Verified Order-based Secure Concurrency Controller in Multilevel Secure Database Management Systems

Yonglak Sohn[*], ***Nonmember*** **and Songchun Moon**[**], ***Member***

[*] Department of Computers Engineering
Seokyeong University
16-1 Jungneung-Dong Sungbuk-Ku,
Seoul 136-704, Korea
Tel.: +82-2-940-7103
Fax.: +82-2-919-0345
E-mail: syl@dbsun3.kaist.ac.kr

[**] Database Systems Laboratory
Graduate School of Management
Korea Advanced Institute of Science and Technology
207-43 Cheongryangri-Dong, Dongdaemun-Ku,
Seoul 130-012, Korea
Tel.: +82-2-958-3315
Fax.: +82-2-958-3604
E-mail: moon@dbsun3.kaist.ac.kr

SUMMARY

While the secure concurrency controllers (SCCs) in multilevel secure database systems (MLS/DBMSs) synchronize transactions cleared at different security levels, they must consider the problem of *covert channel*. We propose a new SCC, named *Verified Order-based secure concurrency controller* (VO) that founds on multiversion database.

VO maintains elaborated information about ordering relationships among transactions in a way of actively investigating and renewing the ordering relationships whenever it receives operations. With the elaborated information, it becomes capable of aborting transactions selectively whose non-interfered executions definitely violate one-copy serializability and providing more recent data versions to read requests than the other multiversion-based SCCs. Therefore, it comes to reduce the abort ratio and provide data versions with improved trustworthiness to transactions. By virtue of the elaborated information, moreover, VO is able to distinguish worthy versions and worthy transactions from unworthy ones, so that it is able to lighten the burdens of maintaining multiple versions and accumulated ordering relationships among transactions. For the aborts that are inevitable for preserving one-copy serializability, VO achieves security by deriving the conflicts to be occurred between transactions that have been cleared at the same security level.

Keywords: *Multilevel security, Database, Concurrency Control, Covert Channel*

1. Introduction

Multilevel security, MLS for short, is a capability that allows data objects with different security levels to be simultaneously stored and processed in an information system with users having different security levels. Preventing users from accessing data objects for which they are unauthorized is the absolute requirement in MLS. For building MLS information systems, MLS/DBMSs become a reality and in use today. Accordingly, SCCs for synchronizing concurrent transactions are receiving a lot of interest. They aim to preserve the correctness of database system with regard to the notion of serializability as well as security.

While transactions execute concurrently and share data objects, conflicts among them are often unavoidable. Conventional insecure concurrency controllers preserve correctness of database in a way of rejecting or delaying the conflicting operations [10]. Although these conventional ones preserve correctness, they are definitely vulnerable to an infringement of security. If such interference occurs from high transaction to low transaction, it opens an unexpected communication channel, named *covert channel*, between the transactions.[1] The problem of covert channel compulsorily demands the attention to security along with the serializability-related correctness. It therefore makes SCCs more complex than conventional concurrency controllers.

Much work, such as [3], [5]-[9], [11, 12], and [17]-[23], has been devoted to develop SCCs. While all the work achieve security successfully, some of them [3, 6, 7, 9, 17, 18, 20, 21, 23] argue that the traditional notion of serializability-related correctness is too restrictive for MLS/DBMS to achieve desirable performance. Thus, under certain circumstances, they willingly submit to the weakened correctness of execution sequence. Although the SCCs are capable of balancing security and performance to a certain extent, they cannot help infringing the correctness of database without a complement of user program.

The other SCCs proposed by [5, 8, 11, 12, 22] achieve correctness and security at the cost of declined performance of high transactions. For instance, whenever the SCC proposed by [5] receives an operation from a low transaction that conflicts with a high transaction, it preserves the correctness and security by aborting the high transaction unilaterally. Such a biased policy infringes the fairness in processing transactions cleared at different security levels.

SCCs proposed by [19, 23] are founded on multiversion database (MVDB). They try achieving security, correctness, and fairness in a way of disposing high transactions before active low transactions. However, they do not consider prudently the factual ordering relationships among transactions. Therefore, they are liable to dispose high transactions at excessively early positions. Accordingly, high transactions are forced to read far outdated data versions. These SCCs, moreover, have a burden of maintaining unbounded number of versions in a database and thus lead to the increased disk I/O requests.

In this paper, we propose a new MVDB-based SCC, named *Verified Order-based secure concurrency controller, VO* for short. It aims to achieve correctness and security at once without provoking serious unfairness in response time, with improving the recentness of data versions being read, and with reducing the number of versions in a database. The improved fairness would play advantageous role for VO to be evaluated as highly available one since it is capable of suppressing impure transactions that are to obstruct the legitimate executions of transactions by generating intentional and intensive conflicts. The improved recentness would contribute to providing more trustworthy data versions than the other MVDB-based SCCs. The reduced number of data versions, moreover, results in the decreased cost for storage and thereby reduces the number of disk I/O requests.

For these achievements, VO pays attention to the fact that non-interfered processing of conflicting operations does not always violate correctness or security. In addition, it takes notice of the fact that retaining sufficient trace of ordering relationships among transactions is very much useful for perceiving justice or injustice of non-interfered execution of conflicting operations. The strengthened capability for verifying ordering relationships among transactions is advantageous for providing sufficiently recent data versions, pointing out unworthy

[1] In this paper, the terms *high* and *low* refer two security levels such that the former is strictly higher than the latter.

transactions, and weeding out garbage versions from a database. VO achieves security in a way of driving all the incorrectness provoking conflicts to occur among transactions cleared at the same level.

The rest of this paper is organized as follows: Section 2 presents the models for transaction and security. Section 3 describes the features and the rules of VO. In Section 4, we offer the rules for weeding out unworthy versions and transactions. Section 5 provides correctness and secureness proofs for VO. Section 6 presents the results of VO's performance evaluation that have been compared with the other representative SCCs. Finally, this paper concludes in Section 7.

2. The Model

2.1 Transaction model

A transaction is an abstract unit of concurrent computation that executes with primitives: *begin*, *read*, *write*, *commit*, and *abort*. Users interact with the database by invoking transactions. History, H, is a set of totally ordered operations that have been synchronized by an SCC. It reflects every state of SCC, such as correctness, security, recentness, and fairness. If H preserves security all the time, for instance, security preservation of the SCC can be approved. If an operation, op_i, precedes the other operation, op_j, in H, their ordering relationship is represented by $op_i \quad op_j$.

A new SCC, VO, proposed in this paper founds on MVDB. In an MVDB, each write operation on a data item ultimately produces a new version of the data item that is to be added into a list of versions. Hence, the new version being created never overwrites its previous versions but instead coexists with them. Such coexisting data versions are called *sibling versions*. By virtue of the coexistence of sibling versions, almost all of the MVDB-based SCCs become capable of processing tardy reads unhesitatingly in a way of providing the second best versions. All the siblings of a data item, x, are totally ordered on purpose that a specific version, x_i whose creator is a transaction T_i, can be selected as an appropriate one for a read on x. A *version order operator*, «, represents the ordering relationship between siblings. For instance, x_i « x_j says that x_i precedes x_j in version order. For proving the correctness of MVDB-based history, the principle of *one-copy serializability*, *1SR* for short, is applied [10].

2.2 Security model

A security model provides a semantic representation in that it describes functional and structural security properties of a system. In describing the security model, two different types of entities, subjects and objects, are needed. Subjects represent the active entities, such as users, processes, and transactions. Objects stands for the passive entities, such as files, relations, and data objects at the level of tuples or attributes. Since we are particularly concerned with SCC, however, we substitute subjects and objects with transactions and data objects for clarity.

To describe the security model definitely, we present several assumptions and definitions that are inevitable for developing SCC.

Assumption 1(Untrustworthiness of transactions): All the transactions manipulated by SCC are regarded as untrusted ones. ❑

Exceptionally, however, transactions invoked by security administrators are regarded as trusted ones.

Assumption 2(Trustworthiness of SCC): SCC by itself never infringes security. It is now regarded as a trusted computing base (TCB). ❑

As SCC is TCB, it never contains any means, such as Trojan Horses or Trapdoors, that may lead to security violation. In accordance with Assumption 1 and Assumption 2, we are now allowed to confine the threats made against security to the only ones caused by untrusted transactions.

For describing the relationships between the transactions and the data objects at the standpoint of security, a means for labeling security level is necessary.

Definition 1(Security level labeling function): Security level labeling function, L, maps a transaction and a data object to their specific security levels. It is represented by

$$L: T \cup D \rightarrow S,$$

where D is a set of data objects, T is a set of transactions, and S is a hierarchical set of levels. ❑

For two entities, E_i and E_j, if $L(E_i) > L(E_j)$, E_i has been cleared at higher security level than E_j.

Assumption 3(Comparability of security levels): All the security levels are comparable each other. ❑

We assume that every pair of terms, high and low, used hereafter is comparable from the standpoint of compartments which implement the principle of least privilege [16]. In other words, we assume that the comparison of entities' compartments has already been completed successfully prior to the comparison of security levels.

In order to filter out unauthorized requests of transactions, an MLS information system needs to define an *access control policy*. In this paper, we adopt the access control policy as follows:

Definition 2(Access control policy): A transaction T_i is allowed to access a data d_j if the ordering relationship between $L(T_i)$ and $L(d_j)$ obeys following two constraints:
1(Read-equal/Read-down): T_i is allowed to read from d_j if and only if

$$L(T_i) \geq L(d_j).$$

2(Write-equal): T_i is allowed to write a new value to d_j if and only if

$$L(T_i) = L(d_j). ❑$$

The access control policy is strictly founded on the Restricted Bell-LaPadula model [1]. We do not allow write-up in that it is vulnerable to the attack of Trojan Horse.

The access control policy is able to cut off strictly any direct information flow from a transaction, T_i, to the other transaction, T_j, with $L(T_i) > L(T_j)$ via accessing the values of data objects. Unfortunately, however, it is insufficient to close a covert channel. The covert channel can be closed if and only if any execution of T_i never interferes that of T_j. If a concurrency controller is to achieve correctness and secureness at once, it must be able to produce *trustedly serializable*, *TSR* for short, histories all the time.

Definition 3(Trusted serializability): A history, H, is trustedly serializable if and only if it satisfies following two requirements simultaneously:
1(Serializability preservation): H is serializable, and
2(Security preservation): For any pair of transactions, T_i and T_j with $L(T_i) > L(T_j)$, in H, any interference from T_i to T_j must have been eradicated. ❑

Since VO founds on MVDB, the first requirement is specified to the notion of 1SR. By accomplishing the second requirement, VO successfully closes the covert channel.

3. Verified Order-based Secure Concurrency Controller: *VO*

We propose, in this section, a new SCC that is capable of producing TSR histories all the time. Achieving TSR is performed in ways of strictly preserving 1SR and forcing the transactions, which are responsible for the unavoidable interference, to be cleared at the same level.

3.1 Verified transaction ordering relationship

For the purpose of producing TSR history, VO maintains information that is capable of delineating obviously which transactions precede and which other transactions follow a specific transaction. VO verifies such *transaction ordering relationship*s, *TOR*s for short, whenever it receives an operation. TORs maintained by VO are now irrelevant to the transactions' beginning points of times but instead they faithfully reflect the actual relationships among transactions via information flows. Therefore, VO becomes capable of avoiding the hasty determination of TORs. When VO detects a TSR violating operation, it decides to reject the operation or to map it with a second-best data version. In this way, VO guards transactions against unnecessary interferences or reading far outdated data versions. VO retains the verified TORs until it can convince that deleting the TORs does not loss any information that is necessary for perceiving the occurrence of TSR violating operation.

Previous SCCs, unlike VO, do not investigate TORs intensively but they vaguely estimate coming TORs. Once a TOR between two transactions has been established, some of them are so pessimistic that they hastily conclude the appearance of new TORs in opposite direction. On the contrary, some others of them are so optimistic that they expect such oppositely directed TORs would never come. Rests of them impatiently establish TORs at the transactions' beginning points of times. These probability-based vague estimations of TORs lead to unnecessary interference or provision of excessively outdated data versions.

The unnecessary interference or the provision of excessively outdated data versions stems from the lack of elaborate information about the ordering relationships among transactions. For overcoming such a lack of information, VO accumulates the transaction ordering relationships that have been established in pursuance of following rules:

Rule 1(Transaction order verification):
1(Transaction order with regard to read-from): If $w_j[x_j]$ c_j $r_i[x_j]$ exists in a history, VO considers T_j to precede T_i.
2(Transaction order with regard to version order): If $x_i \ll x_j$, VO considers T_i to precede T_j.
3(Transaction order with regard to read-preceding): If $r_i[x_j]$ $w_k[x_k]$ or $w_k[x_k]$ $r_i[x_j]$ with $x_j \ll x_k$ in a history, VO considers T_i to follow T_j and to precede T_k. ❑

For determining ordering relationship between siblings, VO has following property:

Property 1(Commit order corresponding version order): If VO processes a commit of T_i before that of T_j and versions, x_i and x_j, have already been created, the version order between x_i and x_j becomes $x_i \ll x_j$. ❐

If there are $w_i[x_i]$ and $w_j[x_j]$ in a history and transactions, T_i and T_j, both are still active, in accordance with Property 1, VO postpones the decision on TOR between T_i and T_j until it receives

c_i or c_j. If VO receives c_i earlier than c_j, it decides on the version ordering relationship between x_i and x_j to be $x_i \ll x_j$. By applying Rule 1 to this case, VO considers T_i to precede T_j. VO opens a version only after its creator commits since it is confident of providing sufficiently recent data versions. By providing committed versions, in addition, it is capable of avoiding cascaded aborts.

In order to utilize the verified TORs for perceiving the occurrences of TSR violating operations, VO maintains following two sets of transactions for each individual transaction: *preceding set* and *following set*.

Definition 4(Set of preceding transactions and set of following transactions): For a transaction, T_i, there are two sets of transactions, $PT(T_i)$ and $FT(T_i)$, that are composed of transactions which precede and which others follow T_i, respectively. ❑

On receiving a begin operation from a transaction, T_i, VO initializes $PT(T_i)$ and $FT(T_i)$ to $\varnothing$. If VO decides to abort T_i, it removes $PT(T_i)$ and $FT(T_i)$. Whenever VO receives an operation, op_i, it decides a TOR between T_i and any other transaction, T_j, as follows: If VO verifies that T_i precedes T_j, it adds T_i to $PT(T_j)$ and at the same time adds T_j to $FT(T_i)$. On the contrary, if VO verifies that T_i follows T_j, it adds T_i to $FT(T_j)$ and T_j to $PT(T_i)$. Otherwise, VO keeps $PT(T_i)$, $FT(T_i)$, $PT(T_j)$ and $FT(T_j)$ intact.

Whenever VO is to add a transaction, T_i, to $PT(T_j)$ or $FT(T_j)$, it does not merely add T_i but also adds all the transactions that precede or follow T_i, respectively. For three transactions, T_i, T_j and T_k, let us assume that T_i precedes T_j and T_j precedes T_k. In this case, T_i precedes T_k and T_k follows T_i naturally. In order to reflect such extended TORs, whenever VO adds T_i to $PT(T_j)$, it sets $PT(T_j)$ to $PT(T_j) \cup \{T_i\} \cup PT(T_i)$. As $T_i \in PT(T_j)$, T_j becomes a transaction that follows T_i and thus all the transactions in $FT(T_j)$ also turn out to follow T_i. Accordingly, VO sets $FT(T_i)$ to $FT(T_i) \cup \{T_j\} \cup FT(T_j)$.

While positioning transactions, VO has following property.

Property 2(Non-intersection of preceding and following transactions sets): For a transaction, T_i, VO maintains $PT(T_i) \cap FT(T_i)$ to be $\varnothing$ all the time. ❐

With Property 2, VO precludes a transaction, T_i, which has already been positioned prior to a transaction, T_j, from being positioned posterior to T_j. By virtue of such a non-intersection in TOR between T_i and T_j, we can educe a theorem with regard to the notion of 1SR as follows:

Theorem 1(Preservation of one-copy serializability with non-intersection of preceding and following transactions sets): A history H is 1SR if and only if $PT(T_i) \cap FT(T_i)$ is $\varnothing$ for any transaction, T_i, in H.
Proof: For an arbitrary set of three transactions, T_i, T_j, and T_k in a history, H, let us suppose that $T_j \in PT(T_i)$ and $T_k \in FT(T_i)$. As long as $PT(T_i) \cap FT(T_i)$ is $\varnothing$, T_k never precedes T_j and at the same time T_j never follows T_k. Thus, the ordering relationships among them are as follows: T_j precedes T_i, T_i precedes T_k, and T_k never precedes T_j. Accordingly, a partial history that is composed of T_i, T_j, and T_k is now 1SR.

As T_j precedes T_i, all the transactions, that precede T_j, precede T_i as well. As T_k follows T_i, moreover, all the transactions that follow T_k follow T_i as well. Accordingly, $PT(T_i) \cup FT(T_i) \cup \{T_i\}$ composes an extended partial history that covers all the transactions in H to which T_i is relevant. As $PT(T_i) \cap FT(T_i)$ is $\varnothing$, the extended partial history is now allowed to be 1SR. For any other arbitrary transaction, T_l, if $PT(T_l) \cap FT(T_l)$ is $\varnothing$, $PT(T_l) \cup FT(T_l) \cup \{T_l\}$ is 1SR as well. As long as Property 2 is preserved for every transaction in H, therefore, every corresponding partial history in H is 1SR. Consequently, H is 1SR. ❐

3.2 Visualization of transaction ordering relationship

For the purpose of visualizing TORs in a history, H, we use *transaction ordering relationship edge, path,* and *graph* defined as follows:

Definition 5(Transaction ordering relationship edge): For two transactions, T_i and T_j, if $T_i \in PT(T_j)$ and at the same time $FT(T_i) \cap PT(T_j)$ is $\emptyset$, TOR between T_i and T_j is depicted in $T_i \rightarrow T_j$ which is named *TOR edge*. ❐

Definition 6(Transaction ordering relationship path): *TOR path* is composed of one or more TOR edges and two or more transactions as nodes. For two transactions, T_i and T_j, TOR path from T_i to T_j is depicted in $T_i \Rightarrow T_j$. Within $T_i \Rightarrow T_j$, there are no ramified TOR edges. ❐

Definition 7(Transaction ordering relationship graph): For a history, H, a *TOR graph*, denoted by TORG(H), is a directed graph that is composed of TOR edges, TOR paths, and all the transactions in H as nodes. ❐

For an arbitrary transaction, T_i, that precedes T_j in a history, H, TORG(H) includes $T_i \Rightarrow T_j$. If there are two TOR paths in TORG(H), such as $T_i \Rightarrow T_j \rightarrow T_k$ and $T_i \Rightarrow T_j \rightarrow T_l$, two TOR edges, $T_j \rightarrow T_k$ and $T_j \rightarrow T_l$, have been ramified from $T_i \Rightarrow T_j$. In this case, T_k and T_l both are relevant to all the transactions in $T_i \Rightarrow T_j$. However, they are irrelevant to each other as long as TORG(H) does not have any TOR paths between T_k and T_l.

Definition 8(Dead-ended transaction ordering relationship path): If a transaction, T_i, appears in a TOR path that flows into T_j also appears in the other TOR path that flows out from T_j, the TOR path between T_i and T_j is said to be *dead-ended*. Such a dead-ended TOR path between T_i and T_j is depicted in $T_i \Leftrightarrow T_j$. ❐

As long as TORG(H) does not include any dead-ended TOR paths, according to Theorem 1, the history, H, assures the preservation of 1SR.

In TORG(H), all the TOR edges could be categorized into two categories, inflowing TOR edges and outflowing TOR edges, defined as follows:

Definition 9(Inflowing and outflowing transaction ordering relationship edges): For some sequences of operations in a history, such as $w_j[x_j]$ $r_i[x_j]$, $r_j[x_p]$ $w_i[x_i]$, or $w_j[x_j]$ c_j $w_i[x_i]$, TOR edge *flows into* T_i: $T_j \rightarrow T_i$. For the other sequences of operations, $w_j[x_j]$ $r_i[x_p]$ with x_p « x_j or $w_j[x_j]$ $w_i[x_i]$ c_i, on the contrary, the ordering edge *flows out* from T_i: $T_j \leftarrow T_i$. The operations written in italic are the ones that arrive at the history just now. ❑

The directions of inflowing and outflowing TOR edges have been categorized from the standpoint of a transaction whose operation is currently arriving at a history. From the directional categorizations, we offer following lemma.

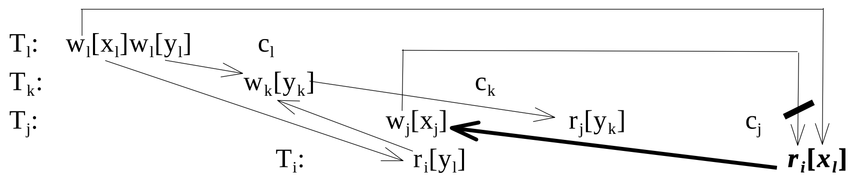
Lemma 1(Absence of write-originated outflowing transaction ordering relationship edge): Whenever a write operation is scheduled, it is destined to form only inflowing TOR edges. It never generates outflowing ones.

Proof: Let us assume that VO is to schedule a write of a transaction, T_i. While T_i is active, the version created by T_i is invisible. Therefore, a read of another active transaction, T_j, cannot induce a TOR edge that flows out from the write of T_i. If the other transaction, T_k, creates a version that is sibling with the version created by T_i, VO cannot generate TOR edge between T_i and T_k until it receives c_i or c_k due to Property 1. If VO receives c_k before c_i, according to Rule 1, it considers T_k to precede T_i. Otherwise, T_i no longer is an active one. VO, consequently, never generates a TOR edge that flows out from a write of an active transaction. ❑

In accordance with Lemma 1, VO comes to convince that every TOR edge that flows out from an active transaction must have been originated from a read.

3.3 Preclusion of trusted serializability violation

For serving a read on multiversion database, providing the latest data version is most desirable. However, simple-minded adherence to this manner could lead a history to violate TSR. Therefore, mapping the read to a second-best version becomes often inevitable. Fig. 1 and Fig. 2 illustrate such inevitabilities in situations of scheduling read and write, respectively.

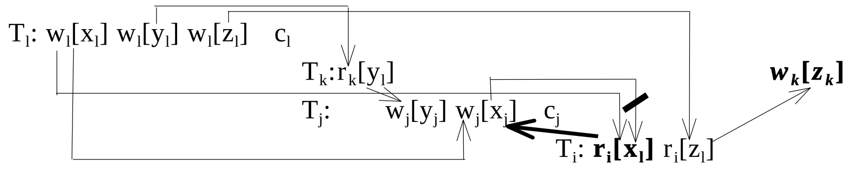


T_l: $w_l[x_l]w_l[y_l]$ c_l
T_k: $w_k[y_k]$ c_k
T_j: $w_j[x_j]$ $r_j[y_k]$ c_j
T_i: $r_i[y_l]$ $r_i[x_l]$

Fig. 1 Preclusion of read-triggered dead end

In Fig. 1, mapping $r_i[x]$ to $r_i[x_j]$ provokes a dead end, $T_i \rightarrow T_k \rightarrow T_j \rightarrow T_i$. By mapping $r_i[x]$ to a second-best version, x_l, instead of x_j where $x_l \ll x_j$, VO comes to dispose T_i prior to T_j. It now, therefore, substitutes $T_j \rightarrow T_i$ in the dead end by $T_i \rightarrow T_j$ and thus precludes the dead end among T_i, T_j, and T_k.

For Fig. 2, although it includes $r_i[x_l]$, let us assume tentatively that $r_i[x]$ has already been mapped to x_j. When $w_k[z_k]$ appears, in this case, the history including $r_i[x_j]$ cannot preserve 1SR without aborting T_k. If $L(T_i) > L(T_k)$, however, such an unavoidable abort of T_k opens a covert channel between T_k and T_i. When $w_k[z]$ arrives, in the end, it is too late to preserve 1SR and security at a time. Before $w_k[z]$ arrives, therefore, we cannot help eradicating any possibilities that are suspected of provoking TSR violation. As depicted in Fig. 2, VO is capable of preserving TSR successfully by mapping $r_i[x]$ to x_l. Through such a mapping, VO replaces $T_j \rightarrow T_i$ with $T_i \rightarrow T_j$ and thereby eradicates the dead end without opening covert channel.



T_l: $w_l[x_l]$ $w_l[y_l]$ $w_l[z_l]$ c_l
T_k:$r_k[y_l]$
$w_k[z_k]$
T_j: $w_j[y_j]$ $w_j[x_j]$ c_j
T_i: $r_i[x_l]$ $r_i[z_l]$

Fig. 2 Preclusion of write-triggered dead end

Although VO preserves TSR at the cost of mitigating the requests of reading the latest versions, it must be capable of serving a transaction to read a version that is the most recent one among the candidates for being selected. When VO is to select such an appropriate version, it observes following rule.

Rule 2(Trusted version selection):

1(One-copy serializability preserving version selection): When VO receives a read, it searches for an appropriate data version until it can convince that providing the selected version never provokes a dead end.

2(Security preserving version selection): Suppose that VO is to select x_j as the appropriate version for $r_i[x]$ in accordance with the rule of *1SR preserving version selection*. At this time, if there is an active transaction, T_k, in $PT(T_j)$ with $L(T_i) > L(T_k)$, VO relinquishes selecting x_j. Instead, it selects the other version, x_l with $x_l \ll x_j$ and $T_k \notin PT(T_l)$. In this case, x_l is naturally the most recently committed version among its siblings that precede x_j. In addition, T_i's reading x_l must not violate the rule of 1SR preserving version selection. ❐

In case of the rule of *security preserving version selection*, VO relinquishes T_i's reading x_j since it becomes apprehensive of opening a covert channel between T_i and T_k. If x_j is selected for the appropriate version to $r_i[x]$, $T_k \Rightarrow T_i$ appears to be established since $T_k \in PT(T_j)$. If a new TOR edge, $T_i \rightarrow T_k$, is established due to $r_i[z_l]$ $w_k[z_k]$ or $w_k[z_k]$ $r_i[z_l]$ with $z_l \ll z_k$, the TOR path between T_i and T_k turns out to be $T_i \Leftrightarrow T_k$. Since VO cannot assure that it will never receive such a sequence of dead end provoking operations, it is obliged to cope with such a situation prudently in a way of providing a second-best data version to read in advance. By selecting x_l instead of x_j for the appropriate version to $r_i[x]$, VO becomes capable of avoiding $T_k \rightarrow T_i$. After $r_i[x_l]$, although any writes of T_k may establish $T_i \rightarrow T_k$, they never induce $T_i \Leftrightarrow T_k$. As a result, such a write is allowed to be processed without being interfered by T_i. Accordingly, VO is now able to close any possible covert channels between T_i and T_k. With the rules of *trusted version selection*, consequently, VO is assured to produce TSR preserving histories for these situations.

For the histories depicted in Fig. 1 and Fig. 2, VO has preserved TSR without provoking any interference. Unfortunately, however, there are many other situations in which such non-interfered executions are destined to produce 1SR violating histories. In these situations, VO does not hesitate to abort a transaction that currently issues a dead end provoking operation. Notwithstanding such an abort, VO preserves TSR due to its capability of deriving the transactions in the dead-ended TOR path to be cleared at the same security level. This will be proved at Section 5.

4. Secure Elimination of Unworthiness

Although we can expect that VO will improve the degree of concurrency due to the elaborated information about TORs, it has intrinsic burdens, such as maintenance of multiple versions and accumulated TORs. In order to lighten such burdens, we offer functions that are capable of eradicating unworthy versions and unworthy TORs.

4.1 Secure elimination of unworthy versions

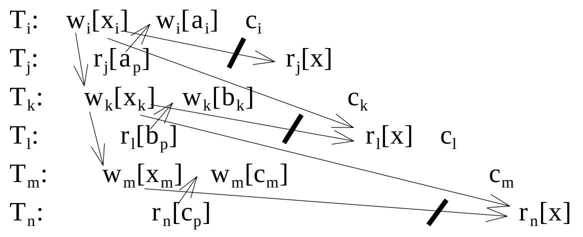
For eliminating unworthy versions, we focus on the fact that versions turn out to be unworthy ones if they would never be read by transactions any more. Otherwise, the versions are sufficiently worthy to be retained. By weeding out such unworthy versions, the size of database appears to be reduced. A function of this nature in VO is called *secure garbage collection* and invoked when VO receives a commit.

4.1.1 Selective elimination of unworthy versions

According to the access control policy, if $L(T_i) \geq L(x)$, a transaction, T_i, is authorized to read from a data item, x. However, not all the siblings of x are *readable* to T_i. For a specific version, x_j, it is readable to T_i if and only if $T_j \notin STT(T_i)$ and VO can convince that processing $r_i[x_j]$ will never induce an unavoidable violation of security. Otherwise, it is regarded as *unreadable* one to T_i.

Let us assume that there are two adjacent siblings, x_j and x_k, with $x_j \ll x_k$. If they both are verified to be readable to a unique active transaction, T_i, VO maps $r_i[x]$ to $r_i[x_k]$ instead of $r_i[x_j]$. Although x_j is readable to T_i in this case, it has been shaded by x_k and thus cannot be read by T_i any more. For any coming transaction, T_l with $L(T_l) \geq L(x)$, moreover, Rule 2 forces T_l to read x_k. VO is now able to convince that deleting x_j will never lead to an information loss. Consequently, it regards x_j as unworthy one. Example 1 shows this.

Example 1(Determination of worth and unworthy versions): Suppose that there are six concurrent transactions, T_i, T_j, T_k, T_l, T_m, and T_n with $L(T_i) = L(T_k) = L(T_m)$, and $L(T_j) \geq L(T_i)$, $L(T_l) \geq L(T_i)$, $L(T_n) \geq L(T_i)$ in Fig. 3. Owing to $w_i[a_i]$, $r_j[a_p]$, and $a_p \ll a_i$, VO adds T_j into $PT(T_i)$ and thus prohibits T_j from reading x_i. When T_i commits, notwithstanding that x_p has been shaded by x_i from the viewpoints of T_k, T_l, T_m, and T_n, it has not been shaded by x_i yet from the viewpoint of T_j since VO cannot assure that it will never receive $r_j[x]$ after that time. x_p therefore is still worthy to T_j. In the same way, x_i is still worthy to T_l when T_k commits. For $r_n[x]$, although $x_k \ll x_m$, x_k is still worthy to T_n.


T_i: $w_i[x_i]$ $w_i[a_i]$ c_i
T_j: $r_j[a_p]$ $r_j[x]$
T_k: $w_k[x_k]$ $w_k[b_k]$ c_k
T_l: $r_l[b_p]$ $r_l[x]$ c_l
T_m: $w_m[x_m]$ $w_m[c_m]$ c_m
T_n: $r_n[c_p]$ $r_n[x]$
Fig. 3 Worthy and unworthy versions

On the other hand, when T_m commits, note that only T_j and T_n are active. At that time, x_i and x_k both are readable to T_n and unreadable to T_j. Therefore, x_i will never be read by both T_j and T_n, so that VO is now allowed to regard x_i as unworthy one. Although x_p precedes the unworthy version, x_i, in this case, it is still worthy to T_j.
End of Example 1 ■

Rule 3 provides formal determination rules for weeding out unworthy versions.

Rule 3(Determination of selective unworthy version): For two adjacent siblings, x_i and x_j with $x_i \ll x_j$, and for two sets of active transactions, ST_a and ST_b, VO determines that x_i is unworthy version and x_j is worthy version if $ST_a \cup ST_b$ covers all the active transactions and one of following requirements is satisfied:
1(Readable shading): x_i and x_j are readable to all transactions in both ST_a and ST_b, or
2(Unreadable shading): x_i and x_j are unreadable to all transactions in both ST_a and ST_b, or
3(Complex shading): x_i and x_j are readable to all transactions in ST_a and they are unreadable to all transactions in ST_b or vice versa. ❑

4.1.3 Multiple elimination of unworthy versions

When a new version, x_i, is created, all the previous siblings of x_i are sometimes turned out to be unworthy ones. In this case, fortunately, maintaining only x_i appears to be sufficient for x. VO achieves such a drastic elimination of versions by applying following rule:

Rule 4 (Determination of multiple unworthy versions): When VO receives a commit from a transaction, T_i, if it meets one of following three cases, it is allowed to determine that x_i is a unique worthy version.

Case 1: There are no active transactions.

Case 2: For every active transaction, T_j, $L(T_i) > L(T_j)$.

Case 3: All the transactions in $PT(T_i)$ have already been committed. ❑

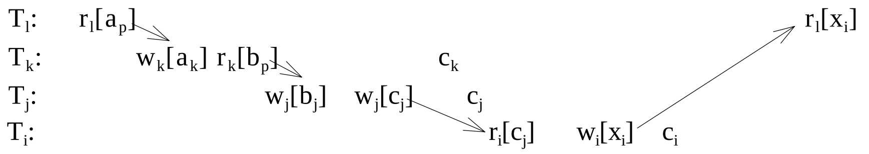
For Case 1, all the coming transactions will never read any versions that precede x_i. For Case 2, T_j has been prohibited from reading a version of x on account of $L(T_i) = L(x) > L(T_j)$. Case 3 implies that any transactions in $PT(T_i)$ will never issue any operations after T_i commits. For the coming transactions, like Case 1, all the preceding siblings are shaded by x_i. By applying Rule 4, VO is now capable of eradicating the preceding siblings of x_i entirely. If there is any active transaction, T_k with $L(T_k) > L(T_i)$, in $PT(T_i)$, however, such a drastic eradication may lead to an absence of version that must be read by T_k.

Unlike $PT(T_i)$, any transactions in $FT(T_i)$ do not distort the results of applying Rule 4. For the committed transactions in $FT(T_i)$, they will never issue any operations after T_i commits. If there is an active transaction, T_l with $L(T_i) > L(T_l)$, in $FT(T_i)$, T_l is not authorized to read from x since $L(T_i) = L(x) > L(T_l)$. If $L(T_l) \geq L(T_i)$, on the contrary, all the preceding siblings are readably shaded by x_i from the viewpoint of T_l. In applying Rule 4, VO is now independent of all the transactions that follow the committing transaction, T_i.

4.2 Secure elimination of unworthy transactions

While maintaining TORs, the accumulated TOR edges become gradually heavy burden for verifying the preservation of TSR. VO lightens such a burden by weeding out unworthy TOR edges. When VO is capable of assuring that a committed transaction will never be included in a dead-ended TOR path, it is allowed to regard all the TOR edges that flow into or out from the transaction as unworthy ones. Example 2 shows how VO discriminates between worthy transactions and unworthy transactions.

Example 2(Perception of worthy and unworthy transactions): Suppose there are four concurrent transactions, T_i, T_j, T_k, and T_l in Fig. 4. When T_i commits, although c_k c_j c_i, T_k and T_j cannot be regarded as unworthy ones. If one or both of them are regarded as unworthy ones, $T_l \rightarrow T_k \rightarrow T_j$, $T_k \rightarrow T_j \rightarrow T_i$, or both may be deleted. Without the information about TOR between T_l and T_i, $r_l[x]$ would be mapped to $r_l[x_i]$. As shown in Fig. 4, such a mapping is destined to generate a dead end, $T_l \rightarrow T_k \rightarrow T_j \rightarrow T_i \rightarrow T_l$. Owing to the existence of active preceding transaction, T_l, therefore, the worthiness of T_k and T_j must be retained.


T_l: $r_l[a_p]$ $r_l[x_i]$
T_k: $w_k[a_k]$ $r_k[b_p]$ c_k
T_j: $w_j[b_j]$ $w_j[c_j]$ c_j
T_i: $r_i[c_j]$ $w_i[x_i]$ c_i
Fig. 4 Perception of worthiness

If T_l has already been committed when T_i commits, on the contrary, $PT(T_k)$ and $PT(T_j)$ come to be composed of only committed transactions. In this case, any transaction in $PT(T_k)$ or $PT(T_j)$ no longer can generate a dead end, $T_i \Leftrightarrow T_k$ or $T_i \Leftrightarrow T_j$. Owing to the absence of active preceding transaction in $PT(T_k)$ and $PT(T_j)$, in the end, VO is allowed to regard T_k and T_j as unworthy ones.

End of Example 2 ■

Example 2 focuses on a transaction that precedes committed transactions. Let us now consider the other case that an active transaction exists as a following one. For instance, when a transaction, T_i, commits, suppose that a committed transaction, T_m, and an active one, T_n, are included in a history, H, and TORG(H) has $T_i \Rightarrow T_m \Rightarrow T_n$. In addition, let us assume that $L(T_n) > L(T_m)$ and T_n requests a read on *x* whose sibling has already been generated by T_m. As T_n follows T_m and there are no active low transactions in $PT(T_n)$, $r_n[x]$ will be mapped to $r_n[x_m]$ or $r_n[x_i]$. Such a mapping keeps $T_m \Rightarrow T_n$ intact. For a write of T_n, owing to $L(T_n) > L(T_m)$, it does not establish any new TOR edge between T_m and T_n. If $L(T_m) = L(T_n)$ and H includes $w_m[x_m]$ c_m $w_n[x_n]$, owing to Property 1 and Rule 1, T_n still follows T_m. As T_m has been committed while T_n is active, H never includes $w_n[x_n]$ c_n $w_m[x_m]$. Any write of T_m, therefore, cannot add $T_m \leftarrow T_n$ to TORG(H). If $L(T_m) > L(T_n)$ and T_n interleaves with T_m, Rule 2 prohibits T_n from preceding T_m. Accordingly, any read or write of T_n will never generate $T_m \leftarrow T_n$. If $L(T_m) > L(T_n)$ and T_n does not interleave with T_m, any read of T_n does not add new TOR edge between T_m and T_n to TORG(H). For any write of T_n, it keeps $T_m \Rightarrow T_n$ intact as well. From all the possible cases considered, we come to conclude that an active transaction, T_n, that follows a committed transaction, T_m, does not affect the TOR between T_m and T_n. Consequently, deleting all the T_m related TOR edges does not make VO to be insensible of TSR violation.

VO now comes to have rules for determining unworthy transaction:

Rule 5(Determination of unworthy transaction):
1(Absence of active transactions): When VO receives a commit, if there are no active transactions, it decides that all the committed transactions are unworthy ones.
2(Absence of active preceding transactions): When VO receives a commit from a transaction, T_i, if there are no active transactions in $PT(T_j)$ of a committed transaction, T_j, that precedes T_i, VO decides on T_j to be unworthy one. ❑

By applying Rule 5, VO becomes capable of reducing the overhead for maintaining TORs drastically.

5. Proof of Trusted Serializability

In this section, we prove VO's preservation of TSR by showing that all the transactions in a dead end have been cleared at the same level. Before getting down to proof, we will begin with Lemma 2.

Lemma 2(Absence of high transaction penetrating read-originated TOR path): No outflowing TOR paths that have originated from read operations of an active transaction, T_i, pass through the other transaction whose security level is higher than $L(T_i)$.

Proof: Let us assume that there is a pair of an active transaction, T_i, and the other active(or committed) transaction, T_k, with $L(T_k) > L(T_i)$ in a history, H. For any TOR path that flows out from T_i, according to Lemma 1, it must have originated from T_i's read. In case that the TOR path does not include any transactions that intermediate between T_i and T_k, it cannot connect T_i and T_k since it is destined to flow into a write. We therefore have to consider the TOR paths that indirectly connect T_i to T_k via a mediating transaction, T_j, i.e., TORG(H) includes $T_i \Rightarrow T_j \rightarrow T_k$. For $T_i \Rightarrow T_j$, its connection can be established in two different cases: (Case 1) without mediation or (Case 2) with mediation.

Case 1: The absence of mediation between T_i and T_j means that a TOR path that flows out from a read of T_i directly flows into T_j's write. That is, H includes $r_i[x_p]$ and $w_j[x_j]$ with $x_p \ll x_j$ and thus $PT(T_j)$ includes T_i. Such $r_i[x]$ and $w_j[x]$ imply $L(T_i) \geq L(T_j)$ and thereby $L(T_k) > L(T_j)$. If VO is to generate $T_j \rightarrow T_k$, it must provide a version, y_j, to T_k due to $L(T_k) > L(T_j)$. At this time, however, $PT(T_j)$ has already included T_i. Owing to $L(T_k) > L(T_i)$ and Rule 2, VO cannot help relinquishing the provision of y_j to T_k but instead is obliged to map $r_k[y]$ to $r_k[y_p]$ with $y_p \ll y_j$. Therefore, it comes to add $T_k \rightarrow T_j$ to TORG(H). In this case, consequently, a TOR path that has flown out from T_i's read never flows into a transaction, T_k with $L(T_i) > L(T_k)$.

Case 2: Let us assume that a transaction, T_l, mediates T_i and T_j. As T_i is still active, according to Lemma 1, $T_i \Rightarrow T_j$ originates from T_i's read. If $T_i \rightarrow T_l$ is to be generated, as described in Case 1, H has to include operations, $r_i[x_p]$ and $w_l[x_l]$ with $x_p \ll x_l$. Such operations imply $L(T_i) \geq L(T_l)$. Like the case of T_i and T_l, VO adds $T_l \rightarrow T_j$ into TORG(H) if $L(T_l) \geq L(T_j)$. Owing to $L(T_k) > L(T_i)$, $L(T_k) > L(T_j)$ as well. If VO is to generate $T_j \rightarrow T_k$, since $L(T_k) > L(T_j)$, it has to provide a version, y_j, to T_k. At this time, T_l and T_i have been included in $PT(T_j)$. In accordance with Rule 2, therefore, VO comes to map $r_k[y]$ to $r_k[y_p]$ with $y_p \ll y_j$ instead of $r_k[y_j]$. Accordingly, it generates $T_k \rightarrow T_j$ instead of $T_j \rightarrow T_k$. In this case, a TOR path that originates from T_i's read never passes through T_k with $L(T_k) > L(T_i)$.

According to Case 1 and Case 2, consequently, any TOR path that flows out from a read of an active transaction never flows into high transactions but flows into low or equal security level transactions. ❐

In accordance with Definition 9, all the possible TOR edges can be categorized into 5 categories from the viewpoint of an arbitrary transaction, T_i, such as (1) $w_j[x_j] \rightarrow r_i[x_j]$, (2) $r_j[x_p] \rightarrow w_i[x_i]$ with $x_p \ll x_i$, (3) $w_j[x_j] \rightarrow w_i[x_i]$ with c_j c_i, (4) $w_j[x_j] \leftarrow r_i[x_p]$ with $x_p \ll x_j$, and (5) $w_j[x_j] \leftarrow w_i[x_i]$ with c_i c_j. We prove TSR preservation of VO by showing that every T_i relevant TOR edge produces TSR preserving history.

Lemma 3(TSR preservation of write-originated/read-pointing inflowing TOR edge (1)): Every inflowing TOR edge that flows out from a write and flows into a read of an active transaction preserves TSR.

Proof: For a history, H, let us assume that there has already been a TOR path, $T_i \Rightarrow T_j$, in TORG(H). When VO receives $r_i[x]$, according to Rule 2, it never maps $r_i[x]$ to $r_i[x_j]$ but instead maps to $r_i[x_p]$ with $x_p \ll x_j$. Such a mapping merely adds $T_p \rightarrow T_i$ and $T_i \rightarrow T_j$ into TORG(H) and thus never provokes $T_i \Leftrightarrow T_j$. VO now can process $r_i[x_p]$ immediately without an apprehension for TSR violation. In case of (1), VO is capable of preserving TSR of H. ❐

Lemma 4(TSR preservation of read-originated/write-pointing inflowing TOR edge (2)): Every inflowing TOR edge that flows out from a read and flows into a write of an active transaction preserves TSR.

Proof: Let us assume that there is a TOR path, $T_i \Rightarrow T_j$, in TORG(H) of a history, H. According to Lemma 2, any transactions included in $T_i \Rightarrow T_j$ have been cleared at levels that are lower than or equal to $L(T_i)$ while T_i is active. The TOR edge, $T_j \rightarrow T_i$, being considered has been generated by $r_j[x_p]$ and $w_i[x_i]$ with $x_p \ll x_i$. Such read and write of T_j and T_i imply $L(T_j) \geq L(T_i)$. Owing to $L(T_i) \geq L(T_j)$ and $L(T_j) \geq L(T_i)$, $L(T_i)$ and $L(T_j)$ now must be the same. By aborting T_i, accordingly, VO comes to preserve 1SR and such an abort does not infringe security of H. In case of (2), consequently, VO is assured to produce TSR preserving histories. ❐

Lemma 5(TSR preservation of write-originated/write-pointing inflowing TOR edge (3)): Every inflowing TOR edge that flows out from a write and flows into the other write of an active transaction preserves TSR.

Proof: For a history, H, suppose that TORG(H) has already included a TOR path, $T_i \Rightarrow T_j$. As proved in Lemma 2, a TOR path that has flown out from an active transaction, T_i, is destined to reach a transaction, T_j with $L(T_i) \geq L(T_j)$. Note that the TOR edge that is to provoke a dead end between T_i and T_j has been generated due to the creation of sibling versions. Thus, $L(T_i) = L(T_j)$. The abort of T_i, therefore, is capable of preserving 1SR without loss of security. Consequently, the write-originated/write-pointing TOR edge does not violate TSR of H. ❐

Lemma 6(TSR preservation of write-pointed/read-originating outflowing TOR edge (4)): Every outflowing TOR edge that flows into a write and flows out from a read of an active transaction preserves TSR.

Proof: For a history, H, suppose that there has already been an ordering path, $T_j \Rightarrow T_i$, in TORG(H). Let us also suppose that $T_i \rightarrow T_j$ is to be generated due to $w_j[x_j]$ $r_i[x_p]$ with $x_p \ll x_j$ in H. In case that $T_j \Rightarrow T_i$ has been originated by T_j's *read*, according to Lemma 2, $L(T_j) \geq L(T_i)$. Owing to $w_j[x_j]$ and $r_i[x_p]$, however, $L(T_i) \geq L(T_j)$. Therefore, T_i and T_j in this case have surely been cleared at the same security level.

For the other case, suppose that $T_j \Rightarrow T_i$ has been originated by T_j's *write*. According to Lemma 1, in this case, T_j must have been committed. If $T_j \Rightarrow T_i$ includes a TOR edge, $T_j \rightarrow T_k$, that flows into T_k's *read*, $r_k[y_j]$, $L(T_k)$ is higher than or equal to $L(T_j)$. However, $w_j[x_j]$ c_j $r_i[x_p]$ with $x_p \ll x_j$ implies that there is an active transaction, T_l with $L(T_i) > L(T_l)$, in $PT(T_j)$. When VO attempts to process $r_k[y_j]$, owing to the existence of T_l with $L(T_k) > L(T_l)$, it is obliged to map $r_k[y]$ to $r_k[y_p]$ with $y_p \ll y_j$. While $L(T_i) > L(T_j)$, therefore, a TOR path that has been extended from $T_j \rightarrow T_k$ never arrives at T_i. However, $T_j \Rightarrow T_i$ has already existed in TORG(H). $L(T_i)$, therefore, cannot be higher than $L(T_j)$. Moreover, $L(T_j)$ cannot be higher than $L(T_i)$ as well since VO received the legitimate operations, $w_j[x]$ and $r_i[x]$. $L(T_i)$ and $L(T_j)$ in the end are the same.

On the other hand, if $T_j \Rightarrow T_i$ includes a TOR edge, $T_j \rightarrow T_k$, that flows out from T_j's *write* and flows into T_k's *write*, $L(T_j)$ and $L(T_k)$ are the same. If VO attempts to add a TOR edge, which flows into a read, to $T_k \Rightarrow T_i$, it disposes the read requesting transaction before any write requesting transactions in $T_k \Rightarrow T_i$. While $L(T_i) > L(T_j)$, in this case, a TOR path that has been extended from $T_j \rightarrow T_k$ never arrives at T_i. Accordingly, $w_j[x_j]$ $r_i[x_p]$ with $x_p \ll x_j$ is to provoke a dead end if and only if $L(T_i)$ and $L(T_j)$ are the same. Consequently, in case of (4), VO produces TSR preserving history by aborting $r_i[x]$. ❐

Lemma 7(TSR preservation of write-pointed/write-originating outflowing TOR edge (5)): Every outflowing TOR edge that flows into a write and flows out from the other write of an active transaction preserves TSR.

Proof: Let us assume that $T_j \Rightarrow T_i$ has already been included in TORG(H) of a history, H. According to Rule 1, VO generates write-pointed/write-originating TOR edge, $T_j \leftarrow T_i$, in case of $x_i \ll x_j$. As VO establishes the version ordering relationship when one of the creators commits, T_i is committing just now. T_j, therefore, is still an active transaction. As long as T_j is active, $T_j \Rightarrow T_i$ implies $L(T_j) \geq L(T_i)$ in accordance with Lemma 2. On the other hand, sibling versions, x_i and x_j, imply $L(T_i) = L(T_j)$. Now that $L(T_j) \geq L(T_i)$ and $L(T_i) = L(T_j)$, the security levels of T_i and T_j in the end are the same. Consequently, TSR of H in this case is successfully preserved in a way of aborting T_i. ❐

With the Lemmas proved above, we come to present Theorem 2.

Theorem 2(Preservation of trusted serializability): All the possible histories produced by VO preserve TSR.

Proof: Lemma 3, 4, 5, 6, and 7 have completely considered all the possible TOR edges generated by VO and proved that VO preserves TSR whenever it is to schedule newly arrived operations. Consequently, all the histories produced by VO are definitely TSR. ❒

6. Performance Evaluation

We evaluate the performance of three SCCs by means of simulation approach. We have excluded the SCCs that weaken the correctness from comparison since such SCCs are liable to infringe integrity that is one of the major aspect of computer security. Among the SCCs that are capable of preserving the correctness strictly, we choose Orange-Locking method [5] and Order-Stamp method [19, 23] (hereafter OL and OS for short) to be compared with VO in that they are representative SCCs in the environments of single version and multiversion database, respectively. The simulation is implemented with the CSIM [4] discrete event simulation language and much of its model has been borrowed from [15].

6.1 Assumptions

In our simulation model, we have adopted following assumptions in order to make the performance evaluation process achievable.

Assumption 4(Closed queuing model): Our simulation employs a closed queuing model for single-site database system. ❑

The simulation model is closed in the sense that the system keeps the number of active transactions at the same all the time.

Assumption 5(Uniform workload): Every transaction accesses data items that are randomly chosen from database. ❑

With Assumption 5, the probability of conflict for each data item becomes uniform as well.

Assumption 6(Version chaining): All the sibling versions are maintained in a way of linked list in database. ❑

An appropriate version is searched from the most recent one to the less recent one. Therefore, the number of disk I/Os appears to increase according as the recentness of an appropriate version becomes degraded.

Assumption 7(Disk resident lock table): The lock table of a database is located in a disk. ❑

By locating the lock table in disk, the access patterns of OL, VO, and OS become fair since Assumption 6 has been adopted. If the lock table locates in main memory, the time for accessing data object turns out to be unilaterally favorable to OL.

6.2 Parameter settings

The input parameters with their brief description and setting values are listed in Table 1.

Table 1 Simulation parameter settings

Parameter	Parameter description	Parameter setting
num_cpus	Number of CPUs	2 CPUs
num_disks	Number of disks	4 disks
cpu_delay	CPU time for accessing an object	12 milliseconds
io_delay	I/O time for accessing an object	35 milliseconds
cc_delay	CPU time for scheduling an operation	3 milliseconds
num_data_item	Number of data items in database	1000 data items
tr_size	Size of transaction	8 ~ 12 operations
mpl	Multiprogramming level	10, 20 ~ 200 transactions in steps of 20
ext_think_time	External think time	5 seconds
fake_restart_pct	Percent of fake restarted transactions	20%
write_pct	Percent of write operations	10%, 20%, 30%
num_sec_level	Number of security levels	1, 2, 4, 8, 12

The computing resource environment is limited with 2 CPUs and 4 disks. Reading an object takes *io_delay* followed by *cpu_delay*. Writing an object, on the other hand, takes CPU resource before I/O resource. For each operation, every SCC spends *cc_delay*. The settings on *num_data_item* and *tr_size* would allow the interesting performance effects to be observed without requiring impossibly long simulation time [15]. *mpl* says the number of transactions that are concurrently synchronized. Whenever a transaction terminates at a specific terminal, a new transaction begins at that terminal after *ext_think_time*. However, *ext_think_time* does not distort the number of concurrent transactions at any time. For restarting an aborted transaction, there are basically two kinds of policy: real restart and fake restart. The policy of real restart assures the restarted transaction of accessing the set of data items that might have been accessed by the aborted transaction. Fake restart means that the restarted transaction is regenerated as a new independent transaction. The policy of real restart generally dominates that of fake restart in actual circumstances. We therefore set *fake_restart_pct* to 20%. *write_pct* is set to 10%, 20%, and 30% for observing the sensitivities founded on it. *num_sec_level* is ranged from 1 to 12. It covers from insecure DBMSs to multilevel secure DBMSs that are capable of handling sufficiently classified entities.

6.3 Performance indices

Average response time and throughput are the major performance indices in our simulation experiments. According to the assumption of closed queuing model and Little's law [2, 14], however, throughput can be inferred from response time. As long as response time is presented, hence, we omit the presentation of throughput due to the limitation on space.

In addition to the major indices, the other ones, such as abort ratio, recentness, version ratio, and differences between the indices, are utilized for comparing simulation results. By observing abort ratio, we are able to investigate the synchronizing patterns of SCCs. Recentness is an important index for comparing VO and OS since it shows which one is capable of providing more trustworthy data values to read requests. The size of database is indicated through version ratio. From the differences in these results, we are able to evaluate the fairness of SCCs. For instance, an SCC with less difference in response time becomes fairer one with respect to response time. In this case, its degree of availability is allowed to be highly evaluated.

We have verified the sensitivities of these indices through sensitivity analysis. The results of the indices have been compared from the viewpoints of *mpl*s, *write_pct*s, *num_sec_level*s, and different security levels.

6.4 Validation of simulation results

Simulation-based performance analysis requires estimates of average value and variance of a performance index for validating the performance results statistically. Firstly, it is generally considered important to discard statistic data collected during the initial transient phase of a simulation run. We estimate the extent of the initial transient phase through graphical procedure and use the response time of each transaction as a random variable for this procedure [13].

Secondly, the determination of run length and number of replications is important experimental design decision as well. For each run, it has the length of 2000 committed transactions after discarding first 800 ones with regard to the initial transient phase. The number of replications is set to 10 runs. Correspondingly, all the statistical data reported in this paper have 95% confidence level with the absolute error of 3%.

6.5 Simulation results and their interpretations

We now present and interpret the results of simulation experiments performed to VO, OS, and OL.

6.5.1 Effect of multiprogramming levels and write percentages

The effects of *mpl*s and *write_pct*s are evaluated in this section. Fig. 5, 6, and 7 depict the average results of SCCs with *num_sec_level*s: 1, 2, 4, 8, and 12. VO_10 in the figures indicates the average results of transactions with *write_pct* = 10% that have been synchronized by VO.

As shown in Fig. 5, all the abort ratios of SCCs go high in proportion to the increment in *mpl*s. For the most part of *mpl*s and *write_pct*s, the abort ratio of VO is lower than those of OS's and OL's since VO decides to abort a transaction if and only if it can assure that non-interfered execution of the selected transaction definitely violates 1SR.

As expected, the response times of all the SCCs increase according as *mpl*s or *write_pct*s increase (Fig. 6). Unlike abort ratios, however, response times appear to be grouped distinctly. VO outperforms explicitly OS and OL. OS outperforms OL. For the case of OL_10, its abort ratio is less than OS_20 and OS_30. However, its response time is longer than OS_20 and OS_30 since orange-lock generally forces OL to abort transactions at later points of time than OS. Accordingly, the number of operations cancelled by OL_10 becomes larger than OS_20 and OS_30.

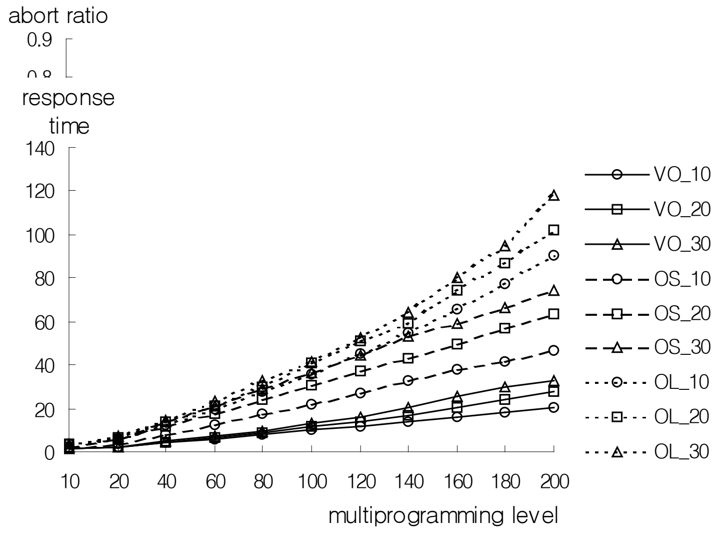


Fig. 6 Response time (num_sec_level = 1, 2, 4, 8, 12)

Fig. 7 illustrates that recentness becomes poor as *write_pct* increases in that the increased *write_pct* draws up the probability of TSR violation. In case of VO, however, such probability-based synchronization is restricted to the situation of security infringement. VO is now capable of providing more recent versions than OS.

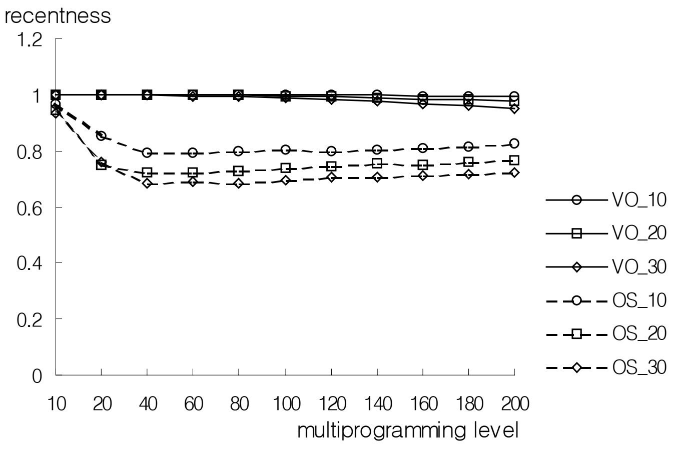


Fig. 7 Recentness (*num_sec_level* = 1, 2, 3, 4, 8, 12)

6.5.2 Effect of number of security levels

Investigating the effects of different numbers of security levels is important in that it is desirable for a security policy of an MLS information system to be enforced independently to the properties of MLS/DBMS. If enforcing an elaborate security policy conspicuously degrades the performance of an MLS information system managed by a specific MLS/DBMS, adopting such an MLS/DBMS will be hesitated.

NL4 depicted in Fig. 8 indicates that the result has been obtained from transactions with four different security levels and *write_pct* = 10%, 20%, 30%. In cases of VO and OS, all the transactions, which are responsible for aborts, have been cleared at the same security level, and besides, the number of transactions for each level decreases as *num_sec_level* increases. Accordingly, the abort ratios of VO and OS decrease as *num_sec_level* increases. Grounded on the results illustrated in Fig. 5, the abort ratio of VO is always lower than that of OS. In case of OL, on the contrary, the number of aborts provoked by orange-locks appears to increase as *num_sec_level* increases. Although the number of deadlocks decreases as *num_sec_level* increases, it is overwhelmed by that of orange-locks.

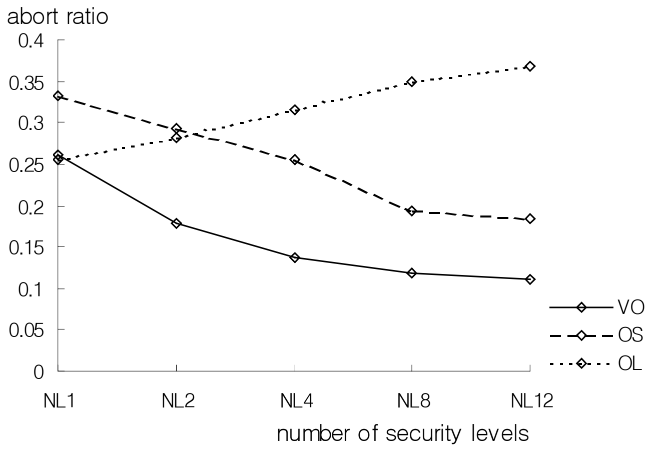
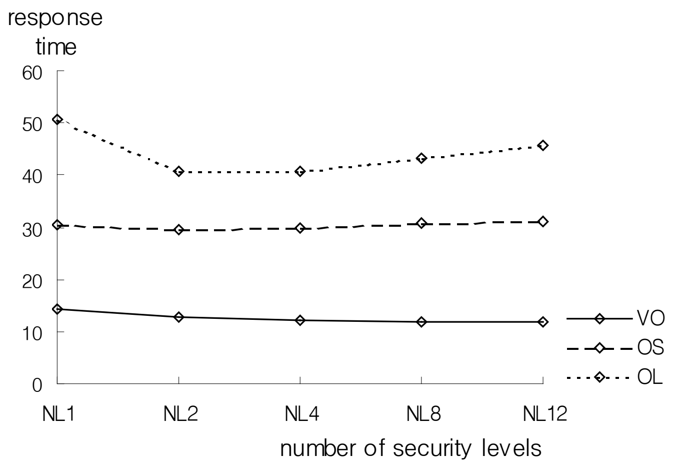


Fig. 8 Abort ratio (*write_pct* = 10%, 20%, 30%, *mpl* = 10 ~ 200)
Fig. 9 Response time (*write_pct* = 10%, 20%, 30%, *mpl* = 10 ~ 200)

For the response times of VO and OS, Fig. 9 says that they almost do not be affected by *num_sec_level*. Note that the number of transactions cleared at each security level increases in proportion to the decrement of *num_sec_level*. In cases of VO and OS, therefore, transactions cleared at small *num_sec_level* are liable to be aborted earlier than the ones cleared at large *num_sec_level*. Such early aborts reduce the number of operations cancelled and thereby counterbalance the increased abort ratios of VO and OS. Accordingly, the response times of VO and OS come to be almost invariant. On the other hand, in case of OL with two or more security levels, although deadlocks decrease as *num_sec_level* increases, the increment in orange-locks overwhelms the decrement of deadlocks. For OL in insecure DBMS, notwithstanding the absence of orange lock, owing to the absence of access control policy, the number of blocked operations and the possibility of deadlock increase. Therefore, OL in insecure DBMS reveals increased



response time.

The recentness depicted in Fig. 10 indicates that VO almost provides the most recent data versions independent of *num_sec_level*. In fact, the recentness of VO is affected only by *mpl*. In case of OS, on the contrary, the declined recentness becomes serious as *num_sec_level* increases. The rationale is that OS unilaterally forces high transactions to read outdated versions whenever there are active low transactions.

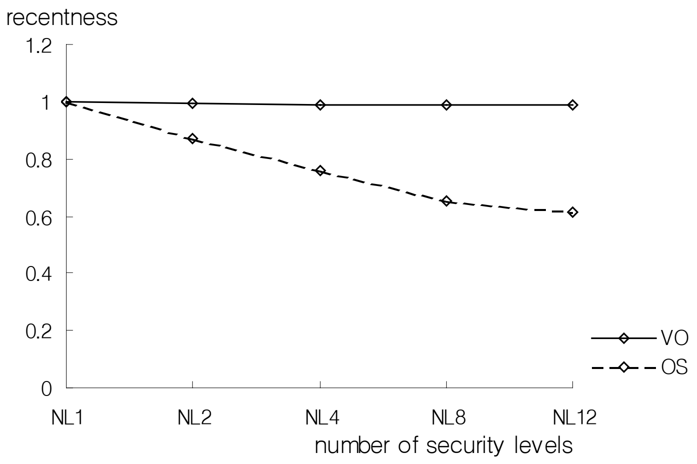


Fig. 10 Recentness (*write_pct* = 10%, 20%, 30%, *mpl* = 10 ~ 200)

6.5.3 Effect of different security levels

In this section, we evaluate the effects of 12 different security levels on the performance of SCCs. In following figures, L12 is the lowest and L1 is the highest security level. Fig. 11 illustrates that the abort ratios of VO and OS decrease as security levels are inclined to be high in that transactions in low security levels are destined to conflict each other intensively. Owing to the probability-based 1SR preservation, moreover, the conflicts of OS lead to more intensive aborts than those of VO. Transactions cleared at L12 and scheduled by OS suffer from especially serious aborts. On the contrary, the abort ratio of OL increases as the security levels of transactions become high due to the corresponding increment in orange-locks.

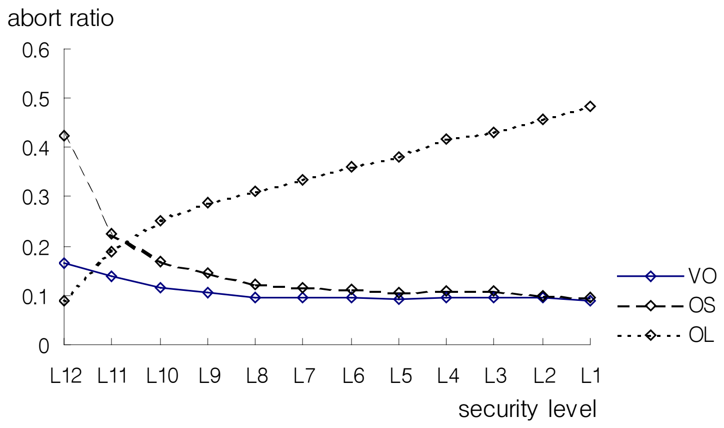


Fig. 11 Abort ratio (*write_pct* = 10%, 20%, 30%, *mpl* = 10 ~ 200)

In case of VO in Fig. 12, the differences in security levels almost do not affect the response time of VO. For OS, compared to the case of abort ratio in Fig. 11, the pattern of response time has been seriously distorted because OS has left alone unworthy versions in storage. On the contrary, although the response time of OL comprehends the effect of blocked operations, owing to the overwhelming of orange-locks, it reflects the change of abort ratio faithfully to a certain extent.

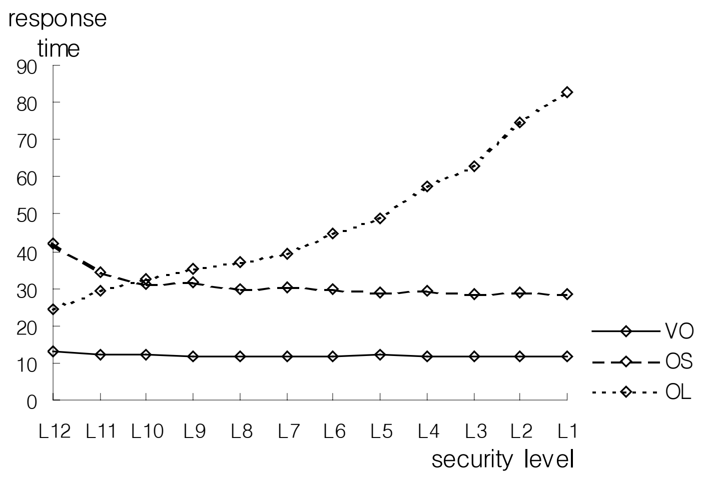


Fig. 12 Response time (*write_pct* = 10%, 20%, 30%, *mpl* = 10 ~ 200)

The recentness of VO, illustrated in Fig. 13, becomes poor as the security levels of transactions become high. Such changes, however, are very slight. In case of OS, on the contrary, there is a wide difference between the recentness of transactions cleared at L12, L11 and that of the other transactions cleared at the other low security levels. The rationale of such a wide difference is that OS positions transactions with L12 or L11 nearly at the very last. Therefore, such transactions are apt to read recent data versions. From L10, the smooth declination of OS's recentness implies that the reorganized TORs do not seriously affect the recentness.

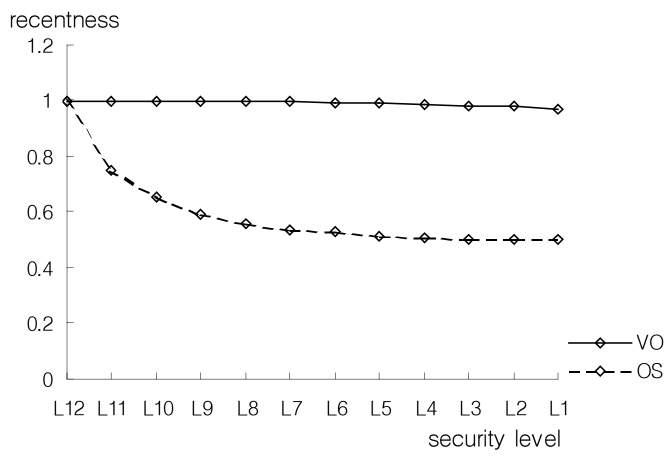
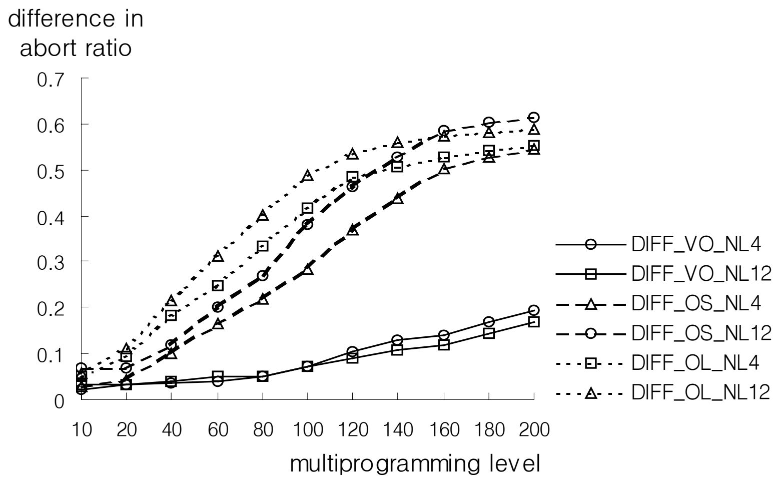


Fig. 13 Recentness (*write_pct* = 10%, 20%, 30%, *mpl* = 10 ~ 200)

6.5.4 Fairness

In this section, we investigate the difference between maximum and minimum results for



each simulation index. We obtain the results by subtracting minimum average value over different security levels and different *write_pct*s from maximum average value over them. If the difference in the results of an SCC is less than that of the other SCC, the former can be evaluated as fairer one than the latter. The rationale is that histories produced by the fair SCC are less affected by the variations in *num_sec_level*s, *write_pct*s, or security levels. Such fairness in abort ratio or response time is advantageous to suppressing the denial of service.

Fig. 14 Difference in abort ratio (*write_pct* = 10%, 20%, 30%)

DIFF_VO_NL4 and DIFF_VO_NL12 in following figures denote differences in the results of transactions synchronized by VO, cleared at four and twelve different security levels, and with *write_pct* = 10%, 20%, 30%. In Fig. 14 and 15, the differences in VO's abort ratios and response times are far less than those of OS and OL over all *mpl*s. The wide differences imply that intentional and intensive conflicting operations that are requested by malicious transactions and scheduled by OS or OL may obstruct the legitimate accesses of the other transactions. This is the typical instance of intentional denial of service. On the contrary, VO is capable of reducing such threats to availability due to its unbiased scheduling feature.

Fig. 15 Difference in response time (*write_pct* = 10%, 20%, 30%)

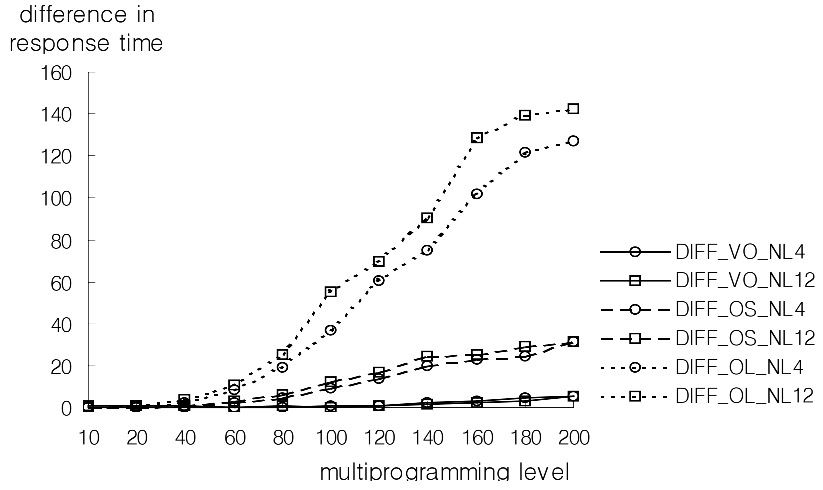
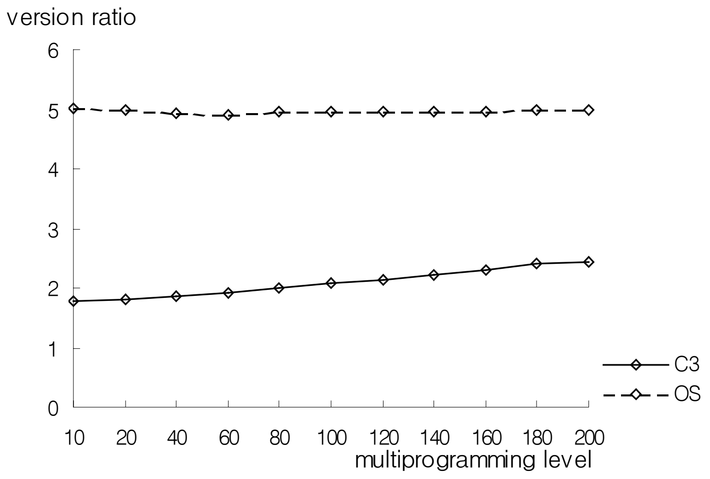


Fig. 16 depicts the difference in recentness. Compared to OS, the differences of DIFF_VO_NL4 and DIFF_VO_NL12 are far slight over all *mpl*s. The serious difference in OS's recentness implies that high transactions are forced to read far outdated versions and thereby it leads to a declined trustworthiness of MLS/DBMS.

Fig. 16 Difference in recentness (*write_pct* = 10%, 20%, 30%)

6.5.5 Effect of secure garbage collection

The enlarged database is the major obstacle to an adoption of MVDB-based SCC. It is not



only the problem of storage cost but also is the problem of response time. Secure garbage collection of VO mitigates these problems. The version ratio depicted in Fig. 17 represents the average number of sibling versions of a data item.

Fig. 17 Version ratio (*num_sec_level* = 4, *write_pct* = 20%)

Fig. 17 is the result of settings: *num_data_item* = 1000, *write_pct* = 20%, and mean *tr_size* = 10. 2000 committed transactions have been observed. As *write_pct* is set to 20%, almost 4000 writes have been processed. Every write of OS's committed transactions produces a version that is destined to be accumulated in database due to the absence of garbage collection. Since the number of data objects in database has been initialized to 1000, the number of version in database becomes 5000. In case of VO, on the contrary, the average number of siblings of a data item is approximately 2 due to secure garbage collection. Such a reduced version ratio helps VO to outperform OS in storage cost and response time.

7. Conclusion

We have proposed a secure concurrency controller, named Verified Ordering-based secure concurrency controller (VO), that founds on multiversion database. It utilizes information, which elaborately describes ordering relationship among transactions, in order to verify whether the newly arrived operation definitely violates one-copy serializability or has a high potential of opening a covert channel. For gathering the elaborated information, whenever VO receives an operation from a transaction, it verifies an ordering relationship between the transaction and the other relevant transaction and accumulates the verified ordering relationship. By referencing the information, VO is capable of guarding transactions against being aborted unnecessarily and reading excessively outdated data versions. By virtue of the information, moreover, VO is allowed to delete unworthy versions and unworthy ordering relationships among transactions. The ability for perceiving such unworthy ones has the advantage of reducing the intrinsic overhead for maintaining multiple versions and accumulated transaction ordering relationships.

We have evaluated the performance of VO in a way of comparing representative single version database-based SCC and multiversion database-based SCC. Through the evaluation, we have ascertained that VO outperforms with respects to the response time, abort ratio, recentness of versions read, and the size of database. Moreover, we have verified that VO serves transactions more fairly than the others. Such an improved fairness will be advantageous for VO to be evaluated as an SCC with higher availability. The improved recentness is expected to upgrade the trustworthiness of VO.

In principle, computer security is composed of three major aspects: secrecy, integrity, and availability. Among them, we have mainly focused on achieving secrecy in this paper. Integrity and availability have been achieved without precise modeling for them from the viewpoint of security. We therefore hope to study the issues for modeling integrity and availability in MLS information system. We also hope to study an implementation of SCC that is capable of safeguarding transactions against threats to any aspects of computer security. With such an SCC, the MLS/DBMS becomes a highly secure one.

References

[1] D. E. Bell and L. J. LaPaduda, Secure Computer Systems: Unified Exposition and Multics Interpretation, Tech. Rep. MTR-2997, The Mitre Corp., 1976.

[2] D. Ferrai, G. Serazzi, and A. Zeigner, Measurement and Tuning of Computer Systems, Prentice Hall, 1983.

[3] E. Bertino, S. Jajodia, L. Mancini, and I. Ray, Advanced Transaction Processing in Multilevel Secure File Stores, IEEE Transactions on Knowledge and Data Engineering, vol.

10, no. 1, pp. 120-135, 1998.

[4] H. Schwetman, CSIM User's Guide for Use with CSIM Revision 16, Microeletronics and Computer Technology Corporation, 1992.

[5] J. McDermott and S. Jajodia, Orange Locking: Channel-Free Database Concurrency Control Via Locking, in: *Proc. IFIP WG11.3 Working Group on Database Security*, B.M. Thuraisingham and C.E. Landwehr, eds, North-Holland, 1992, pp. 267-284.

[6] K.P. Smith, B.T. Blaustein, and S. Jajodia, Correctness Criteria for Multilevel Secure Transactions, IEEE Transactions on Knowledge and Data Engineering, vol. 8, no. 1, pp. 32-45, 1996.

[7] L. Asher and R. Hiltner, Trusted ORACLE7 Server Administrator's Guide, Oracle Corporation, 1993.

[8] L.V. Mancini and I. Ray, Secure Concurrency Control in MLS Databases with Two Versions of Data, Proc. European Symp. Research in Computer Security,.Rome, Italy, pp. 204-225, Sept. 1996.

[9] O. Costich and S. Jajodia, Maintaining Multilevel Transaction Atomicity in MLS Database Systems with Kernalized Architecture, Proc. IFIP WG11.3 Working Group on Database Security, Vancouver, Canada, pp. 207-222, Aug. 1992.

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

[11] P. Ammann and S. Jajodia, A Timestamp Ordering Algorithm for Secure, Single-Version, Multi-Level Databases, Proc. IFIP WG11.3 Working Group on Database Security, West Virginia, USA, pp. 191-202, Nov. 1991.

[12] P. Ammann, F. Jaeckle, and S. Jajodia, A Two Snapshot Algorithm For Concurrency Control in Multi-Level Secure Databases, Proc. IEEE Computer Society Symp., Research in Security and Privacy, Oakland, California, U.S.A., pp. 204-215, May 1992.

[13] P. D. Welch, The Statistical Analysis of Simulation Results in Computer Performance Modeling Handbook, ed. S. Lavenberg, Academic Press, 1983.

[14] P. S. Yu, D. M. Dias, and S. S. Lavenberg, On the Analytical Modeling of Database Concurrency Control, Journal of the ACM, vol. 40, no. 4, pp. 831-872, 1993.

[15] R. Agrawal, M. J. Carey, and M. Livny, Concurrency Control Performance Modeling: Alternatives and Implications, ACM Trans. Database Systems, vol. 12, no. 4, **12**(4), pp. 609 – 654, 1987.

[16] S. Castano, M. Fugini, G. Martella, and P. Samarati, Database Security, Addison-Wesley, 1995.

[17] S. Jajodia and V. Atluri, Alternative Correctness Criteria for Concurrent Execution of Transactions in Multilevel Secure Databases, Proc. IEEE Computer Society Symp., Research in Security and Privacy, Oakland, California, U.S.A., pp. 216-225, May 1992.

[18] S. Jajodia, L. V. Mancini, and I. Ray, Secure Locking Protocols for Multilevel Database Management Systems, Proc. IFIP WG11.3 Working Group on Database Security, Como, Italy, pp. 177-194, July 1996.

[19] T. F. Keefe and W.T. Tsai, Multiversion Concurrency Control for Multilevel Secure

Database Systems, Proc. IEEE Computer Society Symp., Research in Security and Privacy, Oakland, California, U.S.A., pp. 369-383, May 1990.

[20] V. Atluri, E. Bertino, and S. Jajodia, Achieving Stricter Correctness Requirements in Multilevel Secure Databases, Proc. IEEE Computer Society Symp., Research in Security and Privacy, Oakland, California, U.S.A., pp. 135-147, May 1993a.

[21] V. Atluri, E. Bertino, and S. Jajodia, Achieving Stricter Correctness Requirements in Multilevel Secure Databases: The Dynamic Case, IFIP WG11.3 Working Group on Database Security, Lake Guntersville, U.S.A., pp. 135-158, Sept. 1993b.

[22] V. Atluri, S. Jajodia, and T.F. Keefe, Multilevel Secure Transaction Processing: State and Prospects, Proc. IFIP WG11.3 Working Group on Database Security, Como, Italy, pp. 79-98, July 1996.

[23] W. T. Maimone and I.B. Greenberg, Single-Level Multiversion Schedulers for Multilevel Secure Database Systems, Proc. 6th Annual Computer Security Applications Conference, Tucson, Arizona, U.S.A., pp. 157-180, Dec. 1990.

Yonglak Sohn is an Assistant Professor in the Department of Computer Engineering. He received the B.S. degree in Computer Science from Kyeongbuk National University, Korea, in 1986 and the M.S. degree in Computer Science from Korea University, Korea, in 1988. He is currently a Ph. D candidate at Korea Advanced Institute of Science and Technology (KAIST). His principle research interest is security in database management systems, including secure query processing, secure concurrency control in centralized and distributed environments, and security modeling.

Songchun Moon received his Ph. D in Computer Science from the University of Illinois at Urbana-Champaign in 1985. He has been working for KAIST since then. He was a distinguished scholar at the Hungarian Academy of Science and still serves for EUROMICRO as a director. He has developed a multi-user relational database management system IM, which is the first prototype ever in Korea in 1990 and also a distributed database management system DIME in 1992, which is another first prototype ever in Korea. Currently, he is involved in developing a secure DBMS.