*In:Advances in Databases, Proc. of the British National Conf. on Databases (BNCOD15), London, UK, LNCS 1271, Springer, July 1997, pp. 54-68.*

# Detection Arcs for Deadlock Management in Nested Transactions and their Performance

**Fernando de Ferreira Rezende, Theo Härder, Andreas Gloeckner, and Jörg Lutze**

Dept. of Computer Science - University of Kaiserslautern
67653 Kaiserslautern - Germany
E-Mail: {rezende|haerder|gloeckne|lutze}@informatik.uni-kl.de

**Abstract** - In this paper, we address deadlock management in nested transactions. In our strategy, deadlocks are detected through the occurrence of cycles in a waits-for graph (WFG). However, the process of looking for cycles in the WFG considering all its nodes and edges can be very time-consuming. To accelerate this process, we propose the detection arcs. In essence, a detection arc represents a higher level abstraction embodying a hidden waiting relation between two transactions that is caused by a lock wait. Thus, the deadlock detection process needs to traverse only a minimal subset of the WFG's edges when it is started, the set of detection arcs. Therefore, the overall performance of deadlock management is improved, as confirmed by our performance measurements.

## 1 Introduction

When objects are to be accessed in a database (DB) system, transactions require locks on those in order to avoid consistency anomalies due to concurrent accesses. By requiring locks on objects, transactions may sometimes have to wait for other transactions. At this time, there may appear a cyclical sequence of transactions (**T**) each waiting for the next to release a lock it must acquire (**T1** $\rightarrow$ **T2** $\rightarrow$ ... $\rightarrow$ **T1**), and hence no one in the cycle can make any progress. These situations characterize *deadlocks*. To detect and, above all, to resolve such situations are the main tasks of a *deadlock manager*.

There are a lot of strategies to detect deadlocks. One of them is *timeout*, whereby the system, finding that a transaction is waiting too long for a lock, just guesses that there may be a deadlock involving this transaction. It then simply aborts it and restarts it again later. Although this strategy is imprecise in the detection of deadlocks, it does work. Particularly, we feel that timeout does not always offer an optimal solution to deadlocks. Although being very easy to implement, the number of transactions that may be unnecessarily aborted and restarted again may be unacceptably high due to the impreciseness of this technique.

*Waits-for graph* (WFG) [7] is another strategy, whereby the system maintains a directed graph showing which transactions are waiting for other ones. *Nodes* in this graph are labelled with transaction identifiers whereas the *edges* represent waiting situations. There is an edge from node **Ti** to node **Tj** if and only if transaction **Ti** is waiting for transaction **Tj** to release some lock. In such a strategy, deadlock detection is realized by means of searching for cycles in the WFG: When a cycle is found in this graph, it precisely means that the transactions in the cycle are deadlocked. The system then chooses one of them as a *victim*, aborts it, obliterating its effects from the DB, and restarts it again later. It is a fact that the waits-for graph strategy shows a very good precision for all kinds of transactions, independent of their duration.

Detection of deadlocks in nested transactions is more complicated and expensive than in flat transactions. The deadlock manager must be aware of the transactions nesting, since deadlocks may occur among transactions belonging to various transaction hierarchies and even among subtransactions within a single transaction hierarchy. In contrast to single-level transactions where *direct-waits-for-lock* relations are sufficient to search for waiting cycles among transactions, detection of all deadlocks in nested transactions further requires the maintenance of *waits-for-commit* relations. If deadlocks are frequently anticipated, opening-up deadlocks, which may span transaction trees, should be detected as early as possible to save transaction work [9]. For this purpose, our deadlock manager maintains further information (*indirect-waits-for-lock* relations). Finally, our strategy additionally represents in the waits-for graph *detection arcs*, which are a very efficient means to determine cycles in the graph. They represent an abstraction of other waiting relations, and their only purpose is to effectively detect the occurrences of deadlocks. Hence, they alleviate the process of searching for cycles, since it is no longer necessary to analyze all waiting relations in the graph every time a waiting situation occurs.

This paper is organized as follows. In Sect. 2, we briefly introduce the model of nested transactions and the terminology used throughout this paper. In Sect. 3, we discuss deadlock detection in nested transactions by presenting several kinds of possible deadlocks in nested transactions and by introducing the different waiting relations that are necessary to detect any of them. Thereafter, in Sect. 4 we enter the field of deadlock resolution and briefly discuss the many issues involved in choosing the lowest-cost victim for abortion. In Sect. 5, we show the most important performance measurements that we have realized, which confirm the feasibility of our strategy. Finally, in Sect. 6 we summarize the main points considered in this paper.

## 2 A Model of Nested Transactions

We basically follow Moss's terminology [13]. A transaction may contain any number of *subtransactions*, which again may be composed of any number of subtransactions – conceivably resulting in an arbitrarily deep hierarchy of nested transactions. The root transaction which is not enclosed in any transaction is called the *top-level transaction* (TL-transaction). Transactions having subtransactions are called *parents*, and the subtransactions are their *children*. We also speak of *ancestors* and *descendants*. The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation. We use the term *superior* (*inferior*) for the non-reflexive version of the ancestor (descendant). The set of descendants of a transaction together with their parent/child relationships is called the transaction's *hierarchy*. In the following, unless otherwise noted, we use the term *transaction* to denote both TL-transactions and subtransactions in general.

Notwithstanding, in contrast to Moss's nested transaction model [13], where only leaf transactions are supposed to lock objects, we assume a model of nested transactions where every transaction can acquire locks on objects. Thus, deadlocks may occur among transactions belonging to various TL-transactions and even among subtransactions within a single transaction hierarchy. Furthermore, we allow for parent/child as well as sibling parallelism in our model, that is, a parent transaction can concurrently proceed with all its inferiors (non-strict execution). However, it cannot commit before all its inferiors have committed (or aborted).

## 3 Deadlock Detection in Nested Transactions

Deadlocks in nested transactions can be managed by the concepts known for single-level transactions extended by some mechanisms tailored to the properties of the nested transactions [13, 19, 9]. To detect the occurrence of the different kinds of deadlocks, several waiting relations with different meanings must be represented in the WFG.

### 3.1 Direct-Wait Deadlocks

When a transaction requires a lock on an object which is incompatible with a lock held by another concurrent transaction, the requesting transaction is deactivated, and as a consequence a direct wait for the lock holder occurs. All direct waits are represented by *direct-waits-for-lock relations* in the WFG. Using these relations, deadlock detection can be performed immediately when a transaction is blocked or after some elapsed time. A deadlock exists if and only if a cycle is found in the direct-waits-for-lock relations. In the scenario of Fig. 1, a deadlock may be detected between transactions **E** and **H**, which *directly* wait for each other. In the following, we present this first waiting relation.

- **Direct-waits-for-lock**
  A transaction **E** (lock requestor) directly waits for another transaction **H** (lock holder) if the mode of the lock requested by **E** is in conflict with the mode of the lock held by **H**. In Fig. 1, **E** cannot proceed until **H** does, and vice versa, i.e., both are waiting for each other.

Considering the direct-waits-for-lock relations, direct-wait deadlocks can be found. However, to detect all possible deadlocks in nested transactions effectively, we have to represent more information about who is waiting for whom, i.e., more waiting relations. This is true no matter whether a deadlock occurs within a TL-transaction or among subtransactions of various TL-transactions. The next section approaches this topic.

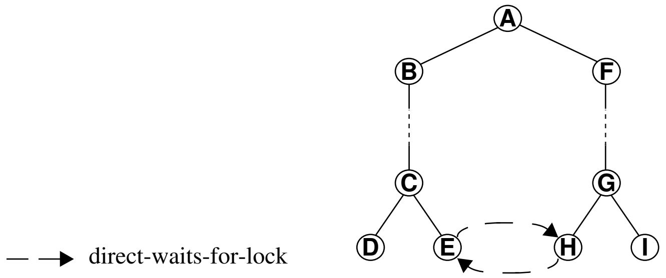


**Fig. 1.** Detection of a direct-wait deadlock.

### 3.2 Ancestor-Descendant Deadlocks

The scenario in Fig. 2 represents another kind of waiting situation. Transaction **I** directly waits for a lock held by **F**. Although progress has not stopped everywhere, because **F**, ..., **G** may proceed for some time, they cannot commit without aborting **I**, and therefore the best decision is to detect and resolve this ancestor-descendant deadlock immediately. A request of **I** causing a lock wait on its superior **F** can be detected by using the already introduced direct-waits-for-lock relation combined with another kind of waiting relation, which takes into account the hierarchical relationships of transactions – the *waits-for-commit relation*. This is explained in the following.

• **Waits-for-commit**
Since a waiting lock requestor **I** cannot proceed with its work, all its superiors must wait as well for commit processing. In Fig. 2, **G** cannot commit until **I** does, **G**'s parent cannot commit until **G** does, and so forth. In fact, all superiors of **I** cannot commit before **I** does. This kind of waiting relation is denoted *waits-for-commit*, and it can be represented by the parent-child relationships among the transactions in a hierarchy.

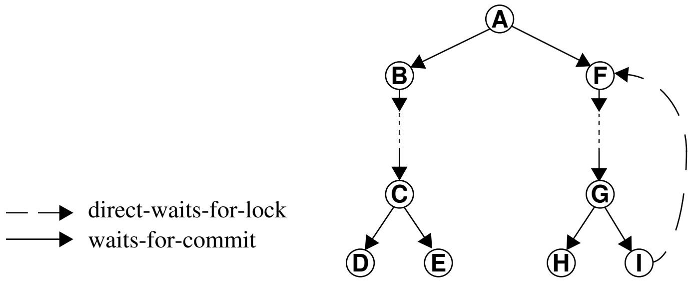


**Fig. 2.** Detection of an ancestor-descendant deadlock.

The combined use of direct-waits-for-lock and waits-for-commit relations turns out to be sufficient to detect existing cyclical waiting situations in nested transactions embodying both direct-wait as well as ancestor-descendant deadlocks. Hence, both relations are sufficient for detecting deadlocks in nested transactions [13]. However, to make this process more effective, Härder and Rothermel [9] have proposed the representation of an extra, refined waiting relation in order to detect opening-up deadlocks.

### 3.3 Opening-Up Deadlocks

Since a direct-waits-for-lock relation is only represented between the lock requestor and the lock holder (or, after commit of the lock holder, the current retainer), waiting situations between the lock requestor and all ancestors of the lock holder (retainer) are not explicitly established in the waits-for information thus far. In the context of nested transactions, however, these *indirect* waiting relationships should be taken into account to make early deadlock detection possible. If we examine only both kinds of waiting relations presented thus far (direct-waits-for-lock and waits-for-commit), the scenario of Fig. 3 represents a deadlock-free situation, since there is no cycle in the WFG involving only those relations. However, there is an emerging deadlock, because **I** indirectly waits for the oldest ancestor of **M** that is not an ancestor of **I**, in this case **J**. On the other hand, **Q** indirectly waits for the oldest ancestor of **D** that is not an ancestor of **Q**, i.e., **A**. If we evaluate this information (**I** $\rightarrow$ **J**, **Q** $\rightarrow$ **A**), we are able to immediately detect a cycle to be opening up. Hence, the representation of such indirect waiting situations as waiting relations in the WFG allows for the detection of *future* deadlocks. This waiting relation is introduced in the following.

• **Indirect-waits-for-lock**
A lock requestor **Q** directly waits for a lock holder **D** if the mode of the lock requested by **Q** is in conflict with the mode of the lock held by **D** (this is the direct-waits-for-lock relation already presented). Further, let **A** be the highest ancestor of **D** that is not an ancestor of **Q**. Then, **Q** indirectly waits for **A** (see Fig. 3). This waiting situation is represented as an *indirect-waits-for-lock relation*.

An optimistic attitude would not care about such an opening-up deadlock, since an abort of any transaction involved would eventually avoid the actual occurrence of the deadlock before all progress ceases. However, transaction aborts are regarded as exceptions and should not be taken into account as a remedy to break opening-up deadlock cycles. In contrast, a pessimistic approach usually saves work [9]. However, the additional representation and management of indirect-waits-for-lock relations imposes some more overhead.

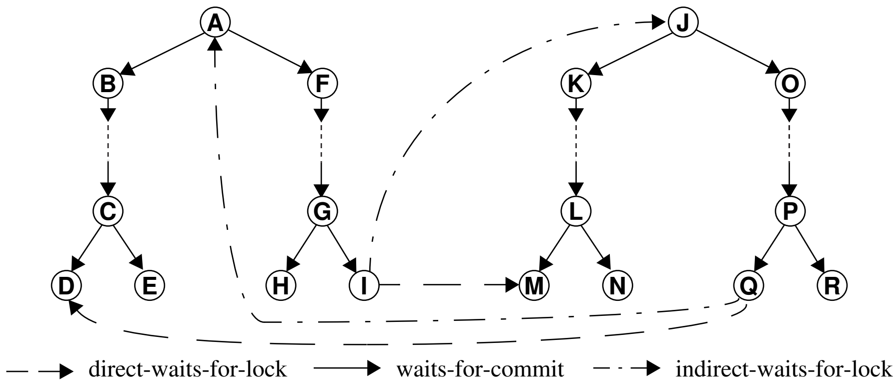


**Fig. 3.** Detection of an opening-up deadlock.

This *indirect-waits-for-lock* relation is called *waits-for-retained-lock* by Härder and Rothermel [9]. We have renamed this relation here because when such a waiting situation occurs between subtransactions of different TL-transactions, as shown in Fig. 3, a subtransaction is not actually waiting for a *retained* lock (as the name suggests), but it in fact *indirectly* waits for a lock of the other TL-transaction hierarchy.

In addition, [9] proposes the representation of such indirect waiting relations for *all* ancestors of the lock holder up to the first non-common ancestor of the lock requestor. In the example of Fig. 3, not only the edges **I** → **J** and **Q** → **A** would be represented, but additionally **I** → **L**, ..., **I** → **K** as well as **Q** → **C**, ..., **Q** → **B**. We advocate that representing all these indirect-waits-for-lock relations is, as well as superfluous, very expensive for the deadlock manager. Therefore, for efficiency reasons, in our scheme we represent just one of them, namely the one involving the highest non-common ancestor between the two transactions, as shown in Fig. 3.

### 3.4 Overview of the Deadlock Detection Algorithm

Based on the previous discussions about the many waiting situations, we present in the following an overview of the algorithm for handling deadlocks. The algorithm is divided into four main bodies in order to capture all the inherent dynamism involved in the lifetime of transactions.

**A.** *Whenever a transaction starts to wait for a lock:*

- **A. i.** Determine all transactions as well as the waiting relations (namely direct-waits-for-lock, waits-for-commit, and indirect-waits-for-lock) involved in this lock wait.
- **A. ii.** Insert all transactions as nodes in the WFG, if they are not already there.
- **A. iii.** Insert all waiting relations as directed edges between the corresponding nodes in the WFG.

**A. iv.** Initiate detection by means of navigating through all the nodes and edges in the WFG looking for all possible cycles.

**A. v.** If cycles are found, then choose, among the transactions involved in the cycles, victims for abortion (refer to Sect. 4).

**B.** *Whenever a TL-transaction commits:*

**B. i.** Check whether there is a node in the WFG for the committing TL-transaction.

**B. ii.** If there is such a node, then remove it and all its incoming edges from the WFG.

**C.** *Whenever a subtransaction commits:*

**C. i.** Check whether there is a node in the WFG for the committing subtransaction.

**C. ii.** If there is such a node, then inherit all its incoming edges to the corresponding node of the committing subtransaction's parent and finally remove the node from the WFG.

**D.** *Whenever a transaction aborts:*[1]

**D. i.** Check whether there is a node in the WFG for the aborting transaction.

**D. ii.** If there is such a node, then remove it and all its incoming as well as outgoing edges from the WFG.

This could be a standard algorithm for deadlock detection in nested transactions using the waiting relations we have introduced so far. Surely, the most time-consuming aspect of this algorithm is the navigation process of looking for cycles in the WFG which involves traversing **all** its nodes and edges (step **A.iv.**). In the following, we present the detection arcs which substantially decrease such processing time.

### 3.5 The Detection Arcs

The main goal we have in mind here is to optimize the time-consuming process of looking for cycles in the WFG. The basic idea to achieve this goal is based on an extension of the indirect-waits-for-lock relations. We have seen previously that a transaction waiting for a lock indirectly waits for the oldest ancestor of the lock holder that is not itself an ancestor of the lock requestor. Using Fig. 4, transaction **I** directly waits for **M** and thus indirectly for **J**. However, if we analyze this waiting situation in more details, we can realize that not only **I** indirectly waits for **J** but also **A**, since **A** cannot commit before **I** does. Hence, the highest non-common ancestor of the lock requestor waits for the highest non-common ancestor of the lock holder. The representation of this waiting relation in the WFG – which we have called *detection arcs* – greatly alleviates the process of looking for cycles. In the following, we introduce detection arcs.

- **Detection arcs**
  A lock requestor **I** directly waits for a lock holder **M** if the mode of the lock requested by **I** is in conflict with the mode of the lock held by **M** (direct-waits-for-lock relation). Let **A** be the highest ancestor of **I** that is not an ancestor of **M**, and similarly, let **J** be the highest ancestor of **M** that is not an ancestor of **I**. Then, **A** indirectly waits for **J** through the direct wait between **I** and **M**. This waiting situation is represented as a *detection arc* in the WFG (Fig. 4).

Essentially, a detection arc represents a higher level abstraction of the direct-waits-

1. We have not considered here partial rollbacks of transactions [11, 15]. However, a routine to cope with that feature can be thought of without serious difficulties.

for-lock and indirect-waits-for-lock relations between two transactions that have been caused by a lock wait. The most important advantage of the explicit detection arc representation is the process speed-up of looking for cycles in the WFG. We have previously seen the three kinds of deadlocks that can happen in nested transactions: direct-wait, ancestor-descendant, and opening-up deadlocks. With the help of the detection arcs, all direct-wait and opening-up deadlocks can be detected solely on their basis. Thus, in these cases the process of looking for cycles in the whole WFG can be reduced to looking for cycles only in a subset of the WFG, namely, the set of detection arcs. In Fig. 4, the cycle **A** → **F** → ... **G** → **I** → **J** → **O** → ... **P** → **Q** → **A** was reduced to the cycle **A** → **J** → **A** by means of detection arcs. Therefore, the detection of these kinds of deadlocks can be made much more efficient.

On the other hand, ancestor-descendant deadlocks cannot be detected with the help of the detection arcs. However, the detection of this kind of deadlock is in fact a very simple matter: whenever a lock wait happens, one must simply check whether the lock requestor is a descendant of the lock holder. Particularly in our implementation, this process was simplified even more, since we catch this information from the identifier of the requesting transaction which carries encoded information of all its ancestor transaction identifiers (refer to [18]). Therefore, whenever a transaction starts to wait for a lock, we check the inferior relationship between the requestor and the holder before the representation and derivation of the information for the WFG takes place. If they are in the same path of a transaction hierarchy, then there is a deadlock. If not, the algorithm for deadlock detection can proceed normally.

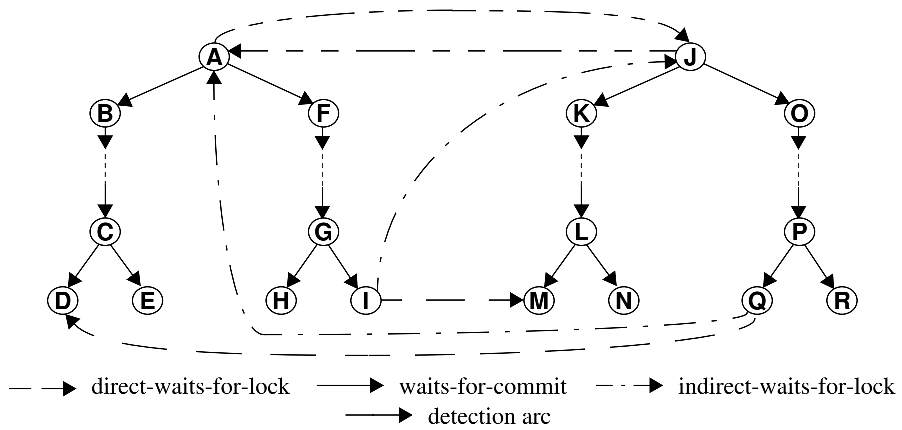


**Fig. 4.** Deadlock detection by means of detection arcs.

Furthermore, it is important to notice that finding a cycle involving the detection arcs only means, of course, that a deadlock has happened. All the transactions de facto involved in such a cyclical waiting situation are not determined. Hence, after having detected the deadlock occurrence, the usual routine for finding cycles in the whole WFG must be initiated. However, since the occurrence of a deadlock is generally considered to be a rare event [5], the time-consuming process of looking for cycles in the whole WFG is rarely started, namely, only when it is quite sure that a deadlock has occurred, and not every time a lock wait happens. Consequently, the overall performance of deadlock management is improved.

### 3.6 Deadlock Detection Algorithm Using Detection Arcs

In the following, we present an overview of the algorithm for handling deadlocks by means of detection arcs. In particular, only the first part (**A.**) of the previous algorithm needs to be adjusted, the other ones (**B.**, **C.**, and **D.**) remain unchanged (refer to Sect. 3.4). Furthermore, we have introduced in this algorithm a *counter* of the number of detection arcs between two nodes. Since the detection arcs represent a higher level abstraction of other waiting relations, there may appear several detection arcs between the very same nodes which in fact have the same meaning and purpose. The main idea behind such a counter is to restrict the number of detection arcs between two nodes. Having outlined the algorithm in the following, we discuss the main advantages of employing such counters.

**A.** *Whenever a transaction starts to wait for a lock:*

- **A. i.** Check the inferior relationship between the lock requestor and the lock holder. If they are in the same path of a transaction hierarchy, then choose the inferior transaction as a victim for abortion (refer to Sect. 4) and conclude the process.
- **A. ii.** Determine all transactions as well as the waiting relations (namely direct-waits-for-lock, waits-for-commit, indirect-waits-for-lock, and detection arcs) involved in this lock wait.
- **A. iii.** Insert all transactions as nodes in the WFG, if they are not already there.
- **A. iv.** Insert the direct-waits-for-lock, indirect-waits-for-lock, and waits-for-commit (this one only if it is not already there) relations as directed edges between the corresponding nodes in the WFG.
- **A. v.** Insert the detection arc as a directed edge between the corresponding nodes in the WFG. If there is already one such edge between both nodes, then only increment a corresponding counter.
- **A. vi.** Initiate detection if and only if a new detection arc was inserted (i.e., if an existing counter was not incremented), by means of navigating only through the detection arcs of the WFG looking for possible cycles.
- **A. vii.** If cycles are found, navigate through all the nodes and edges in the WFG looking for all possible cycles and then choose, among the transactions involved in the cycles, victims for abortion (refer to Sect. 4).

On the one hand, incrementing a counter whenever a detection arc is already available between the nodes (step **A.v.**) is necessary to cope with the deletion of the direct-waits-for-lock relations from the WFG. At that time, it can be efficiently decided by means of this counter whether the corresponding edge must be deleted as well, or whether there still are other direct-waits-for-lock relations below in the hierarchy subsumed under this edge.

On the other hand, another important aspect of this counter is that detection can be initialized only when a new detection arc is inserted in the WFG (step **A.vi.**). Of course, if the WFG is acyclic, after the insertion of an edge (detection arc) with the very same meaning between the very same nodes, it still remains acyclic. Therefore, this counter avoids the insertion of superfluous edges in the WFG. Furthermore, it allows for performance improvement, since the WFG becomes smaller and detection is initiated only when there is de facto the possibility of cycles being closed.

## 4 Deadlock Resolution

Since a deadlock does not go away by itself, after having been detected it must then be resolved. Deadlock resolution is typically done by means of the transaction (partial) rollback mechanism already available for fault tolerance: a deadlock victim must be found and rolled back. However, choosing the right victim for abortion is not a simple matter. In Fig. 5, suppose the last waiting situation signaled by the system is that transaction **F** waits for **B**. Until that point, the WFG was acyclic. However, after inserting the edge **F** $\rightarrow$ **B** (in this example, it does not matter which particular kind of waiting relation this edge represents), several cycles appear at once. Hence, finding the lowest-cost victim for abortion, the exact one which affects the minimal data granules of work lost and at the same time breaks all the cycles in the WFG, may be a time-consuming task.

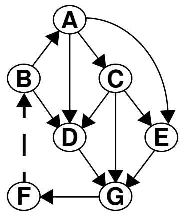


**Fig. 5.** Appearance of several cycles in a WFG at once.

Particularly in the case that the deadlock is of the kind ancestor-descendant, it is simple to find the victim: Since rollback of the ancestor transaction implies rollback of all its inferiors (committed and active), rollback of the descendant transaction is always cheaper than that of the corresponding ancestor, of course. In such a case, the descendant must be chosen as a victim and rolled back.

However, if the deadlock is either of the kind direct-wait or opening-up, then all transactions involved have to be considered to determine the lowest-cost victim for rollback and to break all the cycles appeared. There are several aspects that can influence the selection of the victim, as for example:

- the transaction level in the hierarchy and consequently the number of inferior transactions;
- the number of log records written thus far;
- the total execution time;
- objects changed so far;
- objects touched so far;
- processor time used so far; etc.

In contrast to Moss's nested transaction model [13], in other more flexible models – such as Camelot [20, 4], Clouds [2, 3], Eden [1, 14], LOCUS [12, 21], PRIMA [6, 8], KRISYS [17, 16], etc. – the transactions involved in the deadlock can occur everywhere in the transaction trees. Consequently, the derivation of the backout cost is much more complex; it must include the cost of aborting not only the victim transaction, but also of the corresponding transaction tree of committed and active subtransactions, and it must consider their work lost. Finally, when choosing the victim, the deadlock manager must be concerned about starvation, i.e., it must permit all transactions in the system to proceed eventually, in order to guarantee the progress of the system as a whole.

Particularly in the implementation of our multi-user knowledge base management system [16, 10], we have simplified the deadlock resolution process, since we believe that the costs for finding the ideal victim do not pay off compared to the remote possibility of eventually aborting the wrong transaction. Thus, when a deadlock of the kind ancestor-descendant happens, the deadlock manager requests the abortion of the descendant transaction, of course. Further, if the deadlock is either of the kind direct-wait or opening-up, then the deadlock manager considers just the transactions involved in the last waiting situation for aborting purposes. (Notice, therefore, that we do not need to start the process to look for cycles in the *whole* WFG after having detected the deadlock occurrence through the detection arcs, i.e., in our system the step **A.vii.** of the previous algorithm is saved.) On the basis of their transaction identifiers [18], the deadlock manager derives the information about the levels that those transactions are on and simply aborts the leaf-most one. Finally, if they are on the same level, the deadlock manager aborts the requesting transaction.

# 5 Performance Evaluation

## 5.1 The Conventional Strategy

The strategy we have chosen for comparison is exactly the one proposed by Härder and Rothermel [9] – here referred to as *conventional strategy*, to be compared with the *detection arcs strategy*. As previously presented, the waiting relations that are represented in the WFG by this strategy are: direct-waits-for-lock, waits-for-commit, and indirect-waits-for-lock (or *waits-for-retained-lock* according to their terminology).

## 5.2 The Algorithms and the Environment

The algorithms we have implemented, in C, may be gotten via *anonymous ftp* under *ftp.uni-kl.de* (131.246.94.94, /pub/informatik/software/rezende/Deadlock_NT). We have run the algorithms on a Sun Sparc Station 20[2], with 96 Mbytes of main memory, under Solaris 2.4[2], windows system Sun-X11R6[2]. The time measurements have been made through the High-Resolution-Virtual-Timer of the Sun Workstation. We have performed the algorithms in hypothetical transaction hierarchies by varying depth as well as breadth and the number of direct-waits-for-lock relations, as will become clear in the next sections. Furthermore, we have repeated all functions ten thousand times in order to get good averages that are not so disturbed by occasional machine overloads. We have basically measured the detection process and the maintenance overhead of the WFG, both with respect to processing time.

## 5.3 Detection Process

In both strategies, the detection process (search for cycles in the WFG) is mainly influenced by the following factors:

(1) the level difference (*relative depth*) between the waiting transaction and the highest non-common ancestor of the awaited transaction;
(2) the *number of direct-waits-for-lock relations*; and
(3) the general *level of ramification* of the WFG, i.e., breadth as well as depth of the transaction hierarchies represented in the WFG.

Of course, there are numerous possibilities of varying as well as combining these factors in order to analyze the performance of both strategies. Particularly in our investigation, we have considered four variations of these factors. We have kept the ramifi-

2. Registered trademark of Sun Microsystems, Inc.

cation level of the transaction hierarchies to the minimal value (i.e., only one path in each hierarchy) and varied then both the relative depth between waiting transaction and the highest non-common ancestor (Fig. 6) as well as the number of direct-waits-for-lock relations (Fig. 7). On the other hand, we have considered ramifications in the transaction hierarchies (two paths) and varied both factors as before, i.e., the relative depth (Fig. 8) and the number of direct-waits-for-lock relations (Fig. 9). As can be observed from the measurements, however, the behavior of both strategies relative to each other in non-ramified and ramified WFGs is very similar, just the time figures are somewhat different. In addition, it is important to notice here that the WFG in all its variations was prepared before the detection process was started, since we intended here to measure the processing time of the detection process when facing different situations. Therefore, the potential advantages of the counters of detection arcs discussed in Sect. 3.6 were not exploited in these measurements. In particular, we have considered that the left- and leaf-most transaction in the hierarchies has caused the last waiting situation; thus the detection process is started on it (shadowed circle in the forthcoming figures). In addition, the detection process has considered all necessary nodes and edges of the WFG – in fact, we have not closed any cycle in the WFG so that all nodes and edges must be transitively traversed. The maintenance overhead of the WFG is considered in the next section.

Fig. 6 illustrates the cost in time of the detection process of both strategies in non-ramified WFGs with varying relative depths between waiting transaction and the highest non-common ancestor. The processing time for the detection arcs strategy remains constant independent of the relative depth because the number of detection arcs in the WFG does not change when the depth is varied. On the other hand, the conventional strategy shows rising processing time as the depth is varied, since this strategy must traverse an increasing number of waiting relations in the WFG. In case of detection arcs, checking can be done directly on the representation, whereas the conventional strategy requests the iterative checking of all conceivable waiting relations.

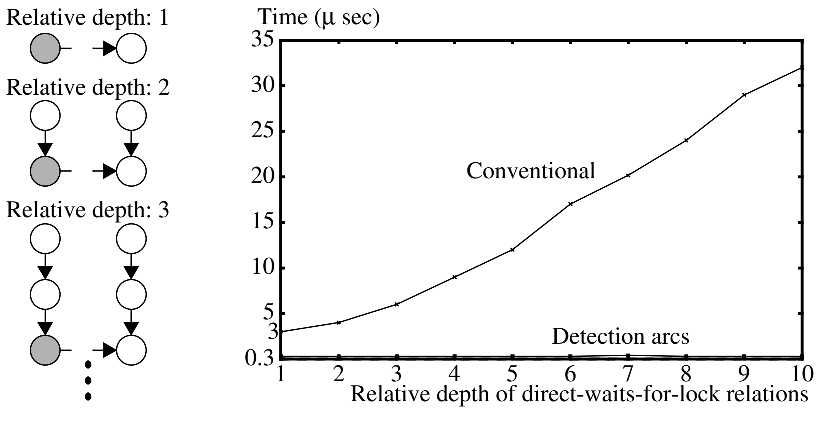


**Fig. 6.** Detection process in non-ramified WFGs – varying relative depths.

In Fig. 7, the performance of both strategies is shown when the number of direct-waits-for-lock relations is varied in non-ramified WFGs. As before, the conventional scheme similarly shows increasing time figures, since it must traverse an increasing number of waiting relations. In turn, the detection arcs strategy shows a slight increase in processing time because the number of detection arcs in the WFG has increased.

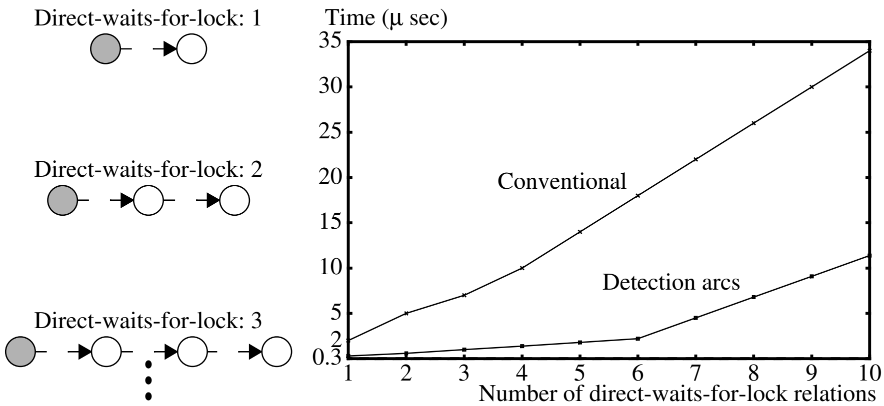


**Fig. 7.** Detection process in non-ramified WFGs – varying direct-waits-for-lock relations.

Considering now ramified WFGs, Fig. 8 illustrates the performance of both strategies when the relative depth is varied. In this case, the detection arcs strategy shows constant processing time because, as before, the number of detection arcs is not influenced by depth variations. On the other hand, the conventional strategy shows much worse processing times due to the high number of waiting relations caused by the ramifications in the WFG.

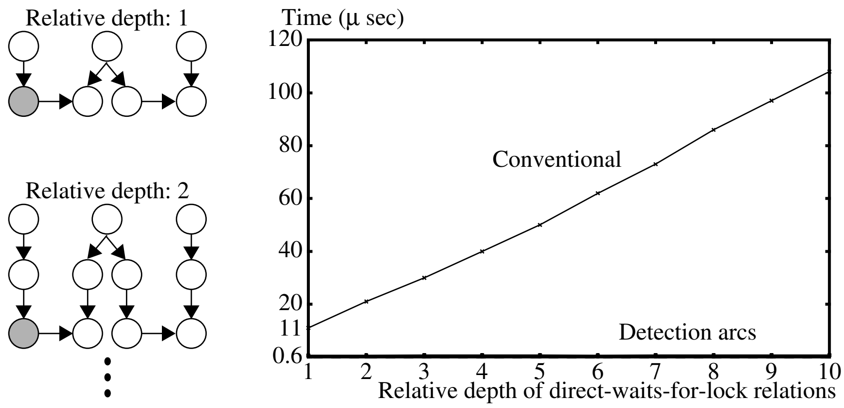


**Fig. 8.** Detection process in ramified WFGs – varying relative depths.

Finally, Fig. 9 illustrates the behavior of both strategies in ramified WFGs when the number of direct-waits-for-lock relations is varied. Similarly, the conventional strategy shows very high and constantly increasing processing times due to the ramifications in the WFG. On the other hand, the detection arcs strategy shows low but now rising processing times because again the number of detection arcs in the WFG has increased.

All in all, the detection arcs strategy achieves processing times for the detection of deadlocks drastically better than the conventional strategy. The general reason for that is what we have been affirming since the beginning of this paper: The number of edges of the WFG to be traversed by the detection process is substantially reduced through the representation and exploitation of the detection arcs.

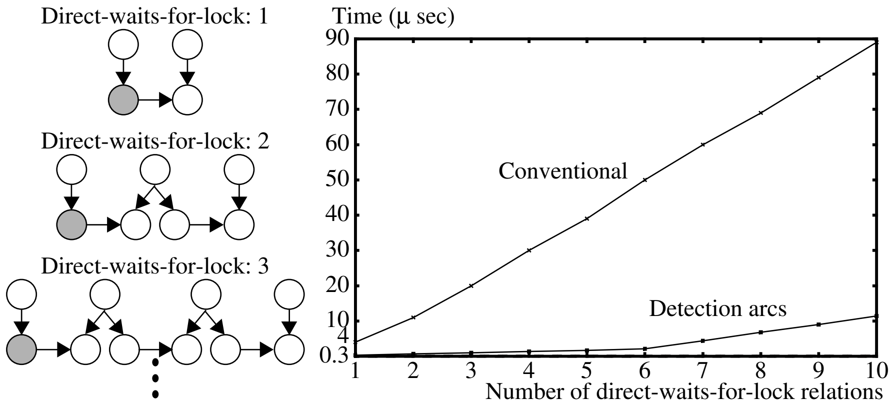


**Fig. 9.** Detection process in ramified WFGs – varying direct-waits-for-lock relations.

### 5.4 Maintenance Overhead of the Waits-For Graph

In order to analyze the maintenance overhead of the WFG in both strategies, we have considered insertion and deletion of nodes in the WFG and, as a consequence, the insertion and deletion of the corresponding waiting relations. Here, we have subsumed these operations under update operations in general. Fig. 10 illustrates the behavior of both strategies during the maintenance of the WFG. In particular, we have considered here a non-ramified WFG with varying relative depths; such variations, however, do not affect the behavior of both strategies relative to each other.

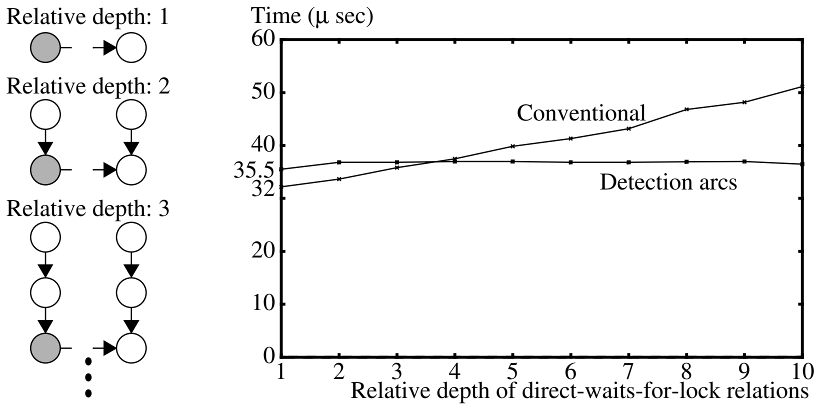


**Fig. 10.** Maintenance overhead for update operations.

The main difference in the performance of both strategies is due to the update of the varying number of waiting relations. In the detection arcs strategy, the maintenance overhead is almost constant independent of the depth of the transaction hierarchies. On the other hand, the conventional strategy shows a slightly-increasing overhead because the number of waiting relations (indirect-waits-for-lock) also increases when the transaction hierarchies become deeper. Finally, the detection arcs strategy has a maintenance overhead greater than the conventional one in the beginning due to the higher costs to determine (twice) the highest non-common ancestor. However, this cost is paid off when the hierarchies are deeper because the conventional strategy must then cope with more waiting relations than the detection arcs strategy.

## 6 Conclusions

In order to correctly handle deadlocks in nested transactions, the concepts known for flat transactions must be extended with special waiting relations to reflect the properties of the nested structure of transactions. The usual direct-waits-for-lock relation reflects the idea that a lock requestor waits for a lock holder due to a conflict between requested and held lock modes. Such a waiting relation enables the detection of direct-wait deadlocks, where only lock requestor and holder are involved.

To detect deadlocks between ancestor and descendant, another kind of waiting relation, called waits-for-commit, must be represented in the WFG. This one reflects the hierarchical relationships between the transactions and represents the fact that an ancestor waits for the commitment of its inferiors. By such a means, a deadlock can be detected as soon as a descendant starts waiting for a lock held by any of its superiors.

If deadlocks are frequently anticipated, opening-up deadlocks should be detected as early as possible to save transaction work. For this purpose, we have additionally used the indirect-waits-for-lock relation. This one reflects the idea that the lock requestor waits for the highest ancestor of the lock holder which is not its ancestor. The representation of such a waiting relation allows for the detection of future deadlock cycles, thereby saving transaction work.

Furthermore, since the process of looking for cycles in the WFG considering all its nodes and edges may be extremely expensive, we have introduced a special kind of waiting relation which represents a high-level abstraction of the direct-waits-for-lock and indirect-waits-for-lock relations between two transactions, the detection arcs. With their help, the deadlock detection process can be made very efficient, because only a minimal subset of the WFG's edges needs to be navigated. Furthermore, such a detection process must be started only when a new detection arc is inserted in the graph, and not necessarily every time a lock wait happens.

We have also considered the principles of deadlock resolution in nested transactions. Essentially, deadlock resolution is based on transaction rollbacks. The transactions involved in the cyclical waiting situation have to be considered, and a victim for abortion must be chosen. The process of choosing such a victim is not trivial, because ideally the victim should be a transaction affecting the minimal data granules of lost work and at the same time breaking all the cyclical waiting situations. Particularly in our implementation, we have simplified this process by using some heuristics.

At last, we have realized many performance measurements, the most important of them we have shown here. With respect to all situations faced by the deadlock detection process, our detection arcs strategy has achieved processing times much better than the conventional one. In turn, the maintenance overhead of the WFG in both strategies has revealed no significant differences. Finally, with the algorithms we have made available via anonymous ftp, we hope to facilitate the work of the ones who might like to implement our ideas in their own systems.

## References


1. Almes, G.T., Black, A.P., Lazowska, E.D., Noe, J.D.: The Eden System: A Technical Review. IEEE Trans. on Software Engineering SE-11 (1985) 43-58
2. Ahamad, M., Dasgupta, P., Blanc, R.J., Wilkes, C.T.: Fault Tolerant Computing in Object-Based Distributed Systems. Proc. IEEE Symp. on Reliability in Distributed Software and Database Systems (1987)
3. Dasgupta, P., Blanc, R.J., Appelbe, W.: The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work. Proc. 8th IEEE Int. Conf. on Distributed Computing Systems, San Jose, USA (1989)
4. Eppinger, J.L., Mummert, L.B., Spector, A.Z. (Eds.): Camelot and Avalon: A Distributed Transaction Facility. Morgan Kaufmann Publ., San Mateo, USA (1991)
5. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publ., San Mateo, USA (1993)
6. Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications. Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, U.K. (1987) 433-442
7. Holt, R.C.: Some Deadlock Properties in Computer Systems. ACM Computing Surveys 4: 3 (1972) 179-196
8. Härder, T., Profit, M., Schöning, H.: Supporting Parallelism in Engineering Databases by Nested Transactions. SFB 124 Res. Report 34/92, Univ. of Kaiserslautern, Germany (1992)
9. Härder, T., Rothermel, K.: Concurrency Control Issues in Nested Transactions. VLDB Journal 2:1 (1993) 39-74
10. Mattos, N.M.: An Approach to Knowledge Base Management. LNAI 513, Springer-Verlag, Berlin, Germany (1991)
11. Mohan, C., Haderle, D., Lindsay, B.G., Pirahesh, H., Schwarz, P.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. on Database Systems 17:1 (1992) 94-162
12. Müller, E.T., Moore, J.D., Popek, G.A.: Nested Transaction Mechanism for LOCUS. Proc. 9th Symp. on Operating Systems Principles (1983) 71-89
13. Moss J.E.B.: Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, USA (1985)
14. Pu, C., Noe, J.D.: Nested Transactions for General Objects: The Eden Implementation. TR-85-12-03, Univ. of Washington, Washington D.C., USA (1985)
15. Rezende, F.F., Baier, T.: Employing Object-Based LSNs in a Recovery Strategy. Proc. 7th Int. Conf. on Database and Expert Systems Applications, Zurich, Switzerland (1996) 116-129
16. Rezende, F.F.: Transaction Services for Knowledge Base Management Systems - Modeling Aspects, Architectural Issues, and Realization Techniques. Doctor Thesis, Univ. of Kaiserslautern, Germany (1997)
17. Rezende, F.F., Härder, T.: Concurrency Control in Nested Transactions with Enhanced Lock Modes for KBMSs. Proc. 6th Int. Conf. on Database and Expert Systems Applications, London, U.K. (1995) 604-613
18. Rezende, F.F., Härder, T., Zielinski, J.: Implementing Identifiers for Nested Transactions. Proc. 16th Brazilian Computer Society Conference - 23rd Integrated Seminar on Software and Hardware (SBC/SEMISH'96), Recife, Brazil (1996) 119-130
19. Rukoz, M.: Hierarchical Deadlock Detection for Nested Transactions. Distributed Computing 4 (1991) 123-129
20. Spector, A.Z., Pausch, R.F., Bruell, G.: Camelot: A Flexible, Distributed Transaction Processing System. Proc. IEEE Spring Computer Conference, USA (1988)
21. Weinstein, M., Page, T., Livezey, B., Popek, G.: Transactions and Synchronization in a Distributed Operating System. Proc. 10th Symp. on Operating Systems Principles (1985) 115-126