \text{'found'}$
- $r(2) = \text{nil}$
- $s.map (T) = s'.map (\text{least (} s'.write-lockholders, k))$
- $s.map (T,k) = \text{nil}$
- $s.run = s'.run \cup \{T\}$

REQUEST-COMMIT($T, v$) with type($T$) = search-write, key($T$) = $k$, data($T$) = nil and $v = (r(1), r(2))$

Precondition:  
- $b$ is an intermediate bucket, i.e., $k \notin s'.keyset$
- $s'.lev \neq \text{lev}(T)$
- $r(2) = h_{\text{lev}(T)}$
- $r(1) = \text{'not found'}$
- $T \in s'.sel-lockholders (b)$

REQUEST-COMMIT($T, v$) with type ($T$) = release access

Precondition:  
- $b$ is the intermediate bucket
- parent ($T$) $\in s'.sel-lockholders (b)$ or parent ($T$) $\in s'.read-lockholders (b)$

Postcondition:  
- If parent ($T$) $\in s'.sel-lockholders (b)$ then
  - $s.sel-lockholders (b) = s'.sel-lockholders (b) - \{\text{parent}(T)\}$
- If parent($T$) $\in s'.read-lockholders(b)$ then
  - $s.read-lockholders (b) = s'.read-lockholders (b) - \{\text{parent}(T)\}$

REQUEST-COMMIT($T, v$) with type($T$) = split-write

In the postconditions, a BM for a new bucket $b'$ is constructed with initial states as $S_0$. $S_0.keyset$ is the initial keyset of bucket $b'$ which contains the keys on the basis of the hash function based on the lev (local-level) value of bucket $b$. $S_0.created$ and $S_0.run$ get values from $s'.created$ and $s'.run$ for transactions that access keys in $S_0.keyset$

Precondition:  
- $T \in \text{accesses (} b \text{)}$
- $T \in s'.created \implies \neg s'.run$
- $b$ is the bucket to be split
- keys $\in s'.keyset$
- $T \in s'.sel-lockholders(b)$
- $\text{putchain (} T \text{)} = \text{next} + N \times 2 \times \# \text{level}$
Postcondition: $S_0.\text{map} (T', k') = S_0.\text{map} (T', k')$ for all $S_0.\text{keyset}$ and for all $T'$
$S_0.\text{mapsize} (T') = \text{number of non-nil keys in } S_0.\text{map} (T')$ for all $T'$
$S_0.\text{run} = S_0.\text{run} \cup \{T\}$
$S_0.\text{lev} = S_0.\text{lev} + 1$

The following are the changes in a state $s$ of the bucket manager BM for the bucket $b$.

$s.\text{keyset} = s'.\text{keyset} - S_0.\text{keyset}$
$s.\text{created} = s'.\text{created} \forall T \in s.\text{keyset}$,
$s.\text{run} = s'.\text{run} \forall T \in s.\text{keyset}$
$s.\text{map} (T', k') = s'.\text{map} (T', k')$ for all $k' \in s.\text{keyset}$
$s.\text{mapsize} (T') = \text{number of non-nil keys in } s.\text{map}(T', k)$ for all $T'$
$s.\text{lev} = s'.\text{lev} + 1$
$r(1) = \text{oever}$

**INFORM-COMMIT-AT-(b) OF $T$**

The only precondition says that $b$ is the bucket accessed by $T$. The postconditions say that the respective locks are released to the parent of the transaction $T$ on commit.

Precondition: $b$ is the bucket containing key $k$ or is a split or intermediate bucket

Postcondition:
If $T \in s'.\text{sel-lockholders} (b)$ then $s.\text{sel-lockholders} (b) = s'.\text{sel-lockholders} (b) - \{T\} \cup \{\text{parent}(T)\}$
If $T \in s'.\text{read-lockholders} (b)$ then
$s.\text{read-lockholders} (b) = s'.\text{read-lockholders} (b) - \{T\} \cup \{\text{parent}(T)\}$
If $T \in s'.\text{write-lock} (k)$ then $s.\text{write-lock} (k) = s'.\text{write-lock} (k) - \{T\} \cup \{\text{parent}(T)\}$
If $T \in s'.\text{read-lockholders} (k)$ then
$s.\text{read-lockholders} (k) = s'.\text{read-lockholders} (k) - \{T\} \cup \{\text{parent}(T)\}$

**INFORM-ABORT-AT-(b) OF $T$**

The postconditions assert that when a transaction aborts, the locks held by all its descendants are released.

Precondition: $b$ is the bucket containing the key $k$ or is a split or intermediate bucket

Postcondition
$s.\text{sel-lockholders}(b) = s'.\text{sel-lockholders} (b) - \text{descendants} (T)$
$s.\text{read-lockholders} (b) = s'.\text{read-lockholders} (b) - \text{descendants} (T)$
$s.\text{write-lockholders} (k) = s'.\text{write-lockholders} (k) - \text{descendants} (T)$
$s.\text{read-lockholders} (k) = s'.\text{read-lockholders} (k) - \text{descendants} (T)$
$s.\text{map} (U) = s'.\text{map} (U)$ for all $U \in s.\text{write-lockholders} (k)$
6. Link manager automaton

The merge-write subtransactions access the dynamic link manager automaton corresponding to the two bucket manager automata to be merged. The states of the link manager automaton are a crossproduct of the states of the two respective bucket manager automata. The assertion \( s . \text{created} = s'.\text{created} \cup \{ T \} \), where \( s \) and \( s' \) are the states of the link manager automaton, has the following interpretation: If \( s = (S_1, S_2) \) and \( s' = (S'_1, S'_2) \) then \( S'_1.\text{created} = S_1.\text{created} \cup \{ T \} \) and \( S'_2.\text{created} = S_2.\text{created} \cup T \). Other components in a state of a link manager automaton are derived similarly. All the state components used are as defined earlier. accesses\((b, b')\) is the set of transactions accessing the link manager automaton of buckets \( b \) and \( b' \), and excl-lockholders\((b, b')\) is the set of transactions holding an excl-lock on the link manager automaton of buckets \( b \) and \( b' \).

**CREATE**\((T)\)

The precondition says that \( T \) should be an access to both the merging buckets.

**Precondition:** \( T \in \text{accesses}(b, b') \)

**Postcondition:** \( s.\text{created} = s'.\text{created} \cup \{ T \} \)

**OBTAIN-LOCKTYPE**\((T)\) where \( \text{type}(T) = \text{merge-write} \)

The precondition is that \( T \) is created and an exclusive-lock on the pair \((b, b')\) must be with ancestors of \( T \). The postcondition assumes that an excl-lock is granted to \( T \) on the pair \((b, b')\).

**Precondition:**
\[
T \in s'.\text{created} \\
s'.\text{excl-lockholders} \cup \text{ancestors}(T)
\]

**Postcondition:**
\[
s.\text{excl-lockholders} \cup \text{ancestors}(b, b') = s'.\text{excl-lockholders} \cup \text{ancestors}(T)
\]

**REQUEST-COMMIT**\((T, v)\) with \( \text{type}(T) = \text{merge-write} \) and \( T \in \text{accesses}(b, b') \)

The preconditions following first say that the keys belong to keyset in state \( s' \) and an excl-lock is held by \( T \) on the pair of buckets \((b, b')\). The components not appearing in postconditions with respect to state \( s \) remain same in pre- and post-conditions. In the postconditions, \( S_1 \) and \( S_2 \) are the states of the bucket manager automata for \( b \) and \( b' \), respectively, after REQUEST-COMMIT.

**Precondition:**
\[
T \in s'.\text{created} - s'.\text{run} \\
\text{keys} \in s'.\text{keyset} \\
T \in s'.\text{excl-lockholders} \cup (b, b')
\]

**Postcondition:**
\[
s.\text{run} = s'.\text{run} \cup \{ T \} \\
\text{changes to the bucket manager BM for bucket } b \\
s_1.\text{keyset} = s'_1.\text{keyset} \cup s'_2.\text{keyset} \\
s_1.\text{created} = s'_1.\text{created} \cup s'_2.\text{created} \\
s_1.\text{map} \cup (T', k') = s'_1.\text{map} \cup (T', k') \text{ for all keys } k' \text{ in } b \\
s_1.\text{map} \cup (T', k'') = s'_2.\text{map} \cup (T', k'') \text{ for all keys } k'' \text{ in } b' \\
s_1.\text{mapsize} = s'_1.\text{mapsize} \cup s'_2.\text{mapsize} \\
s_1.\text{run} = s'_1.\text{run} \cup s'_2.\text{run}
\]
\[ s_1.\text{excl-lockholders} (b) = s_1.\text{excl-lockholders} (b) \cup \{T\} \]
\[ r(1) = \text{`merge'} \]

Changes at the deallocated bucket
All the variables in the state \( s \) will be nil.

**INFORM-COMMIT-AT-(b, b') OF T**

**Precondition:** \((b, b') \) is the pair of buckets merged

**Postcondition:** \( s.\text{excl-lockholders} (b, b') = s'.\text{excl-lockholders} (b, b') - T \cup \{\text{parent}(T)\} \)

**INFORM-ABORT-AT-(b, b') OF T**

**Precondition:** \((b, b') \) is the bucket pair

**Postcondition:** \( s.\text{excl-lockholders} (b, b') = s'.\text{excl-lockholders} (b, b') - \text{descendants} (T) \)
\( s.\text{read-lockholders} (k) = s'.\text{read-lockholders} (k) - \text{descendants} (T) \) for all \( k \) in \((b, b')\)
\( s.\text{write-lockholders}(k) = s'.\text{write-lockholders}(k) - \text{descendants} (T) \) for all \( k \) in \((b, b')\)

7. Basic properties

We begin with some definitions, and state and prove some properties about the states of transactions deducible from the bucket manager automaton.

First, we state the following result (without proof) similar to the lemma given in [23].

**Lemma 1.** Let \( \gamma \) be a schedule of the BM for the bucket \( b \) then \( \gamma \) is well-formed.

**Definition 1.** Let \( \gamma \) be a sequence of operations of BM for the bucket \( b \), and \( T \) is an access to \( b \) with \( T' \) being an ancestor of \( T \). We say that \( T \) is committed at \( b \) to \( T' \) in \( \gamma \), if \( \gamma \) contains a subsequence \( \gamma' \) consisting of an INFORM-COMMIT-AT-(b) of \( U \) event for every \( U \) that is an ancestor of \( T \) and a proper descendant of \( T' \), arranged in the ascending order so that the INFORM-COMMIT for parent \( (U) \) is preceded by that for \( U \).

**Definition 2.** Let \( \gamma \) be a well-formed sequence of operations of BM, \( T \) be an access to \( b \), and \( T' \) be any transaction. We say that \( T \) is visible at \( b \) to \( T' \) in \( \gamma \) if \( T \) is committed at \( b \) to the least-common-ancestor \((T, T')\) denoted as lca \((T, T')\). We denote visible\( _b \) \((\gamma, T)\), the subsequence of \( \gamma \) that consists of operations of \( b \) whose transactions are visible at \( b \) to \( T \) in \( \gamma \), visible\( _b \) \((\gamma, T)\) is a well-formed sequence of operations of the bucket \( b \).

**Definition 3.** A transaction \( T \) is an orphan at \( b \) in \( \gamma \) if INFORM-ABORT-AT-(b) of \( U \) occurs in \( \gamma \) for some ancestor \( U \) of \( T \).
Definition 4. Given any sequence of operations $\gamma$ of $b$, we define $\text{write}(\gamma)$ to be the subsequence consisting of the $\text{REQUEST-COMMIT}(T, v)s$ for the write accesses $T$.

Definition 5. Given any well-formed sequence $\gamma$ of operations of BM for the bucket $b$, let $\text{essence}(\gamma)$ denote the sequence obtained from $\text{write}(\gamma)$ by placing a $\text{CREATE}(U)$ event immediately preceding a $\text{REQUEST-COMMIT}(U, u)$ event. Since $\gamma$ is well-formed, $\text{essence}(\gamma)$ consists of a subsequence of events of $\gamma$ and is well-formed. Clearly, $\gamma$ and $\text{essence}(\gamma)$ are write-equal (both schedule produces the same value).

To begin with, we have a fundamental invariant of the state of the BM which expresses the fact that the conflicting locks are never held by the transactions except that when one transaction is an ancestor of the other. This condition is enforced when locks are granted, and preserved thereafter by all the actions.

Property 1. Let $s$ be a state of BM for the bucket $b$.

(a) If $T' \in s.sel\cdot\text{lockholders}(b) \cup s.excl - \text{lockholders}(b) \cup s.write - \text{lockholders}(k) \cup s.read - \text{lockholders}(k)$ and $T \in s.sel\cdot\text{lockholders}(b) \cup s.excl - \text{lockholders}(b) \cup s.write - \text{lockholders}(k)$, or

(b) if $T' \in s.sel - \text{lockholders}(b) \cup s.excl - \text{lockholders}(b) \cup s.read - \text{lockholders}(b)$ and $T \in s.sel - \text{lockholders}(b) \cup s.excl - \text{lockholders}(b) \cup s.write - \text{lockholders}(k)$, then either $T$ is an ancestor of $T'$ or else $T'$ is an ancestor of $T$.

Proof. 1(a). Recall that the sel-lock, excl-lock and write-lock conflicts with each other and with other similar locks. Also, the excl-lock and the write-lock conflict with read-locks. Therefore, if $T \in s.sel\cdot\text{lockholders}(b) \cup s.excl\cdot\text{lockholders}(b) \cup s.write\cdot\text{lockholders}(k)$ and $T'$ is the holder of conflicting locks then $T'$ must be an ancestor of $T$ and vice versa. This is because before granting a lock, say to $T$, all the holders of conflicting locks (say $T'$) must be the ancestors of $T$ according to the locking rules [27]. Similarly, 1(b) also holds. $\square$

Property 2. Let $s$ be a state of BM for the bucket $b$. Then the following holds.

(a) The union over all $k$ of $s.write\cdot\text{lockholders}(k)$ is a subset of $s.sel\cdot\text{lockholders}(b)$.

(b) The union over all $k$ of $s.read\cdot\text{lockholders}(k)$ is a subset of $s.sel\cdot\text{lockholders}(b)$.

Proof. 2(a). Before obtaining the write-lock on key $k \in b$, a sel-lock must have been obtained on bucket $b$ according to the locking rules. The sel-lock is obtained in state $s$ by the $\text{OBTAINED-LOCKTYPE}$ operation of the automaton. Therefore, the union of all transactions accessing some key $k \in b$ must be a subset of $s.sel\cdot\text{lockholders}$. By similar arguments, 2(b) also holds. $\square$

The following property shows the transactions which hold locks after a schedule of BM for the bucket $b$.

Property 3. Let $\gamma$ be a generic object well-formed schedule of BM for the bucket $b$. Let $s$ be a state of the BM. Suppose that $\gamma$ can leave the BM in state $s$. Let $T$ be an access to the BM such that $\text{REQUEST-COMMIT}(T, v)$ occurs in $\gamma$ and $T$ is not an orphan at $b$ in $\gamma$ and let $T'$ be the highest ancestor of $T$ such that $T$ is committed at $b$ to $T'$ in $\gamma$. Then the following holds:
(a) If $T$ is a search-write access then $T'$ must be a member of $s$.sel-lockholders$(b) \cap s$.write-lockholders $(k)$ where $b$ is not the intermediate bucket. If $b$ is the intermediate bucket then $T \in b$.sel-lockholders$(b)$.

(b) If $T$ is a search-read access then $T'$ must be a member of $s$.read-lockholders$(b) \cap s$.read-lockholders$(k)$ where $b$ is not the intermediate bucket. If $b$ is the intermediate bucket then $T \in b$.read-lockholders$(b)$.

(c) If $T$ is a search-split then $T'$ must be a member of $s$.sel-lockholders.$$

(d) If $T$ is a merge-access to $(b, b')$ then $T'$ must be a member of $s$.excl-lockholders$(b, b')$. (In this case $T$ is an access to $LM$.)

**Proof.** 3(a). Since REQUEST-COMMIT$(T,v)$ occurs in $\gamma$ and $T'$ is the highest ancestor of $T$ such that $T$ is committed at $b$ to $T'$ in $\gamma$, therefore according to the locking rules, $T'$ must have inherited the locks from the search-write access $T$ on its commit. There are two cases:

1. $T$ is a search-write access to the bucket $b$ such that $k \in b$.
2. $T$ is a search-write access to the bucket $b$ such that $k \not\in b$.

In case 1, $T$ must have obtained a sel-lock on $b$ during OBTAIN-LOCKTYPE operation before getting the write-lock on key $k$ in state $s$ of REQUEST-COMMIT$(T,v)$. On commit, $T$ releases all its locks to the highest ancestor $T'$. In case 2, if $b$ is the intermediate bucket then $T$ must have obtained a sel-lock on $b$. Therefore, on commit, $T$ releases its lock to the highest ancestor $T'$. Similarly 3(b)-(d) also hold. $\square$

The following property [23] shows that when an access $T'$ occurs, all prior conflicting accesses must either be local orphans or visible to $T'$.

**Property 4.** Let $\gamma$ be a generic object well-formed schedule of a BM. Suppose distinct events $\pi = \text{REQUEST-COMMIT}(T,v)$ and $\pi' = \text{REQUEST-COMMIT}(T',v)$ occur in $\gamma$ where $T$ and $T'$ conflict. If $\pi$ precedes $\pi'$ in $\gamma$ then either $T$ is local orphan in $\gamma'$ or $T$ is visible to $T'$ in $\gamma'$, $\gamma'$ is the prefix of $\gamma$ preceding $\pi'$.

The following property characterizes the write-buffer component of the state and shows that the write-buffer$(T)$ reflects the effects of all transactions that are visible to $T$ [23].

**Property 5.** Let $\gamma$ be a well-formed schedule of BM for the bucket $b$. Let $s$ be a state of BM for the bucket $b$. If $T$ is a transaction that is not an orphan at $b$ in $\gamma$, then $\delta = \text{essence}_{b}(\gamma, T)$ is a well-formed schedule of the basic object $b$. Furthermore, when $\delta$ is applied to the bucket manager automaton BM for $b$ in its initial state, it can leave BM in the state $s$.map($T', k$) where $T'$ is the least ancestor of $T$ such that $T' \in s$.sel-lockholders $\cap s$.write-lockholders $(k)$.

7.1. A search property of system $B$

**Assertion 1.** A search access for a key $k$ in the system $B$ always leads to the correct bucket $b$ which contains the key $k$.

**Proof.** A good state of a search structure is as defined in [14,15]. Here, we first give a variant of that definition for linear hash structure. A good state of a linear hash structure satisfies the
condition that each key is contained in exactly one bucket. The necessary conditions guaranteeing successful search are as follows:

1. Each search structure begins in a good state and each operation of the type search-read, search-write, split-write and merge-write maps a good state to another good state. This condition implies that when a search for a key starts from the initial bucket through the initial hash function and no other operation interferes with the search, then it always leads to the correct bucket.

2. No operation reduces the set of keys in a bucket reachable from an another bucket that is not deleted. This condition enables the continuation of search after being interrupted by other operations like splitting without having to backtrack the search. We will see that the additional information of local-level at each bucket ensures that this condition is satisfied during splitting.

3. No transaction accesses a deleted bucket.
   Conditions 1 and 2 are similar to those stated in [14,15] for the general link techniques. Condition 3 is an additional requirement when concurrent merging operations are considered. Merging of two buckets results in the deletion of one bucket after copying the contents of the deleted bucket to the other bucket. If merging is an atomic action, then it maps a good state to a good state. Conditions 2 and 3 ensure that when search for the key is interrupted by splitting or merging, the search will still succeed in finding the key. No search will ever need to backtrack.

To show that Condition 1 is satisfied in system B, we note that the initial state of each hash structure must be a good state. Also the only operations that affect the location of keys are splitting and merging operations. A splitting operation (split-TM) distributes some keys in a bucket \( b \) to a new bucket reachable from bucket \( b \) using the local-level value at \( b \) with the help of a split-write access subtransaction. The merging operation (merge-TM) moves all the keys of one bucket to another bucket and all those keys are reachable using the local-level value at bucket \( b \) with the help of the merge-write access subtransactions. A split-write and a merge-write do not disturb other search processes. This is because once a search-write access acquires a sel-lock on bucket \( b \), the split-write and the merge-write access subtransactions cannot get locks on \( b \). In case a search-read access holds a read-lock on bucket \( b \), a merge-write cannot get an excl-lock. However, a split-write can get a sel-lock on \( b \) while the search-read holds a read-lock on \( b \) and still the search-read can reach key \( k \) with the help of local-level value. This is because the bucket at the new address is written using the hash function before the new version replaces the contents of the bucket \( b \). If the local-level value at \( b \) is not changed then the search-read can still find the key \( k \) at \( b \) else it will find the key \( k \) at the new bucket. Hence, a search process always leads to the correct bucket.

Condition 2 is satisfied because no operation reduces the set of keys reachable using the local-level value from an undeleted bucket \( b \). A splitting process only reduces the keyset of a bucket \( b \), but the keys removed from \( b \) are still reachable from \( b \) using the local-level value at \( b \).

Condition 3 is satisfied because the only operation which may delete a bucket is merge. A merge operation copies all the keys from a bucket and appends them to another bucket \( b \) and updates (decreases) the local-level value at the bucket \( b \). Hence, no transaction can access the deleted bucket once the local-level value of the bucket is decreased.
8. Direct data locking objects

Let $k$ be any key in $H$. The set $D = \{k; k \in H\}$ is a database where a key $k$ is directly accessible. Each object in $D$ is called a direct data locking (DDL) object. The generalized Moss's two phase locking algorithm [6] is used on the data objects in $D$. This direct data locking system is denoted as system $A$.

The nested transaction system $A = (\tau_4, \text{parent}_4, O_4, V_4)$ has the following structure. In system $A$, all the non-access transactions are user-visible. The nested transaction tree of system $A$ has access subtransactions at the leaf node and the user-visible transactions at one level above access transactions. parent$_4$ is a mapping which gives the parent-child relationships in the transaction tree. $O_4$ is the set of logical data objects which partitions the set of accesses such that each partition contains accesses to one particular data object. All the data objects of system $A$ are user-visible; that is, they are direct data objects (keys $k$ here) without any in-build search structure.

In the system $A$, each transaction and the DDL object represent an automaton which communicates with the scheduler. The scheduler controls the order to invoke child transactions or access objects.

The system $A$ is the composition of a set of I/O automata and is similar to the $R-W$ locking system of [6]. The serial correctness of system $A$ follows from [6].

9. Correctness

The correctness of our algorithm has a two tier structure. First, we will observe that the system $B$ in a nested transaction environment is a structural extension of the system $A$. Also, the concurrent system $B$ is correct with respect to concurrent system $A$, i.e., a user-visible transaction cannot distinguish between the systems $A$ and $B$. That is, the user-visible transactions have the same schedule in both systems $A$ and $B$ and values associated with the keys observed by them are same.

Having proved that, we infer from the previous result that the system $A$ is serially correct. Hence, the serial correctness of the system $B$ follows from the transitivity of systems.

9.1. Correctness of system $B$

In this section, we compare the system $B$ with the system $A$. In both $A$ and $B$, the transaction tree structures up to the level of user-visible transactions are the same. In $A$, all non-access transactions are the user-visible transactions. Also, if $T$ is a user-visible transaction in $B$ then there is a corresponding user-visible transaction in $A$. Each non-access transaction is represented by the same automaton in both $A$ and $B$. If $T$ is a read-TM (write-TM) in system $B$ then there is a corresponding read access (write access) transaction in system $A$. If $T$ is a split-TM or merge-TM in $B$ then there is no corresponding transaction in system $A$. Also, there are no transactions in $A$ which correspond to the search-write, search-read, merge-write, split-write and release access transactions of system $B$. The objects $k \in b$ will be the direct access read-write objects $o(k), k \in H$ in system $A$, i.e., there is no search structure in $A$. By comparing two systems, it follows that systems $A$ and $B$ are such that the sets $\tau_B$ and $O_B$ contain $\tau_A$ and $O_A$, respectively, $V_B = V_A$, and parent$_B$, restricted to $\tau_A = \text{parent}_A$. That is, the system $B$ is a structural extension of the system $A$. 
Consider a schedule $\beta$ of $B$ consisting of operations corresponding to user-visible transactions, user-invisible transactions, TMs, search-read, search-write, split-write and merge-write, release access transactions. We construct a sequence of operations $\alpha$ of system $A$ by

1. removing from $\beta$ all the REQUEST-CREATE($T$), CREATE($T$), REQUEST-COMMIT($T$, $v$), REPORT-COMMIT($T$, $v$) and REPORT-ABORT($T$), INFORM-COMMIT-AT-(b) OF $T$, INFORM-ABORT-AT-(b) OF $T$, INFORM-COMMIT-AT-(b, b') OF $T$, INFORM-ABORT-AT-(b, b') OF $T$, ABORT($T$), CREATE($T$), OBTAIN-LOCKTYPE($T$) operations for all transactions of the type split-TM, merge-TM, search-read, search-write, split-write, merge-write and release access;

2. by interpreting $T$; a user-visible transaction in $B$, to stand for the corresponding user-visible transaction in $A$;

3. replacing the read/write-TM operations by equivalent read/write access operations.

Note that the equivalent user-visible transactions in the constructed sequence of operations $\alpha$ of $A$ and in the schedule $\beta$ of $B$ have the same order. Therefore, in order to establish the correctness of the serial system $B$ with respect to the system $A$, we have to only show that $\alpha$ is indeed a schedule of the system $A$, and that the read accesses in $\alpha$ return the same values as those returned by the corresponding read-TMs in $\beta$.

The proof of this is by induction on the length of $\beta$.

**Induction hypothesis:** If $\beta$ is a schedule for system $B$, then the sequence of operations $\alpha$ obtained by the above construction is a schedule of $A$ and that the read accesses in $\alpha$ returns the same values as those returned by their counterparts, read-TMs in $\beta$.

**Base case:** Consider $\beta$ to be the empty schedule. Therefore, $\alpha$ is also empty and hence, $\alpha$ is a schedule of $A$.

**Induction step:** Assume that the hypothesis is true for all schedules of length $l$ or less of the system $B$. Let $\beta'$ be a schedule of length $l$, and let $\alpha'$ be the equivalent schedule of $A$. Let $\beta = \beta' \pi_\beta$. There are five cases depending upon the type of operation $\pi_\beta$ is.

**Case 1.** $\pi_\beta$ is an I/O operation REQUEST-CREATE($T$), CREATE($T$), REQUEST-COMMIT($T$, $v$), REPORT-COMMIT($T$, $v$), REPORT-ABORT($T$), FOR search-read, search-write, split-write, merge-write, release accesses or split-TM or merge-TM.

**Proof of Case 1.** In this case, by construction, there is no operation in $\alpha$ which corresponds to $\pi_\beta$ and therefore, $\alpha = \alpha'$ is a schedule of $A$. \qed

**Case 2.** $\pi_\beta$ is an output operation of a TM.

**Proof of Case 2.** If $\pi_\beta$ is a REQUEST-COMMIT($T$, $v$) operation for some $T \in \text{tm}(H)$ (set of transaction managers for non-access transactions) and $k \in b \in H$, then by construction the corresponding operation in $A$ is the same as $\pi_\beta$. Therefore, by the definition of $A$, there is an access (transaction) to a direct read-write object $o(k) \in H$ which corresponds to the transaction $T$. Since $\beta$ is a well-formed, CREATE($T$) occurs in $\alpha'$. Therefore, the precondition for REQUEST-COMMIT($T$, $v'$) is satisfied in $A$ for some $v'$. We need to show that $v = v'$ if $T$ is a write-TM or read-TM. If $T$ is a write-TM, then $v = v' = \text{nil}$. Also in $\beta'$, the write-TM has subtransactions that search through the bucket and finally perform the write operation. As shown in Assertion 1, a search process always leads to the vertex that contains the key $k$. Now to show
that \( v = v' \) if \( T \) is a read-TM. By the construction, Assertion 1 and Properties 1 and 2, the read operation reads the value seen by the least common ancestor of the search-read access that holds a sel-lock on the bucket \( b \) and a write-lock on the key \( k \in b \). Therefore, \( \alpha' \) has seen the same value as seen by the least ancestor of the read-TM in \( \beta' \), and the return value of the read operation in \( \alpha' \) and \( \beta' \) is \( s.map(\text{least}(s.write-lockholders(k))) \) which is the value seen by the least ancestor of the read transaction among the transactions that hold the write-lock on the key \( k \in b \). Therefore, we have \( \alpha'| o(k), k \in H \) (a subsequence of \( \alpha \) consisting of operations of \( o(k), k \in H \) only) = the sequence of operations in \( \beta' \) on the key \( k \in b \in H \). Therefore, the last write access in \( \alpha' \) on \( o(k), k \in H \), has the same value as the last write-TM on \( k \in b \in H \) in \( \beta' \). Hence the value associated with the state of \( o(k), k \in H \), after \( \alpha' \) is logical value after \( \beta' \). Therefore, we have \( v = v' \).

If there is no write-TM invoked in \( \beta' \), there are no write accesses to \( o(k), k \in H \), in \( \alpha' \). Therefore, the value \( v \) of \( o(k), k \in H \), after \( \alpha' \) is the initial value which is the logical value after \( \beta' \). Similar reasoning holds for all the keys \( k \in H \) in system \( B \).

Hence, the read-TM returns the same value of the key \( k \) in \( B \) as those returned by corresponding read access in \( A \). \( \square \)

**Case 3.** \( \pi_B \) is an output operation of a user-visible transaction.

**Proof of Case 3.** If \( \pi_B \) is an output operation of some user-visible transaction \( T \), then by construction the corresponding operation in \( A \) is same as \( \pi_B \). By construction, the user-visible transaction \( T \) in \( B \) and the corresponding transaction \( T' \) in \( A \) are modeled by the same automaton. By the induction hypothesis, \( \alpha'| T' = \beta'| T \). Therefore, since the preconditions for \( \pi_B \) associated with the state of the user-visible transaction \( T \) are satisfied after \( \beta' \), they must also satisfied in the associated state of corresponding \( T' \) after \( \alpha' \). Therefore, \( \alpha'| T' \) is a schedule of \( T' \) and by Composition Lemma 1 \([23]\), \( \alpha \) is a schedule of \( A \). \( \square \)

**Case 4.** \( \pi_B \) is an output operation of the scheduler (except those already covered by Case 1).

**Proof of Case 4.** If \( \pi_B \) is a \textsc{Create}(T), \textsc{Commit}(T), \textsc{Abort}(T), \textsc{Inform-Commit-At}(b) \textsc{Of} \; T, \textsc{Inform-Abort-At}(b) \textsc{Of} \; T, \textsc{Obtain-LockType}(T) \) where \( T \) is a user-visible transaction, or \( T \) is read-TM or write-TM for some data object \( k \in b \in H \). Then by construction, the corresponding operation in \( A \) is same as \( \pi_B \). Using the fact that the schedule \( \beta \) is well-formed and by construction, the preconditions for \( \pi_B \) are satisfied for all these output operations (details are omitted). Therefore, \( \alpha \) is a schedule of \( A \).

Hence, in all the cases, \( \alpha \) is a schedule of the system \( A \) and that the read-TMs of \( B \) returns the same value of all the keys as those returned by equivalent read accesses of \( A \).

Thus, we have shown that the system \( B \) is correct by showing that it is the same as \( A \) in terms of user-visible transactions.

The system \( A \) is equivalent to \( R/W \) locking objects given in [6]. A proof of serial correctness of \( R/W \) locking objects as given in [6] states that every schedule of the \( R/W \) locking objects is serially correct for every non-orphan, non-access transactions. Therefore, the system \( A \) is serially correct and hence, the system \( B \) is correct by transitivity of systems.
10. Implementation

In this section, we briefly discuss the layered system architecture of our model discussed in the previous sections. Our system design is based on the client/server model as shown in Fig. 3. Each module in the layered system is designed and implemented separately. The communications among various components in our system are done via messages.

The transaction automata while implementing are modeled as methods that correspond to nested transactions. These methods are further implemented as multithreads. Multithreading is a programming paradigm that allows the application processes to run concurrently. When the process performs multiple tasks at the same time, multithreading split themselves into separate threads that run concurrently and independently. Each thread performs one atomic action. The threads execute individually and are unaware of the other threads in a process. Thus, the multithreading can be used very efficiently to implement the behavior of the nested transactions.

Fig. 3. Layered architecture for client/server model.
Finally, we have designed and implemented our system using layered system architecture in a client/server environment, which allows more flexibility in terms of decomposing of the application programs whose modules may be designed and implemented independently.

The architecture of our model consists of six layers distributed between the clients and the server. Each client site has one layer called user interface (UI) layer. The other layers are implemented at the server site. These layers are ordered as operation layer, transaction manager (TM) layer, scheduler layer, bucket manager (BM) layer, and finally file manager (FM) layer, respectively. All the layers at the client and the server site work in chronological order to achieve a transaction’s computation.

The purpose of this layering system is to partition a transaction into the nested transactions. At each level, the subtransactions can operate concurrently without leading to an inconsistency and hence a higher concurrency can be achieved in our system. A very important feature of our layered system architecture is that if a subtransaction fails at any level; the parent at a higher layer level can initiate another subtransaction without aborting the entire transaction.

UI layer: This is the only layer that has been implemented at the client site. It works as an interface between the users and the server. UI layer’s task is to receive an operation request from the user and pass it to the operation layer at the server. In UI layer, two components exist. They are user interface operations (i.e., find, insert, and delete) and the agent. When a user initiates one of the operations, the agent will pass the message containing the key requested by the user to the appropriate operation at the server site. At the end of computations at the server site, the agent will receive a message from the server site which could be either ‘record is found’, ‘record does not exist’, or confirmation messages.

Operation layer: This layer is considered to be the main layer in the server as it creates the transactions depending on the operation and pass them to the appropriate TMs in the next layer. The operations it contains are find, insert, and delete operations. These operations will receive the corresponding messages from the TM layer. Once an operation is executed, it passes an appropriate message to the client to indicate an operation’s completion. In case the message received from the TM layer is an ‘overflow’ or ‘underflow’ (as a result of insert or delete), the split and merge will be called, respectively. If the message is ‘overflow’, the insert operation will initiate a split-TM to complete the operation. If the message is ‘underflow’ then the delete operation will initiate a merge-TM to complete the operation.

TM layer: This layer receives the appropriate messages from the operation layer to initiate a TM. The TM layer contains the four types of TMs; read-TM, write-TM, split-TM, and merge-TM as explained before. Each TM creates subtransactions to complete the requested operation. TMs work concurrently and independently. After a TM completes its execution, it reports back the results of the operation to the operation layer. Transactions created at this layer will be passed to the scheduler at the next layer, which synchronises the various subtransactions created by TMs. The subtransactions created by TMs work concurrently and only the leaf level subtransactions will access the buckets directly.

Scheduler layer: At this layer we have a scheduler that synchronises all the transactions. The scheduler will determine when to create, commit, and abort a transaction. The scheduler will send the synchronized subtransactions to the BM layer to access the buckets. When a transaction completes, the scheduler will report back to the TMs the state of each transaction whether it is successfully created, committed or aborted.
**BM Layer.** At this layer, each BM handles a bucket and its entire overflow buckets. Before a transaction accesses any bucket, the corresponding BM will determine whether to permit the transaction to access the bucket or suspend it with the help of the lock compatibility-matrix shown in Fig. 1. If two transactions accessing the same bucket interfere with each other, BM will permit only one of the transactions to access the bucket while suspending the other until previous transaction finishes. Thus, BMs will restrict accesses to the buckets that lead to an inconsistent database state. To insure serial-correctness, BM also handles the keys' locks within the bucket chain. The subtransactions, which access the BMs, are called decisive subtransactions as they are the only subtransactions that access the buckets directly and any interference between transactions at this layer will lead to an inconsistency. BMs work independently of each other.

**FM layer.** This is the last layer in our system. FM layer will insure the durability of the transaction. Any updates on the buckets will be stored in the database file (i.e., DB File in Fig. 3) which resides on a permanent disk space. Updates on the database file will be performed after the commit of the top-level transaction to insure the consistency of our database file. FM is also responsible for loading the keys from the database file to the memory pages (i.e., buckets). It can also restore the current database state in case of a system crash.

We have implemented our layered system architecture discussed here using object-oriented methodology. We have omitted further details here and readers are referred to [31] for complete details.

10.1. Object-oriented design

Multithreads have been used to implement the nested transactions in our system. Multithreads are managed in such a way that they do not interfere with each other. Life cycle of a thread is hidden from the clients. During a thread's life cycle, many multithreads may be created. A thread may be halted, or an inconsistency may occur, however, these situations will not affect the clients. For the clients, a creation and a commit of the top-level thread is important. As the clients operate concurrently, many multithreads will be created. To control these threads from interfering, the locking scheme discussed earlier is implemented. The operations in our model will require multithreads to complete their jobs such as searching for the object, inserting, deleting, etc.

We have implemented our layered system architecture using object-oriented methodology in the client–server environment. The server side in our implementation consists of eight components namely, server, agent, server socket, FM, user operations, socket file, BM, and LH operations. The client side in our implementation consists of four main components (classes), namely, client, client socket, agent, and socket file. These components are implemented as classes using Microsoft Foundation Classes (MFC). Below, we discuss the scenario of some of the classes at the server side only. Client side classes are similar, therefore we omit the details from here. More details on the implementation of transaction managers, bucket managers and the classes can be found in [28].

**Agent:** A `Agent` class is responsible for creating a new thread for each operation received from the client through the socket file. An agent starts by calling `FM` class to load the DB file before enabling windows sockets. After loading the DB file, the agent will call the server socket class to create the windows sockets. When connecting a client to the server, the agent will be ready to receive a user request. It then creates a thread that will be passed to the user operations class to perform the requested operation.
**File manager:** When instantiating this class, the DB file will be loaded into memory so that data can be accessible any time. Three main member functions exist in this class. These functions are: Store, Load, and Update. Store function stores a key from memory to the DB file. Load function fetches a key from the DB file to the memory. An update function is called whenever an update is occurred to the DB file. FM sends and receives messages, containing the target key, to the TM class as this class is directly connected to the buckets.

**User operations:** User operations class contains the three main operations for database information retrieval. These operations are: insert, delete and find. These operations initiated when the user operations class receives a thread from the agent. Multithreads work concurrently at this class. Each thread is responsible for performing an atomic task. Before the three operations insert, find, and delete pass the message to the TM to perform the required operation, they will call the BM class to set a lock on the target bucket before accessing it. When reading or writing a key, BM will be called again to set a lock for the key.

If an insert operation is initiated, it will pass a message to the TM class to perform an insert operation. If after the insertion, the user operation class receives an 'overflow' message from the TM class, the insert function will send another message to the TM class to perform a split operation. In case an 'underflow' message is received, the delete function will send a message to the TM class to perform a merge operation. Each time the user operations class call TM, it will call BM first to set a lock depending on the operation.

**Bucket manager:** BM is responsible for locking and unlocking the buckets and the keys. BM receives a message from the user operations class to set a lock depending on the operation to be performed. After the operation is finished, BM will be called again to unlock the bucket or the key. Locking the bucket and the key afterwards will insure the serializability. Locking on key is implemented using the read/write lock technique in which a key is granted a read-lock while being read, and a write-lock while updating it. The read-lock, selective-lock, and exclusive-lock variables, for locking the buckets, are also declared in this class.

OODBMS may provide locking at the object and/or page level. We choose a physical page level as locking granularity; a bucket in our design. As each LH object has two buckets, holding a lock on the object level will decrease the concurrency among transactions that access the buckets.

**Transaction managers:** These TMs are implemented using methods. The ReadTM method will search for a key in a bucket. The WriteTM method will either insert a key or delete a key depending on the message received. The SplitTM method will split the next bucket and will send a message to the user operations class to create a new object if needed for the new bucket. The MergeTM method will merge the two buckets involved in the merge operation (next and partner buckets), and will send a message to the user operations class to delete the object if it has emptied the buckets.

TMs' methods will communicate with the FM class to load or store keys and to update the file after a write operation. Also, TM class will send messages to the socket file to synchronise data. That is, sending data to the clients. In our model, each LH object contains two buckets. We have considered two buckets in each object rather than one to reduce the system's overhead during the split and merge operations. This is due to the fact that when the first bucket of any object say O1 needs to be split, a new object say O2 will be created with two buckets. However, when the second bucket from the object O1 splits, no new object (bucket) has to be created as the keys from this bucket can move to the second bucket of the object O2. Similarly, no object needs to be deleted
when one of the buckets becomes empty after a merge operation as the other bucket may contain
some data. An object will be deleted to free some memory space, only if both the buckets become
empty. On the other hand, having more buckets in each object will waste memory space. Thus,
having two buckets in an object is a compromise between speediness and space utilization.
Whenever a split operation is called, the agent will read the local-level of the last bucket of the last
object. If the local-level is 0 (which indicates that this bucket has not been accessed yet), the keys
of the split bucket will move to this bucket. Otherwise, a new object will be created and the keys
will move to the first bucket of this object in this object. At each level of the hash function, we
have a maximum of \(2^{\text{level}}\) objects. We find an object id (OID) using the following:

\[
\text{bucket no.} = \text{key mod}(2^{\text{level}} \times N) \\
\text{OID} = \text{bucket no.}/N
\]

where \(N\) is the number of buckets contained in each LH object (in our model, \(N = 2.\))

11. Conclusions

In this paper, we have modeled the linear hash structure algorithm [4] in nested transaction
environment using I/O automaton model. Lock management on root variables, keys and buckets
in a nested transaction environment is provided by Moss's two phase locking algorithm [27] and
the locking algorithm using the lock-coupling given in [4]. We have modeled find, insert, delete,
split and merge operations in a nested transaction environment using I/O automata by specifying
pre- and post-conditions for each of these operations. We have formally proved that our linear
hash structure algorithm using the nested transactions is 'serially correct' in the sense of [23]. The
complete specifications of the conditions are ideal for the system implementation of transaction
processing algorithm and for their verification where data are stored using linear hash structures.
We have implemented this algorithm in client–server environment under windows-NT. The
complete implementation and performance can be found in [28]. We have also removed the
drawbacks of earlier work in B-trees framework [9], within the framework of a linear hashing
model. We do think that the open issue is how to improve on concurrency and re-apply our model
and proof technique to the B-tree case and other hashing techniques. We also plan to study the
pass recovery issues under linear has structure environment.

Appendix A. Write-TM

**CREATE\((T)\) with root\((T) = \text{'initial hash function'}\)**

**REQUEST-CREATE\((T')\) with type\((T') = \text{search-write, key}(T') = k \text{ and data}(T) = d \text{ or nil}**

Precondition:

\[s'.\text{created} = \text{false} \]
\[\text{read-lockholders } (rv) \subseteq \text{ancestors } (T)\]

Postcondition:

\[s.\text{phase} = \text{'searching'}\]
\[s.\text{created} = \text{true}\]
\[s.\text{data} = \text{nil}\]
read-lockholders(rv) = read-lockholders(rv) \cup \{T\}
rv = (level, next)
lev(T) = level
\{exists a sequence h_1, h_2, \ldots h_j \exists
h_1 = root(T)
h_i = h_{lev(T)+j}(k) \forall i \leq j
h_i < next \forall i < j
h_j \geq next
s.nextbucket = h_j\}

Precondition: \ T' \in access (s'.nextbucket)
s'.created = true
s'.phase = 'searching'
k = key(T)
lev(T') = lev(T)

REPORT-COMMIT(T', v) with type (T') = search-write and v = (r(1), r(2))

Precondition: s'.phase = 'searching'

Postcondition: If T \in read-lockholders(rv) then read-lockholders(rv) = read-lockholders(rv) \setminus \{T\} \cup \{parent(T)\}
If r(1) = 'found and underflow' or 'found and overflow' or 'found' then
\{s.phase = 'finished', s.data = r(2)
If s.count = 1 then s.p = true
If r(1) = 'not found' then
\{s.release-bucket_2 = s'.release-bucket_1
s.release-bucket_1 = s'.nextbucket
s.nextbucket = r(2)
lev (T) = lev (T) + 1
s.count = s.count + 1
If s.count \geq 2 then \{s.p = true, s.i = true\}\}
s.write-committed = s'.write-committed \cup \{T'\}

REQUEST-CREATE(T') with type(T') = release access

Precondition: s'.created = true
s'.phase = 'searching' or 'finished'
s'.p = true
s'.i = true
s'.release-bucket_2 \neq nil and r(1) \neq 'found'
T' \in access (s'.release-bucket_2)
last T'' \in access (s'.release-bucket_2) \in s'.write-committed
Precondition: 
\[ s'.\text{created} = \text{true} \]
\[ s'.\text{phase} = \text{‘searching’ or ‘finished’} \]
\[ s'.p = \text{true} \]
\[ s'.i = \text{true} \]
\[ s'.\text{release-bucket}_2 = \text{nil or } r(1) = \text{‘found’} \]
\[ T' \in \text{access } (s'.\text{release-bucket}_1) \]
\[ \text{last } T'' \in \text{access } (s'.\text{release-bucket}_1) \in s'.\text{write-committed} \]

**REPORT-COMMIT**\((T', v)\) with \(\text{type}(T') = \text{release access}\)

Postcondition: 
\[ s.i = \text{false} \]
If \(r(1) = \text{‘found and underflow’ or ‘found and overflow’ or ‘found’ then }\)
\[ s.p = \text{false} \]

**REPORT-ABORT**\((T')\)

Postcondition: 
no change

**REQUEST-COMMIT**\((T, v)\)

Precondition: 
\[ s'.\text{phase} = \text{‘finished’} \]
\[ s'.p = \text{false} \]
\[ v = \text{‘overflow’ or ‘underflow’ or nil} \]

Postcondition: 
\[ s.\text{phase} = \text{‘idle’} \]

**Appendix B. Split-TM**

**CREATE**\((T)\)

Precondition: 
\[ s'.\text{created} = \text{false} \]
\[ \text{sel-lockholders } (rv) \subseteq \text{ancestors}(T) \]

Postcondition: 
\[ s.\text{created} = \text{true} \]
\[ s.\text{phase} = \text{‘split’} \]
\[ \text{sel-lockholders } (rv) = \text{sel-lockholders } (rv) \cup \{T\} \]
\[ rv = (\text{level, next}) \]
\[ s.\text{nextbucket} = \text{next} \]

**REQUEST-CREATE**\((T')\) with \(\text{type}(T') = \text{split-write}\)

Precondition: 
\[ T' \in \text{access } (s'.\text{nextbucket}) \]
\[ s'.\text{phase} = \text{‘split’} \]
\[ \text{keys} \in s'.\text{keyset} \]
\[ \text{putchain}(T') = \text{next} + N \ast 2 \ast \ast \text{level} \]
**REPORT-COMMIT**($T'$, $v$) with type($T'$) = split-write

**Precondition:**
- $s'$.phase = 'split'
- $r(1)$ = over

**Postcondition:**
- $s$.phase = 'finished'
- next = (next + 1) MOD ($N' \times 2 \times \ast$ level)
- If next = 0 then level = level + 1
  - sel-lockholders ($rv$) = sel-lockholders ($rv$) - $\{T\} \cup \{\text{parent}(T)\}$

**REPORT-ABORT**($T'$)

**Precondition:**
- no change

**REQUEST-COMMIT**($T$, $v$)

**Precondition:**
- $s'$.phase = 'finished'
- $v$ = nil

**Postcondition:**
- $s$.phase = 'idle'

---

**Appendix C. Merge-TM**

**CREATE**($T$)

**Precondition:**
- $s'$.created = false
- excl-lockholders ($rv$) $\subseteq$ ancestors($T$)

**Postcondition:**
- $s$.created = true
- $s$.phase = 'merge'
- excl-lockholders ($rv$) = excl-lockholders ($rv$) $\cup$ $\{T\}$
- next = (next - 1) MOD ($N' \times 2 \times \ast$ level)
- If next = 0 then level = level + 1
- $s$.merge-1 = next
- $s$.merge - 2 = next + $N' \times 2 \times \ast$ level

**REQUEST-CREATE**($T'$) with type ($T'$) = merge-write

**Precondition:**
- $T'$ $\in$ access ($s'$.merge-2, $s'$.merge-1)
- $s'$.phase = 'merge'
- keys $\subseteq$ $s'$.keyset

**REPORT-COMMIT**($T'$, $v$) with type ($T'$) = merge-write

**Precondition:**
- $r(1)$ = 'merge'
- next = next - 1
Postcondition: \( s\.phase = \text{‘finished’} \)
\[ \text{excl-lockholders (rv)} = \text{excl-lockholders (rv)} - \{ T \} \cup \{ \text{parent}(T) \} \]

**REPORT-ABORT**\( (T) \)

**Precondition:** no change

**REQUEST-COMMIT**\( (T, v) \)

**Precondition:** \( s’.phase = \text{‘finished’} \)
\[ v = \text{nil} \]

**Postcondition:** \( s\.phase = \text{‘idle’} \)

**References**


Sanjay Kumar Madria received his Ph.D. in Computer Science from Indian Institute of Technology, Delhi, India in 1995. He is an Assistant Professor, Department of Computer Science, at University of Missouri-Rolla, USA. Earlier he was Visiting Assistant Professor in the Department of Computer Science, Purdue University, West Lafayette, USA. He has also held appointments at Nanyang Technological University in Singapore and University Sains Malaysia in Malaysia. His research interests are in the area of nested transaction processing, mobile transaction processing and management of web data. He is a Guest-editor of WWW Journal. He is Program-Chair for EC & WEB 2000 conference held in UK. He is PC member of many international database conferences and reviewers of many database journals.

S.N. Maheshwari received his Ph.D. in Computer Science from Northwestern University, USA. He is currently a Professor of Computer Science and Engineering at the Indian Institute of Technology, Delhi, India, where he has held the appointments of Head, Department of Computer Science and Engineering, and Dean, Undergraduate Studies. His research interests range from graph algorithms, computational geometry, distributed and parallel computing, to theory of transaction processing.
B. Chandra is a Professor of Computer Applications at the Indian Institute of Technology, Delhi, for the past two decades. Her area of specializations includes Distributed Databases with special reference to recovery, knowledge acquisition using Neural Networks. She has published a number of research papers in reputed journals in these two areas. She has held visiting faculty positions at the University of Pittsburgh, USA and the Penn State University, USA. She has also held positions at the World Bank at Washington DC and INRIA, Paris.