Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Systems 30 (2005) 151–166

www.elsevier.com/locate/infosys

A formal approach to lock escalation ☆

Ji-Woong Chang[a,*], Kyu-Young Whang[a], Young-Koo Lee[a], Jae-Heon Yang[a], Yong-Chul Oh[b]

[a] *Department of Computer Science and Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology (KAIST), 373-1, Kusong-Dong, Yusong-gu, Taejon 305-701, South Korea*
[b] *Department of Computer Engineering, Korea Polytechnic University, South Korea*

Received 14 December 2001; received in revised form 20 October 2003; accepted 24 October 2003

Abstract

Since database management systems(DBMSs) have limited lock resources, transactions requesting locks beyond the limit must be aborted, degrading the performance abruptly. Lock escalation is considered a solution to this problem. However, existing lock escalation methods have been designed in an ad hoc manner. So, they do not provide a complete solution. In this paper, we propose a formal model of lock escalation. Using the model, we analyze the roles of lock escalation formally and solve the problems of the existing methods systematically. In particular, we introduce the concept of the *unescalatable lock* that cannot be escalated due to conflicts. We identify that the unescalatable lock is the major cause of exhausting lock resources. We then analyze the reasons why unescalatable locks are generated and propose a new lock escalation method, *adaptive lock escalation*, which controls lock escalation based on the number of unescalatable locks. Through extensive simulation, we show that adaptive lock escalation significantly outperforms existing methods reducing the number of aborts and the average response time and increasing the throughput to a great extent. Adaptive lock escalation drastically reduces (more than 10 fold) the number of lock resources required to maintain the same level of throughput and average response time. At the same time, the throughput and average response time when using adaptive lock escalation are rather insensitive to the number of lock resources. Existing methods rely on users to estimate this number accurately at system initialization time. Adaptive lock escalation greatly alleviates this burden.

Keywords: Concurrency control; Lock escalation; Database management systems

1. Introduction

Locking is a widely used technique for concurrency control in DBMSs. In most cases, locks are managed in shared memory. Normally, the system's shared memory cannot be allocated dynamically; instead, it is preallocated at system initialization time. Therefore, it is important for the system administrator to estimate accurately the

☆ Recommanded by Bettina Kemme.

*Corresponding author. Fax: +82-42-869-3510.
E-mail addresses: jwchang@mozart.kaist.ac.kr (J.-W. Chang), kywhang@mozart.kaist.ac.kr (K.-Y. Whang), yklee@mozart.kaist.ac.kr (Y.-K. Lee), jhyang@cs.kaist.ac.kr (J.-H. Yang), oh@kpu.ac.kr (Y.-C. Oh).

0306-4379/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/j.is.2003.10.009

size of the shared memory required by the DBMS for locking.

Since the size of the shared memory is fixed at runtime [1], DBMSs have limited lock resources[1]

When locks are requested excessively, the transactions that are not able to secure locks should be aborted [2]. We call this situation *lock resource exhaustion*. If the amount of lock resources is underestimated, lock resource exhaustion easily occurs. It is very difficult to accurately estimate the amount of lock resources needed because the number of locks depends on the number of concurrent transactions and the number of locks acquired by each transaction, which are widely varying according to not only the types of the applications but also the size of the database.

Lock resource exhaustion may cause a transaction to fall in cyclic restart [3,4]. In the worst case, all transactions could fall in cyclic restart and none of them commits. We call this situation *live halt*. Theoretically, live halt can occur even though there are a large amount of lock resources if a proportionately large number of transactions are running concurrently. Live halt is a fatal situation where the system becomes paralyzed.

We may consider various methods to solve the lock resource exhaustion. A query processor can determine the lock granularity before initiating a transaction or an aborted transaction can use coarse lock granularity when it is restarted. Those methods can solve the problem partially. However, they cannot solve the problem completely since they use ad hoc approaches. There has been no formal approach proposed in the literature. In this paper, we propose a solution with a formal model.

Lock escalation used with multigranularity locking [5] is considered a solution to lock resource exhaustion. Lock escalation reduces the number of locks in use by converting many fine granularity locks to a coarse granularity lock [4]. However, one should not execute needless lock escalation since it decreases the concurrency by increasing the lock granule [6]. Existing lock escalation methods [6,7] use a kind of local approach, in which individual transactions determine whether or not to execute lock escalation.

This local approach has several problems. First, it decreases the concurrency by executing needless lock escalations, even when there are sufficient extra lock resources available. Second, it causes needless transaction aborts by preventing the transactions from executing lock escalation, even if there are no more extra lock resources. Third, it cannot prevent live halt.

As a simple solution, we may consider a global approach, *global lock escalation* [8], in which the system determines whether or not to execute lock escalation based on the total number of locks. However, this method does not solve the problems completely as we see in Section 2.2.

Even though lock resource exhaustion has been introduced in several literatures [2,4,6,9], it has been overlooked. To handle the problem of lock resource exhaustion, many DBMSs rely on the system administrators. They stop all locking operations and wait the system administrators to reset the maximum number of lock resources [9]. They also recommend the users to make transactions shorter or to lower the degree of isolation [6]. These kinds of remedy are not desirable since a lot of burden is imposed on the users, who are not knowledgeable on the system's internals.

Although lock escalation has been employed in many conventional DBMSs, there has been no formal model of lock escalation proposed in the literature. In this paper, we propose a general model for lock escalation by analyzing systematically the underlying mechanisms of lock escalation. Especially, we present the new notion of the *unescalatable lock*, which cannot be escalated due to conflicts, and show that the unescalatable lock is the determining factor in lock resource exhaustion. We further analyze the reasons why unescalatable locks are generated and then propose a new lock escalation method, *adaptive lock escalation*, that controls lock escalation based on the number of unescalatable locks. Adaptive lock escalation significantly increases the number of concurrent transactions allowable given the same amount of lock resources without sacrificing performance. Moreover, it drastically reduces the number of lock resources required for the same number of

[1] We use the term *lock resource* to distinguish free locks in the lock pool from the locks actively in use.

concurrent transactions. At the same time, the throughput and the average response time of adaptive lock escalation is rather insensitive to the maximum number of lock resources. This effect is very useful for self-tuning DBMSs since the accuracy in estimating the maximum number of lock resources is not critical for performance. Adaptive lock escalation guarantees there be no live halt under excessive lock requests gradually transiting to a serial execution of transactions.

The paper is organized as follows. In Section 2, we briefly review multigranularity locking and existing lock escalation methods. In Section 3, we propose a general model for lock escalation and introduce the notion of the unescalatable lock. In Section 4, we identify the causes that generate unescalatable locks and propose techniques to counter them. In Section 5, we present the adaptive lock escalation algorithm that utilizes the concept of the unescalatable lock. In Section 6, we perform extensive experiments through simulation. Finally, in Section 7, we summarize the results and conclude the paper.

2. Backgrounds

2.1. Multigranularity locking and lock escalation

Multigranularity locking provides several lock granules in a DBMS to allow a transaction to determine lock granularity for itself. These lock granules form a hierarchy(*lock hierarchy*, in short). The lock on a higher-level granule implicitly locks those at the lower levels. Coarse granularity has the advantage of low locking overhead when accessing a large number of records, but has the disadvantage of low concurrency. In contrast, fine granularity has the disadvantage of high locking overhead, but has the advantage of high concurrency [5,10,11].

Refs. [11–13] have reported the simulation results on which lock granularity should be used for better performance depending on the characteristics of the transactions. Nevertheless, there has not been significant research on lock escalation as a method for managing lock resources.

Most DBMSs using multigranularity locking provide a two-level hierarchy involving the file and the record as lock granules. We also assume a two-level hierarchy using the term *file* for the coarse granule and the term *record* for the fine granule to help readers understand this paper. However, the proposed model and the lock escalation algorithm can be adapted to more-than-two-level hierarchies of lock granules.

Since a DBMS cannot predict how many records a transaction will access without additional information, it tends to favor record locks and then adopt file locks only on specific demand [4]. As the number of record locks acquired by transactions grows, all the locks may become in active use. Lock escalation can be used to alleviate this situation. Lock escalation consists of two steps: lock conversion and lock release. *Lock conversion* is the step for converting the mode of the lock (*lock mode*, in short) on the file from intension shared (IS) mode to shared (S) mode or from intention exclusive (IX) mode to exclusive (X) mode. *Lock release* is the step for releasing all the locks on the records that belong to the file. Lock escalation is especially useful in practice when there are no more lock resources remaining.

2.2. Existing lock escalation methods

Since existing lock escalation methods have been largely designed in an ad hoc manner, it is difficult to come up with a nice classification. However, we attempt to classify them into three categories: (1) lock escalation based on the number of locks per transaction and per file (*LETF*) and (2) lock escalation based on the number of locks per transaction (*LET*), and (3) lock escalation based on the total number of locks (*Global*).

In LETF, a transaction executes lock escalation for a file when it requests record locks that belong to that file over the predetermined lock escalation threshold [7]. In LET, a transaction selects one of the files it accesses and executes lock escalation for that file when the total number of locks it requests goes over the lock escalation threshold [6]. Both LETF and LET have the following problems because transactions execute lock escalation individually without global consideration.

First, they might execute needless lock escalation when a transaction requests record locks over the threshold even though there are extra lock resources available. Second, a transaction might not execute lock escalation because the number of locks requested is less than the threshold even when there are no more lock resources available. If many transactions execute concurrently, lock resources could be exhausted even though no one transaction requests locks over the threshold. LET can possibly alleviate this problem by allowing the transaction requesting a lock to execute lock escalation when there are no more lock resources available even though it has requested locks below the threshold [6]. Considering the total number of locks as in this method, however, does not solve the problem completely. When there are no more lock resources remaining, but the transaction is not able to execute lock escalation because of lock conflicts, the transaction is aborted even though it is still possible that other transactions execute lock escalation instead. These problems are caused by not considering the global status of the lock resources. A naive method to solve the problems of LETF and LET, which are based on local decisions, would be a global approach selecting an appropriate transaction and a file to execute lock escalation when the total number of locks exceeds the lock escalation threshold. We call this method *global lock escalation* [8]. Here, lock escalation threshold is set to a value smaller than the maximum number of lock resources (say, 80%). The reason for this is as follows. Suppose the lock escalation threshold is set to the maximum number of lock resources. Then, when the total number of locks reaches the lock escalation threshold, if the transaction is not able to execute lock escalation due to lock conflicts, it must be aborted. If the threshold is smaller, however, the transaction can continue using the extra lock resources while waiting for the lock conflict to be resolved.

Global lock escalation does not have the problem of aborting a transaction unnecessarily since it considers the total number of lock resources available and globally determines the transaction to execute lock escalation and the target file. Nevertheless, it still has drawbacks. It reserves some extra lock resources for the situation where transactions cannot execute lock escalation due to lock conflicts. However, those reserved lock resources are not used unless this situation occurs. This means that the effective amount of lock resources is reduced by the reserved amount in normal cases. Therefore, transactions execute lock escalation early decreasing concurrency. Moreover, global lock escalation cannot prevent cyclic restart and live halt because the reserved lock resources can also be exhausted.

3. A formal model for lock escalation

In this section we present a formal model of lock escalation. We make a simplifying assumption without loss of generality. A transaction does not request an S or X mode lock on a file except when the request is made through lock escalation. We define the states of a file according to how lock escalation can be handled.

Definition 1. A *free state* is a state of the file where no locks are held by any transaction. A *free file* is a file in a free state.

Definition 2. An *escalatable state* is a state of the file, the locks held on whose records can be escalated without causing lock conflict. An *escalatable file* is a file in an escalatable state. An *escalatable lock* is a record lock that is held on a record in an escalatable file.

Definition 3. An *unescalatable state* is a state of the file, the locks held on whose records cannot be escalated because of lock conflict. An *unescalatable file* is a file in an unescalatable state. An *unescalatable lock* is a record lock that is held on a record in an unescalatable file.

Definition 4. A *fully escalated state* is the state of the file, on which only file locks are held. Therefore, there is no record lock that belongs to the file in the fully escalated state. A *fully escalated file* is a file in a fully escalated state.

We can identify the state of a file by the combination of the lock modes on the file. Typical

lock modes for a file are the S, X, IS, and IX modes. Table 1 represents the compatibility between lock modes [4]. The row means the lock mode acquired, and the column the one requested. Table 2 shows the relationship between the states of a file and the combinations of the lock modes on the file. It contains all possible combinations feasible under the assumption made earlier.

The combination of the lock modes on the file, and accordingly, the state of the file varies by granting a new lock, releasing a lock, and executing lock escalation. Fig. 1 shows the state transition diagram for a file. In Fig. 1, each node represents the combination of the lock modes on the file. NL (no lock) means that there is no granted lock on the file. There are five kinds of arrows in Fig. 1. A double line arrow means lock escalation; a single solid line arrow granting a lock; and a single broken line arrow releasing a lock. The label on the arrow indicates the lock mode except for 'E', which means lock escalation.

Table 1
The compatibility between lock modes

	IS	IX	S	X
IS	T	T	T	F
IX	T	T	F	F
S	T	F	T	F
X	F	F	F	F

T: compatible; F: incompatible.

Table 2
The state of a file and the possible combinations of the lock modes on the file

State of a file	Combination of the locks modes on the file
Free state	None
Fully escalated states	$\{X\}^1$ $\{S\}^{1+}$
Unescalatable states	$\{IX\}^{2+}$ $\{IX\}^{1+}$, $\{IS\}^{1+}$
Escalatable states	$\{IX\}^1$ $\{IS\}^{1+}$ $\{S\}^1$, $\{IS\}^{1+}$

A^1: only one A mode lock is granted; A^{n+}: n or more A mode locks are granted; A^{n+}, B^{m+}: n or more A more locks and m or more B mode locks are granted.

In Fig. 1, the nodes $[\{IX\}^1]$, $[\{IS\}^{1+}]$, and $[\{S\}^{1+}, \{IS\}^{1+}]$ all represent escalatable states, but there are minute differences. The node $[\{IX\}^1]$ or $[\{IS\}^{1+}]$ is converted to an unescalatable state when a new lock is granted: the node $[\{IX\}^1]$ can be converted to $[\{IX\}^{2+}]$ or $[\{IX\}^{1+}, \{IS\}^{1+}]$, which is an unescalatable state, when an IX lock or IS lock is newly granted; the node $[\{IS\}^{1+}]$ to $[\{IX\}^{1+}, \{IS\}^{1+}]$, an unescalatable state, when an IX lock is granted. In contrast, the node $[\{S\}^{1+}, \{IS\}^{1+}]$ is not directly converted to an unescalatable state by one lock request because a request on an IX lock will not be granted due to conflict with the S lock. From the observation, we can classify the escalatable states into two categories as in Definitions 5 and 6.

Definition 5. An *unsafe escalatable state* is an escalatable state that can be directly converted into an unescalatable state. An *unsafe escalated file* is a file in an unsafe escalatable state.

Definition 6. A *safe escalatable state* is an escalatable state that cannot be directly converted into an unescalatable state. A *safe escalatable file* is a file in a safe escalatable state.

Fig. 2 is a simplified version of Fig. 1 redrawn with an emphasis on the file states. Nodes represent states of the file with arrows having the same meaning as in Fig. 1. Bullet-ended solid and broken lines are added to represent granting and releasing record locks. These arrows for record locks are needed for explaining the causes of generating unescalatable locks in Section 4, but do not affect the state of the file.

Let us make a few interesting observations on lock escalation. First, we note that lock escalation can be executed only for an escalatable file, but cannot be done for a free, fully escalated, or unescalatable file. Lock escalation for an unescalatable file conflicts with other locks, and the transaction will be blocked until all conflicting locks are released converting the file state to an escalatable state. Second, lock resource exhaustion only occurs when there are

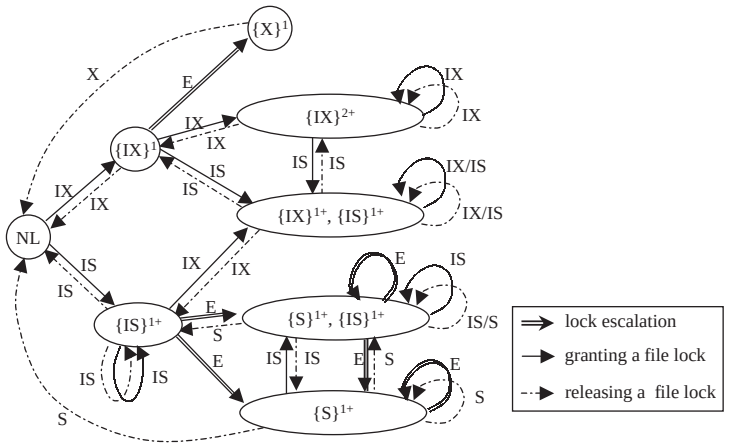

Fig. 1. State transition diagram for a file.

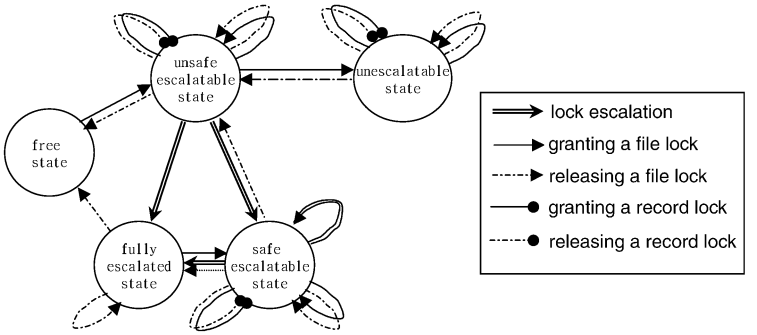

Fig. 2. The simplified state transition diagram of the file.

no more escalatable locks because locks can always be made available by executing lock escalation as long as there are escalatable locks remaining. Therefore, we conclude that the unescalatable lock is the determining factor in lock resource exhaustion and that the lock resources must be managed based on the number of unescalatable locks rather than simply based on the total number of locks.

4. Unescalatable locks

In this section, using the proposed lock escalation model, we analyze the reasons why unescalatable locks are generated. We then examine the effect of lock escalation on the growth of unescalatable locks and propose the mechanisms to control the growth of unescalatable locks.

Fig. 3 is the part of Fig. 2 that involves the unescalatable state. The arrows labeled 'Cause 1', 'Cause 2', and 'Cause 3' represent different causes of the growth of unescalatable locks.

Cause 1 (*State conversion*): As the state of the file is converted from an unsafe escalatable state to an unescalatable state, all the locks that belong to the file are converted from the escalatable locks to unescalatable ones.

Cause 2 (*Granting additional unescalatable locks*): As a transaction accessing an unescalatable file acquires additional record locks, these locks automatically become unescalatable locks.

Cause 3 (*Increasing the number of transactions*): As transactions accessing an unescalatable file are

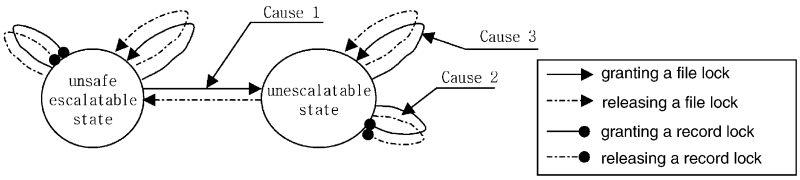

Fig. 3. The part of the state transition diagram that involves the unescalatable state.

added, the record locks they acquire become unescalatable locks.

The existing research has treated lock escalation simply as a method of controlling the number of locks being actively used regardless whether they are escalatable or unescalatable. Here, based on the proposed lock escalation model, we analyze the effect of lock escalation on unescalatable locks. The double line arrows starting from the unsafe escalatable state in Fig. 2 represent the lock escalations that convert a file in the unsafe escalatable state to one in the safe escalatable state or in the fully escalated state. These transitions eliminate Cause 1 by preventing the file state from drifting to the unescalatable state. In contrast, the lock escalation at the safe escalatable state only plays the role of releasing the locks. Therefore, to suppress the growth of unescalatable locks, we need to execute lock escalation for a file in the unsafe escalatable state. Since the existing lock escalation methods do not distinguish the states of a file, they cannot effectively control the number of unescalatable locks.

Now, based on the above analysis, we consider mechanisms to counter the causes for the growth of unescalatable locks. We can eliminate all causes by blocking the lock requests that generate unescalatable locks.

First, as a solution for eliminating Cause 1, we propose the notion of *semi-lock escalation* to block the lock requests that convert the file state from the unsafe escalatable state into the unescalatable state.

Definition 7. *Semi-lock escalation* is lock escalation in which only the first step (lock conversion) is executed.

Semi-lock escalation is a simple operation that eliminates Cause 1 like regular lock escalation. The part of lock escalation effective in eliminating Cause 1 is the first step (lock conversion) defined in Section 2. The first step changes the combination of lock modes on the file, and accordingly, blocks the lock requests that convert the file state into the unescalatable state. Therefore, to control the number of unescalatable locks, executing the first step suffices.

Next, as a solution for eliminating Cause 3, we propose the notion of *meta-locking* to block the lock requests for the unescalatable file. Meta-locking eliminates Cause 3 by explicitly prohibiting the grant of new locks on an unescalatable file, i.e., by blocking the transactions requesting new locks even if the modes of the new locks are compatible with existing lock modes.

Definition 8. *Meta-locking* is an operation that prevents a new lock from being granted regardless of the compatibility of lock modes. *Meta-unlocking* is the reverse operation cancelling the effect of meta locking.

Finally, as a solution to Cause 2, we propose *selective relief*. While Causes 1 and 3 are due to granting a new file lock, Cause 2 is due to granting new record locks. In spite of semi-lock escalation and meta-locking, the number of unescalatable locks may increase due to Cause 2, and lock resource exhaustion could happen. To solve this problem, when there are no more lock resources available, we may block the transaction requesting a lock until some locks are returned instead of aborting the transaction. However, this method does not solve the problem completely. The reason is as follows. If there is no escalatable file, lock escalation cannot be executed, and locks are returned only when a transaction terminates. If all the transactions request locks, however, all of

them are blocked, and locks are not returned. Thus, the only way to resolve the situation is to select a victim and abort it. Now, we define selective relief in Definition 9.

Definition 9. *Selective relief* is a method that guarantees completion of a transaction by excepting it from the candidates for victims and by executing lock escalation on all the files it accesses. We call this transaction the *immortal transaction*. To guarantee the completion of the immortal transaction, all the transactions having locks conflicting with lock escalation as well as acquisition of new locks by the immortal transaction are aborted. In case the immortal transaction accesses a new file, it also executes lock escalation for the file.

By definition, the immortal transaction does not have lock conflicts any longer and will not wait for the lock, due to lock conflict, or lock resource. Since at least one transaction, i.e., the immortal transaction, can complete without getting into cyclic restart, it is guaranteed that the system does not become paralyzed.

We note that the immortal transaction reduces the number of unescalatable locks. Aborting the transactions that have locks conflicting with lock escalation of the immortal transaction converts the state of the file for which lock escalation is executed from unescalatable to escalatable. Thus, all locks belonging to the file are converted from being unescalatable to escalatable.

The concepts of semi lock escalation, meta locking, and selective relief are rather simple. However, we note that the contributions of those operations are significant in that they have been derived from formal analysis for the purpose of controlling the number of unescalatable locks.

5. Adaptive lock escalation

In this section, based on the proposed model and the mechanisms, we propose a new lock escalation method that we call *adaptive lock escalation*. Adaptive lock escalation extends global lock escalation by adopting the notion of the unescalatable lock and by determining execution of lock escalation based on the number of unescalatable locks. In addition to this basic extension, adaptive lock escalation uses semi lock escalation and meta locking to suppress the growth of unescalatable locks more effectively. If there are no more lock resources available in spite of using these methods, selective relief is used.

We present the adaptive lock escalation algorithm in Fig. 4. The algorithm consists of three parts. The first part is activated by each lock request operation; the second by the demon process detecting the situation where all the transactions are blocked waiting for the lock or lock resource; the third by each lock release operation.

In the first part, in step A1, the transaction requesting a lock controls the number of unescalatable locks with semi-lock escalation (in step A1.1) and meta locking (in step A1.2) when the lock escalation threshold is exceeded. In step A1.1, we execute semi lock escalation for all unsafe escalatable files to prevent their states from being converted to the unescalatable state (eliminating Cause 1). In step A1.2, we execute meta locking for all unescalatable files to prevent new transactions from accessing the file. This prevents increase of transactions accessing these files (eliminating Cause 3). Step A2 describes what to do when there are no more lock resources available. Step A2.1 is for the case where there are files for which semi-lock escalation has been executed, but lock release has not. In this case, we select one of these files and complete lock escalation by executing lock release to get lock resources returned. Step A2.2 is for the case where there are no files for which semi-lock escalation has been executed, but there are escalatable files. In steps A2.1 and A2.2, we select a safe escalatable file that releases the largest number of record locks.

This case can occur since, even though all the lock resources are used, the number of unescalatable locks may not exceed the threshold. In this case, we select one of these files and execute lock escalation to get lock resources returned. Step A2.3 is for the case where there are no files for which semi-lock escalation has been executed and there are no escalatable files. In this case, the transaction requesting the lock is blocked until

ALGORITHM Adaptive Lock Escalation

/* Part A: activated by each lock request operation */

A1. If the total number of unescalatable locks exceeds the lock escalation threshold, DO

A1.1 Execute semi lock escalation for all escalatable files.

A1.2 Execute meta locking for all unescalatable files.

A2. If no more lock resources are available, DO

A2.1 If there are files for which semi lock escalation has been executed, but lock release has not, select one file, execute lock release (i.e., complete lock escalation), and return.

A2.2 If there are escalatable files, select one, execute lock escalation, and return.

A2.3 Otherwise, block waiting for the lock resource.

/* Part B: activated by the demon process */

B. If no more lock resources are available, there are no escalatable files, and all the transactions are blocked waiting for the lock or the lock resource, perform selective relief by electing the immortal transaction[2] and by aborting all the transactions conflicting with the immortal transaction.

/* Part C: activated by each lock release operation */

C. If the total number of unescalatable locks decreases below the lock escalation threshold, DO

C.1 Execute deescalation for the files for which lock release has not yet been executed.

C.2 Execute meta unlocking for all the files for which lock blocking has been executed.

Fig. 4. The adaptive lock escalation algorithm.

some locks are released since it cannot secure a lock resource.

The second part (step B) is activated only when there are no more lock resources available, all the transactions are blocked waiting for the lock or lock resource, and there are no escalatable files. In this case, we use selective relief.[2]

The third part (step C) is the action to take when the number of unescalatable locks decreases below the threshold. In step C.1, we undo semi lock escalation (i.e., deescalate) by reverting the lock modes of the files for which lock release has not been performed yet. In step C.2, we execute meta unlocking reverting the actions taken in step A1. These actions allow all the transactions that have been put on hold through semi-lock escalation or meta locking to continue.

Activation of mechanisms in adaptive lock escalation such as semi lock escalation or meta-locking is based on the total number of unescalatable locks rather than on the total number of locks. Thus, as long as the number of unescalatable locks is held below the threshold, the total number of locks is free to exceed the threshold being only limited by the total amount of lock resources. This means that adaptive lock escalation does not have the problem that lock resources are underutilized as in global lock escalation.

[2] Here, we simply elect the oldest transaction as the immortal transaction. Many heuristic methods can be developed for optimally electing the immortal transaction. Since the details are not the focus of this paper, we leave it as a further study.

In the algorithm, we assume a two-level hierarchy using the file and the record locks. In case where the lock hierarchy consists of more than two levels, we can extend the algorithm in a straightforward manner. The possible states of a higher-level lock granule are the same as those of a file defined in Chapter 3 since they have the same possible combinations of the lock modes. Thus, we can adapt the proposed lock escalation model to all higher-level lock granules. Semi-lock escalation and meta-locking can be applied to all levels of lock granules. For example, in step A1, we execute semi-lock escalation for all escalatable higher-level granules and meta-locking for all unescalatable higher-level granules.[3]

The adaptive lock escalation algorithm can be easily implemented in the existing lock manager.[4] Semi-lock escalation can be implemented in a way similar to lock conversion. Meta locking can be implemented by setting a flag indicating that a new lock for the given file must be blocked regardless of the compatibility of lock modes. Selective relief can be implemented by extending the deadlock resolution algorithm choosing as victims all the transactions conflicting with the immortal transaction.

The overhead of running the adaptive lock escalation algorithm is negligible. The reasons are as follows. The number of unescalatable locks must be maintained whenever a transaction acquires a lock, releases a lock, or executes lock escalation. The operation for maintaining the number of unescalatable locks consists of checking the status of the given file and changing the counter for the number of unescalatable locks. This operation is much simpler than the operation of acquiring a lock, releasing a lock, or executing lock escalation. Moreover, semi-lock escalation is simpler than lock escalation since it does not execute lock release, the second step of lock escalation, whose overhead is heavier. Meta locking is also simple since it just sets a flag and checking the flag is a small fraction of locking. Selective relief can have some overhead since it chooses many transactions as victims to prevent cyclic restart. However, this overhead is much lighter than that of cyclic restart, which would occur otherwise.

[3] In case where the lock hierarchy consists of more than two levels, many heuristic methods can be developed for optimally selecting lock granules for which lock escalation will be executed. However, the details are not the focus of this paper, and we leave it as a further study.

[4] The algorithm has been implemented in the storage manager of the Odysseus object-relational DBMS being developed at KAIST [14].

6. Performance evaluation

We have performed extensive experiments. Section 6.1 describes system architecture and environment used for the experiments; Section 6.2 presents the experimental results.

6.1. System architecture and environment

Fig. 5 shows the experimental system consisting of a server and multiple clients. The transaction generator, a component of the client, generates transactions one after another as soon as the previous one commits.

The server consists of a transaction manager, lock manager, and resource manager. The transaction manager executes the transactions generated by clients. The lock manager handles the transactions' lock requests. The lock escalation manager, being a part of the lock manager, handles lock escalation. The resource manager manages requests for the CPU and disks in the first-come-first-served (FCFS) manner. The lock manager used for the experiments is a real implementation. However, to experiment various cases by controlling the parameters, the transaction manager and the resource manager are implemented as simulators.

Transactions follow strict 2-phase locking. We use the multigranularity locking protocol with the file and the record locks forming the lock hierarchy. We assume that there are two types of transactions: read-only and read-and-write. The former requests only an IS mode lock for the file and the latter only an IX mode lock. An S or X mode lock for the file is requested only by lock escalation. We also assume that a transaction is aborted only when it becomes a victim or when it

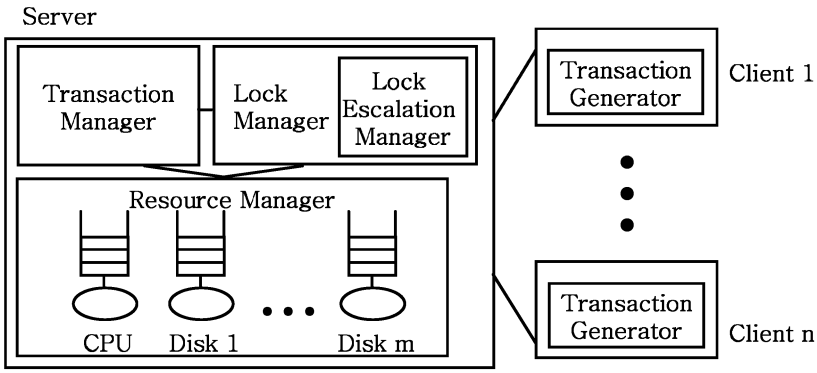

Fig. 5. Experimental system architecture.

is not able to get the lock it needs, but is not aborted voluntarily. An aborted transaction is restarted immediately.

Table 3 shows the system parameters used in the experiments. The database consists of 100 files, each containing 100 000 records stored in five disks. The number of files accessed by each transaction is 2. The number of records accessed by each transaction has an exponential distribution with the average of 100. This number is the same as the number of record locks that would be acquired by the transaction if it did not execute lock escalation. The operations of a transaction consist of (1) the CPU operations of the user transaction, (2) disk I/O's, and (3) locking and lock escalation operations. We assume that the CPU operations and disk I/O's of the user transaction among algorithms are identical with a ratio of 1:1. The lock escalation operation is not part of the user transaction, and thus, is not included in the CPU/disk ratio of the user transaction. On the other hand, the locking and lock escalation time is incorporated as the real time spent for locking and lock escalation operations in the total elapsed time measured.

We compare five cases: (1) no lock escalation done (No escalation); (2) lock escalation based on locks per transaction and per file (LETF); (3) lock escalation based on locks per transaction (LET); (4) global lock escalation (Global), and (5) adaptive lock escalation (Adaptive). The lock escalation threshold for LETF is set to 80% of the average number of records in a file accessed by a transaction, and that for LET to 80% of the average total number of records accessed by a transaction. The threshold for Global and Adaptive is set to 80% of the total number of lock resources.

Table 3
System parameters

Number of files	100
Number of records per file	100 000
Number of disks	5
Number of files accessed by a transaction	2
Number of records accessed by a transaction	Exponential distribution with the average of 100
CPU cost per operation	3 ms
DISK I/O cost per operation	9 ms
Buffer hit ratio	0.66

A run in an experiment continues until 10 000 transactions commit. In Experiment 1, in a commercial DBMS [9], we measure the number of concurrent transactions causing lock resource exhaustion as we vary the number of lock resources provided by the DBMS from 1000 to 4000. In Experiment 2, we measure the performance as we vary the number of concurrent transactions when the number of lock resources provided by the DBMS is set to 1000. We use the average number of aborts per committed transaction (= the total number of aborts/the number of commits), the average response time for the transactions, and the number of transactions that

commit per unit time (throughput) as the performance measures. In Experiment 3, we measure the throughput and average response time as we vary the number of lock resources provided by the DBMS from 1000 to 100 000, when 512 transactions are running concurrently. In Experiment 4, we compare the throughputs between Adaptive and Global with different amount of lock resources.

6.2. Experimental results

6.2.1. Experiment 1: lock resource exhaustion in a commercial DBMS

In a commercial DBMS [9], we measure the number of concurrent transactions causing lock resource exhaustion as we vary the number of lock resources provided by the DBMS from 1000 to 4000. When lock resource exhaustion occurs, the DBMS stops all running transactions and waits for the administrator to reset the number of lock resources.

Table 4 shows the experimental parameters used in Experiment 1. The number of files, the number of records per file, the number of files accessed by a transaction, and the number of records accessed by a transaction are equal to those in Table 3. The number of disks is 1 and the lock escalation threshold is set to 40 (80% of the average number of records in a file accessed by a transaction). The ratio of the numbers of read-only transactions and read-and-write transactions (read:write in short) is 8:2.

Table 5 shows the experimental results. In Table 5, when 16 transactions are running concurrently, lock resource exhaustion occurs and the system become paralyzed. As the number of lock resources increases, more transactions can run concurrently. However, increasing lock resources cannot guarantee there be no lock resource exhaustion as we see in Table 5.

Table 4
Experimental parameters in Experiment 1

Number of files	100
Number of records per file	100 000
Number of files accessed by a transaction	2
Number of records accessed by a transaction	Exponential distribution with the average of 100
Number of disks	1
Lock escalation threshold	40
The ratio of the numbers of read-only transactions and read-and-write transactions	8:2

Table 5
Number of concurrent transactions causing lock resource exhaustion

Number of lock resources	Number of concurrent transactions causing lock resource exhaustion
1000	16
2000	30
3000	45
4000	57

6.2.2. Experiment 2: effect of the number of concurrent transactions

We first perform experiments for typical cases where read:write is 8:2 and 2:8. We then perform experiments varying other parameters to analyze sensitivity: the file size, the number of files accessed by a transaction, the ratio of the CPU time over the disk time, and the number of disks. Since the results show similar trends, we omit the detailed results here. Interested readers are referred to Ref. [15].

Fig. 6 shows the experimental results when read:write is 8:2 and the number of lock resources is 1000. Fig. 6 (a) shows the throughput, Fig. 6 (b) the average number of aborts per committed transaction, and Fig. 6 (c) the average response time. In Fig. 6, when the number of concurrent transactions is 1, i.e., when transactions are executed serially, the throughput is 0.5. The throughput increases as the number of concurrent transactions increases at the initial stage for all lock escalation methods, but does not increase further after a certain point, and decreases in the end. This is because transactions are aborted more frequently being unable to acquire lock resources as the number of concurrent transactions

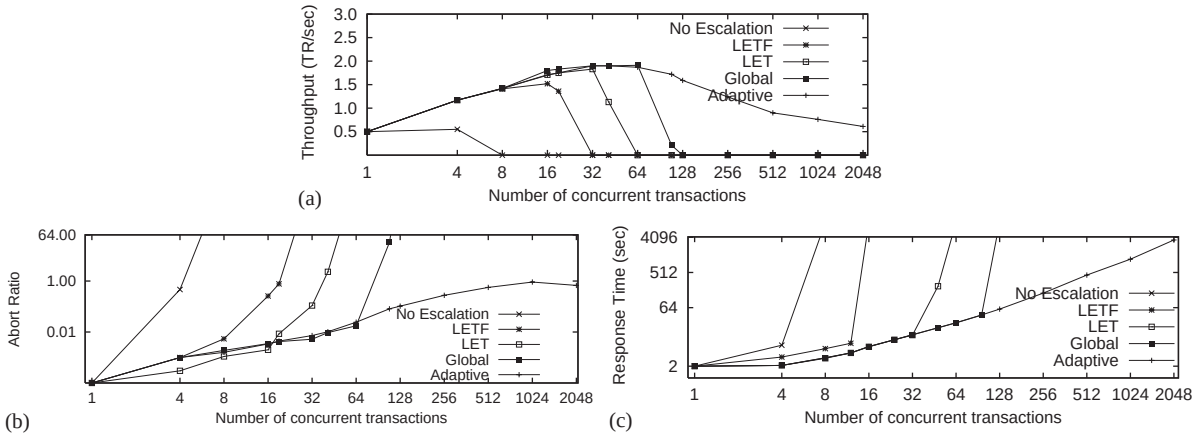

Fig. 6. Performance comparison of lock escalation methods when read:write is 8:2: (a) throughput; (b) average number of aborts per committed transaction; (c) average response time.

increases. Since the aborted transaction is restarted, the average response time increases, and the throughput decreases.

In No escalation, when eight concurrent transactions are running, the system become paralyzed, the average number of aborts per committed transaction and the average response time become infinite, and the throughput becomes 0. When four transactions are running concurrently, the abort ratio is close to 1. Four concurrent transaction may access much more than 400 records (say, more than 1000) since the number of records accessed by a transaction has an exponential distribution with the average of 100. If a transaction accesses more records than the total number of lock resources (in this case, 1000), lock resource exhaustion occurs even though only a few transactions are running concurrently.

In LETF, LET, and Global, the performances degrade abruptly when more than 32, 64, and 128 concurrent transactions are running, respectively. While there are abrupt changes in these methods, in Adaptive, the performance degrades gracefully as the number of concurrent transactions increases, and the system does not become paralyzed even when 2048 transactions are running concurrently. When there are excessively many concurrent transactions, most of them will be blocked waiting for the lock resource and only a small number of transactions will be able to progress. In the worst case, only the immortal transaction survives. This case is equivalent to a serial execution of transactions. We note that in Adaptive, the throughput with 2048 concurrent transactions (0.61) is similar to that of a serial execution (0.5).

LETF shows somewhat lower performance than LET. This is because LETF does not consider the total number of locks being actively used. LET executes lock escalation when there are no more lock resources available even if the lock escalation threshold is not exceeded. Here, we observe that global consideration on the total number of locks being used plays an important role in enhancing the performance.

Fig. 7 shows the throughputs of lock escalation methods when read:write is 2:8. The trend in Fig. 7 is similar to that in Fig. 6(c). However, the performance decreases more dramatically in the existing lock escalation methods, with the maximum number of transactions that can be run concurrently decreasing by 25%. This is because read-and-write transactions request more IX mode locks for files, making lock escalation more difficult.

6.2.3. *Experiment 3: effect of the number of lock resources*

Figs. 8(a) and (b) show the throughput and average response time for the lock escalation

methods as the number of lock resources varies. The number of concurrent transactions is 512 and read:write is 8:2. In all methods, as the number of lock resources increases, the throughput increases. Here, we make an interesting observation that the throughput and average response time for Adaptive are rather insensitive to the number of lock resources compared to those of other existing methods. Thus, the accuracy in estimating the number of lock resources is not critical for the throughput or average response time. This effect is very useful in practice since it relieves the users of the responsibility to estimate the amount of lock resources accurately at system initialization time.

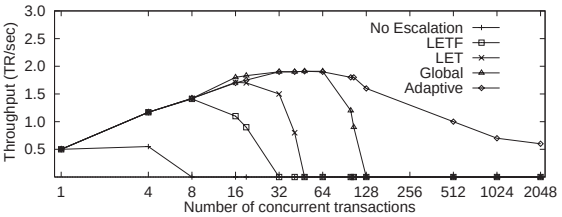

Fig. 7. Comparison of the throughputs of the lock escalation methods when read:write is 2:8.

6.2.4. Experiment 4: number of lock resources required

Fig. 9 compares the throughputs for four cases as the number of concurrent transactions varies: (1) Adaptive with 1000 lock resources (Adaptive-1000); (2) Adaptive with 10 000 lock resources (Adaptive-10000); (3) Global with 10 000 lock resources (Global-10000), and (4) Global with 100 000 lock resources (Global-100000). As we see in Fig. 9, Adaptive-10000 shows almost the same throughput as Global-100000. Adaptive-1000 also closely matches with Global-10000. This result indicates that Adaptive offers the same throughput with only less than 10% of lock resources that existing methods require, drastically reducing the requirement for lock resources. Here, we note that Global offers the best performance among existing methods.

7. Conclusions

We have addressed the problems of existing lock escalation methods: decreasing concurrency by

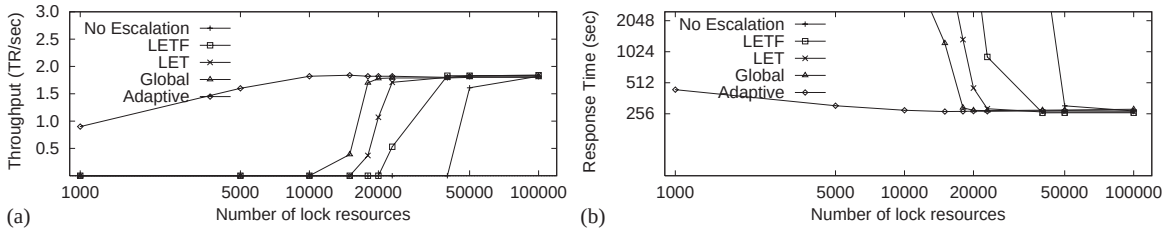

Fig. 8. Performance comparison of the lock escalation methods when 512 concurrent transactions are running.

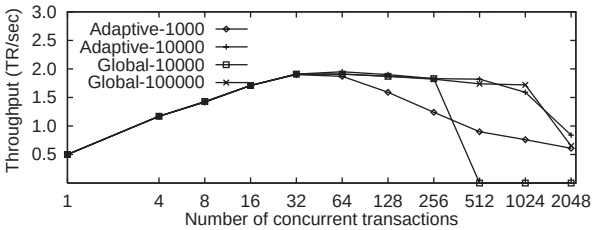

Fig. 9. Comparison of throughputs between Adaptive and Global with different amounts of lock resources.

needless lock escalation, needless transaction aborts by indolent lock escalation, and abrupt performance degradation by lock resource exhaustion. In this paper, we have proposed a general solution to these problems through systematic analysis and formal understanding of the underlying mechanisms. We summarize the contributions of the paper below.

First, we have proposed a formal model of lock escalation. To the extent of the authors' knowledge, there has been no research in the past proposing a formal model of lock escalation. Instead, lock escalation has simply been treated as a tool to have locks returned for availability. Using the model, we have analyzed the roles of lock escalation formally and have solved the problems of the existing methods systematically. In the model, we have proposed the concept of the unescalatable lock and identified it as the primary cause making the transactions to abort.

Second, we have analyzed the reasons why unescalatable locks are generated and proposed techniques to counter them: semi lock escalation, meta locking, and selective relief. Based on these techniques, we have proposed adaptive lock escalation, which controls lock escalation based on the number of unescalatable locks.

Third, through extensive simulation, we have shown that adaptive lock escalation indeed outperforms the existing lock escalation methods. The results show, compared with the existing ones, adaptive lock escalation significantly reduces the number of aborts and the average response time and, at the same time, increases the throughput. Furthermore, under excessive lock requests, adaptive lock escalation provides graceful performance degradation—gradually transiting to a serial execution of transactions—while existing methods suffer from abrupt changes in performance. Overall, the results indicate that adaptive lock escalation drastically (more than 10 fold) reduces the number of lock resources needed to maintain the same level of throughput and average response time. Moreover, the result shows that the throughput and average response time of adaptive lock escalation are rather insensitive to the number of lock resources. Existing lock escalation methods rely on the users to handle the problems of excessive lock requests obliging them to estimate the necessary amount of lock resources. In contrast, adaptive lock escalation relieves the users of this responsibility to a great extent by providing graceful performance degradation through automatic control of unescalatable locks and by providing insensitivity to the number of lock resources.

Acknowledgements

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

References

[1] K.-Y. Whang, R. Krishnamurthy, Query optimization in a memory-resident domain relational calculus database system, ACM Trans. Database Systems 15 (1) (1990) 67–95.
[2] B. Klotz, R. Bamford, Method and apparatus for dynamic lock granularity escalation and de-escalation in a computer system, United States Patent 6144983, November 7, 2000.
[3] P. Bernstein, M. Schkolnick, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.
[4] J. Gray, A. Reuter, Transaction Processing: Concepts and Technology, Morgan Kaufmann, Los Altos, CA, 1993.
[5] J. Gray, R. Lorie, G. Putzolu, Granularity of locks in a shared data base, in: Proceedings of the International Conference on Very Large Data Bases, Boston, September 1975, pp. 428–451.
[6] IBM, IBM DB2 Universal Database Administration Guide, Version 6, ftp://ftp.software.ibm.com/ps/products/db2/info/vr6/htm/db2d0/index.htm, 2000.
[7] UniSQL, Database Administration Guide (all products), 1996.
[8] J.-W. Chang, Y.-K. Lee, K.-Y. Whang, Global lock escalation in database management systems, Information Processing Letters, pp. 179–186, May 2002.
[9] Sybase, System Administration Guide (Adaptive Server Enterprise12.5), September 2002.
[10] C. Papadimitriou, The Theory of Database Concurrency Control, Computer Science Press, Rockville, MD, 1986.
[11] D.R. Ries, M.R. Stonbraker, Locking granularity revisited, ACM Trans. Database Systems 4 (2) (1979) 210–227.
[12] W. Kohler, K. Wilner, J. Stankovic, An experimental comparision of locking policies in a testbed database system, Proceedings of the International Conference on

Management of Data, ACM SIGMOD, San Jose, CA, May 1983, pp. 108–119.
[13] D.R. Ries, M.R. Stonbraker, Effects of locking granularity in a database management system, ACM Trans. Database Systems 2 (3) (1977) 233–246.
[14] W. Han, K. Whang, Y. Moon, I. Song, Prefetching based on the type-level access pattern in object-relational DBMSs, in: Proceedings of the International Conference on Data Engineering, Heidelberg, April 2001, pp. 651–660.
[15] J.-W. Chang, Y.-K. Lee, K.-Y. Whang, J.-H. Yang, A formal approach to lock escalation, Technical Report 01-11-002, Advanced Information Technology Research Center (AITrc), KAIST, 2001 (available at http://aitrc.kaist.ac.kr/research/search.html).