공학 연구논문집 제22권 제2호 pp. 87~105, 1991, 울산대학교
Journal of Engineering Research Vol. 22, No. 2, pp. 87~105, 1991.
University of Ulsan.

여러 로크 모드를 가진 순차 트랜잭션 처리를 위한 교착상태의 연속적 검출 및 회복 기법 *

박 영 철
전자계산학과

〈요 약〉

이 논문은 엄격한 두 단계 로킹이 직열성 보장을 위하여 적용되고 다단계 로킹규약이 다섯 가지 로크 모드(IS, IX, S, SIX 그리고 X)를 적용하는 환경에서의 순차 트랜잭션 처리에 있어서의 교착상태 문제를 다루는 알고리즘을 제시한다. 각 자원에 대하여 우리의 스케쥴링 정책은 로크의 변경을 제외하고는 먼저온 순서대로 로크 요구를 취급한다. 교착상태의 검출과 회복을 위한 기본 착상은 H/W-TWBG라는 새로운 지향그래프의 형성에 있다. 이 그래프는 여러가지 바람직한 특성을 가지고 있어서 대기그래프를 대신하여 사용할 수 있다. 교착상태 원에서 희생자들의 판별원리를 설정하고, 그리고 타당한 시간 및 면적 복잡성을 가지고 그 해결책이 거의 최적인 검출 및 회복 알고리즘을 제시한다. 우리의 교착상태 검출기법의 한 중요한 특징은 어떤 교착상태들은 어느 트랜잭션의 포기없이도 해결될 수 있다.

A Continuous Deadlock Detection and Resolution Algorithm for Sequential Transaction Processing with Multiple Lock Modes

Park, Young-Chul
Department of Computer Science

〈Abstract〉

An algorithm for deadlock detection and resolution in the sequential

* This research is supported in part by a grant from Asan Foundation

transaction processing is presented, where two-phase locking is assumed for ensuring serializability, the lock requests obey the granularity locking protocol and each granule may be locked in one of the following lock modes: IS, IX, S, SIX and X. For each object, lock requests are honored according to a first-come-first-served basis except for lock conversions. The basic idea for the deadlock detection and resolution is in the construction of a new directed graph called a Holder/Waiter-Transaction Waited-By Graph. We establish guidelines for the identification of a victim in a deadlock cycle and propose a resolution algorithm whose time and space requirements are resonable and its solution is near optimal. In addition, our algorithm allows us to resolve some deadlocks without aborting any transaction.

1. Introduction

In conventional transaction mode, a transaction is defined as an atomic execution of a program that manipulates the database by means of read and write operations, thereby transforms the database from one consistent state to another consistent state. In this environment, each transaction requests at most one lock at a time and when a request cannot be granted immediately, the transaction is blocked until either the request is granted or the transaction itself is aborted by some reason(deadlock or site failure). Deadlocks are usually characterized in terms of *a transaction wait-for graph*(TWFG). The TWFG is a directed graph where each vertex represents a transaction;each edge of the form $T_i \rightarrow T_j$ in TWFG means that transaction Ti is waiting for the completion of T_j. It has been shown that there exists a deadlock if and only if there is a cycle in TWFG[9].

Agrawal et al.[1] presented a linear time and space algorithm with respect to the number of transactions considering only exclusive lock modes (*X locks*). They extended the algorithm by allowing shared lock modes(*S locks*) as well as X locks. However, to maintain the same time and space complexity, even when a writer T_i is blocked by multiple readers, only one of the readers, say T_j, is selected to represent a wait-for relationship $T_i \rightarrow T_j$ in TWFG. Because of this, detection of some deadlocks can be delayed and some transaction may hold resources or wait for other transactions unnecessarily even they might be aborted later. in [2] and [3], they permit each transaction to have multiple wait-for edges but they did not show the data structure for the TWFG construction, explain how to detect deadlock cycles and how minimal cost victim can be selected for a found cycle.

Jiang [11] proposed an algorithm to

cure those problems in Agrawal et al. [1] while supporting the same lock modes. The TWFG, in his algorithm, is represented by a (n+1)*n-matrix and $O(e)$ time is required for finding a cycle and listing all participators in the cycle, where n is the number of transactions and e is the number of wait-for relationships in the graph. However, in general, when a deadlock is involved in multiple cycles and each cycle has a different cycle length, to list all participators in each cycle, in the worst case, he showed that it works in an exponential function of the number of transactions: $O(3^{n/3})$.

Elmagarmid[6] developed an algorithm for multiple outstanding lock requests with the same lock modes as the above two algorithms. Because a special case of the multiple outstanding lock requests can be the conventional transaction mode, through the slight modification, his algorithm can be compared to the others. Instead of TWFG, in his algorithm, two tables are maintained: T-table and R-Table. The T-table keeps all the blocked transactions with their requesting resources and requesting lock modes. R-table keeps all the resources which are held by some transactions by recording holding transactions with their requesting lock modes. Compared to the above two algorithms, his one works in $O(n+e)$ space and time complexity, where n is the number of blocked transactions and e is the number of edges in T-table and R-table. However, by aborting the current blocker whenever there is a deadlock, victim selection is so simple but far from the optimal[2]. Moreover, because each resource to be locked does not contain its own queue of blocked requests, whenever all the holders of a resource are completed, the whole T-table has to be searched to schedule some waiting requests and that scheduling might become unfair and contains the possibility of live-lock[4].

In contrast to the above listed schemes, the algorithm proposed here permits lock conversions, allows multiple lock modes, detects deadlocks whenever they occur and gives a victim selection mechanism which is sub-optimal without sacrificing the time and space complexity. The remainder of this paper is organized as follows. In Section 2, the underlying transaction model that is examined in this paper is explained. A storage efficient deadlock detection graph is proposed and its characteristics are shown is Section 3. In Section 4, the basic deadlock resolution strategy used in our scheme is described and our deadlock detection and resolution algorthm is presented. Preliminary simulation results are included in Section 5. Section 6 gives some concluding remarks.

2. Underlying Transaction Management

In our model of the database system, a transaction is a sequence of read, write operations and several transactions can run concurrently.

For ensuring serializability, our model takes *strict two-phase locking* [4], [5], which requires that a transaction has to lock a resource before it accesses the resource and all locks of a transaction are held until the transaction terminates. We allow *multiple lock modes*(IS, IX, S, SIX and X locks) [4], [8], [9] and our model is upward compatible with the *multiple granularity locking*(MGL) protocol[8] [9] in the sense that it integrated without changes into a system that supports a resource hierarchy.

We say that two lock requests for the same resource by two different transactions are compatible if they can be granted concurrently. The *compatibility matrix*, say *Comp*, for the given lock modes is represented in Table 1, where NL means No Lock and Comp[lock1, lock2] is true if lock1 and lock2 are compatible; otherwise, it is false. For example, Comp[S, IS] is true but Comp[IX, SIX] is false.

	NL	IS	IX	SIX	S	X
NL	t	t	t	t	t	t
IS	t	t	t	t	t	f
IX	t	t	t	f	f	f
SIX	t	t	f	f	f	f
S	t	t	f	f	t	f
X	t	f	f	f	f	f

Table. 1 Compatibility Matrix

	NL	IS	IX	SIX	S	X
NL	NL	IS	IX	SIX	S	X
IS	IS	IS	IX	SIX	S	X
IX	IX	IX	IX	SIX	SIX	X
SIX	SIX	SIX	SIX	SIX	SIX	X
S	S	S	SIX	SIX	S	X
X	X	X	X	X	X	X

Table. 2 Conversion Matrix

Sometimes, a transaction which holdes a resource might re-request the same resource to convert a lock from the granted mode to a more exclusive mode. We call such re-requests lock *conversions*. When a request is found to be a conversion, the granted mode of the requestor and the newly requested mode are used to compute the new mode, by using a *conversion matrix*, say *Conv*, which is represented in Table 2, with the granted mode as row and the requested mode as column. For example, when a transaction hold IX lock on a resource and re-requests the resource with S lock, the transaction eventually wants to hold SIX lock for the resource which is Conv[IX, S].

For scheduling lock requests in a first-in-first-out basis and to keep track of the requests of each transaction, the *lock manager* maintains a *lock table* which holds the following information for each resource to be locked: a holder list, a queue and a *total mode* of the holders. Each holder in a holder list takes three attributes: a transaction identifier (tid), a granted mode (gm) and a blocked mode (bm), i.e. (tid, gm, bm); each request in a queue takes two attributes: a transaction identifier (tid) and a blocked lock mode (bm), i.e. (tid, bm); and the total mode of the holder, assuming that n requests are in the holder list, where each one takes(T_i, gm_i, bm_i) and $1 \leq i \leq n$, is defined as follows: Conv[…Conv[Conv[gm_1, bm_1], gm_2], …bm_n].

When a request to a resource is received, whether the requestor is new one for the resource or that request is lock conversion is checked first. In the case of a new requestor for the resource, the queue status is checked. If the queue is not empty, the request is not granted and appended to the end of the queue. If the queue is empty and the requested mode is compatible with the total mode of the resource, then the request is added to the holder list, the total mode is replaced by Conv[current total mode, requested mode] and the request is granted. Otherwise, the request is appended to the queue. When a request is found to be a conversion, the new mode is computed by Conv[granted mode, requested mode] and if it is compatible with the granted mode of all the other holders, the granted mode of the requestor becomes the new mode, the total mode of the resource is recomputed as Conv[current total mode, requested mode] and the request is granted. Otherwise the granted mode of the requestor remains the same, the blocked mode is replaced by the new mode, the total mode of the resource is changed as before and the requestor is blocked until all the other granted modes are compatible with the blocked mode of the requestor.

For example, when two transactions T_i and T_j hold IS and IX lock respectively for a resource, the holder list of the resource has entries of (T_i, IS, NL) and (T_j, IX, NL), and the total mode is IX lock. Later, when transaction T_i re-requests the resource with S lock, because the new mode which T_i wants to hold for the resource is S lock(Conv[IS, S]) and it is not compatible with the granted mode of transaction T_j, this request is blocked, the holder list of the resource has entries of (T_i, IS, S) and (T_j, IX, NL), and the total mode of the resource becomes SIX. When transaction T_k requests S lock for the resource, because S and SIX are not compatible, the request cannot be granted and is appended to the queue with(T_k, S) entry. However, if transaction T_k requests IS lock and the queue is empty, it is granted.

When two or more requests in a holder list are blocked by lock conversions, for the given lock modes (IS, IX, S, SIX and X locks), the following observation can be found.

Observation 1. For any two requests which are blocked by lock conversions in a holder list, say(T_i, gm_i, bm_i) and (T_j, gm_j, bm_j),

1) if Comp[bm_i, bm_j], any one of the two requests can be scheduled in the presence of another,
2) if Comp[bm_i, gm_j] and not Comp[gm_i, bm_j], the request of T_i can be scheduled before that of T_j but

the reverse is not possible, and

3) if not Comp[bm_i, gm_j] and not Comp[bm_j, gm_i], any one of the two requests cannot be scheduled in the presence of another, i.e. this is a kind of deadlock.

Based on Observation 1, when a conversion is blocked, the position of the requestor, say(T_i, gm_i, bm_i), in the holder list is rearranged according to the following upgrader positioning rule(UPR):

1) If there are some requests whose bm are not NL and are compatible with bmi, put(T_i, gm_i, bm_i) right before the first request among them.
2) If UPR 1) cannot be applicable and there are some requests whose gm are compatible with bm_i and gm_i is not compatible with the bm of them, put(T_i, gm_i, bm_i) right before the first request among them.
3) If UPR 1) and 2) cannot be applicable, put(T_i, gm_i, bm_i) after all requests whose bm are not Nl and before all requests whose bm are NL.

When some holders are deleted from the holder list, owing to UPR and by putting all the blocked conversions ahead of the non-blocked ones in a holder list, checking whether some blocked request can be granted can start from the front of the holder list down to the end of it and can stop immediately when one cannot be granted or a non-blocked one is found. All the newly granted ones are put after the blocked holders and if the queue of the resource is not empty, the queue is checked to grant some requests from the first waiter. Once the total mode of the resource is compatible with the waiter's requested mode, the request is added to the holder list, the total mode is replaced by Conv[current total mode, requested mode], and the request is granted. This continues until the queue becomes empty or the total mode of the resource is not compatible with the waiter's requested mode.

3. Graph Construction

In this section, we introduce a new directed graph called a Holder/Waiter-Transaction Waited-By Graph(H/W-TWBG). Each vertex of H/W-TWBG is a transaction identifier and each edge $T_i \rightarrow T_j$ is labeled with H/W, where the completion of transaction T_i is waited by transaction T_j and either T_i is a holder of the resource which T_j is waiting(H-label) or another waiter in the queue of the resource(W-label). Edges in H/W-TWBG are constructed by the following *edge construction rules* (ECR):

1) For every two requests in the holder list of a resource, say(T_i, gm_i, bm_i) and (T_j, gm_j, bm_j), where (T_i, gm_i, bm_i) precedes(T_j, gm_j, bm_j), if not Comp[gm_i, bm_j] or not Comp [bm_i, bm_j], then add edge $T_i \rightarrow T_j$ with H-label, and if not Comp [gm_j, bm_i], then add an edge $T_j \rightarrow T_i$ with H-label.

2) For each request(T_i, gm_i, bm_i) in the holder list of a resource, find the first request(T_j, bm_j), if any, in the queue of the resource whose blocked mode bm_j is not compatible with either gm_i or bm_i, and add an edge Ti→Tj with H-label.
3) For every two requests, say(T_i, bm_i) and (T_j, bm_j), which are adjacent in the queue of a resource and (T_i, bm_i) precedes(T_j, bm_j), add an edge $T_i \rightarrow T_j$ with W-label.

Example 3.1. Assume that the following is a part of the lock table:

R1[SIX]: Holder((T_1, IX, SIX) (T_2, IS, S) (T_3, IX, NL) (T_4, IS, NL))

Queue((T_5, IX) (T_6, S) (T_7, IX))

According to UPR given in Section 2, the entry of T_1 precedes that of T_2 in the holder list and according to ECR, H/W-TWBG which is related to resource R1 is shown in Figure 3.1. Note that T_4 does not block any request and T_1 is not blocked by T_2.

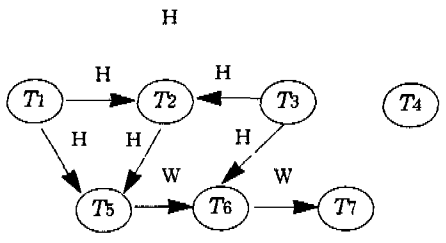


Figure 3.1 H/W-TWBG for example 3.1

In H/W-TWBG, each H-labeled edge is followed by a sequence (possibly zero) of W-labeled edges. We call a path which consists of one H-labeled edge and all of its following W-labeled edges, possibly empty, as a *Transaction Resource Request Path* (TRRP), by the way, it shows a partial status of the holder list and the queue of a resource. A cycle in H/W-TWBG thus can be defined as a cyclic sequence of two or more TRRPs such that a cycle detection problem can be transformed into finding those TRRPs which constitute a cycle.

Example 3.2. Assume that the following is a part of the lock table:

R1[X]: Holder((T_1, N, NL)) Queue((T_2, S) (T_3, S) (T_4, X)

R2[S]: Holder((T_4, S, NL) (T_5, S, NL)) Queue((T_6, X) (T_7, X) (T_1, S))

According to H/W-TWBG construction rule, Figure 3.2 can be obtained, where there are 3 TRRPs: $TRRP_1$(T_1, T_2, T_3, T_4), $TRRP_2$(T_4, T_6, T_7, T_1), and $TRRP_3$(T_5, T_6, T_7, T_1). Among those, $TRRP_1$ and TRRP2 constitute a cycle.

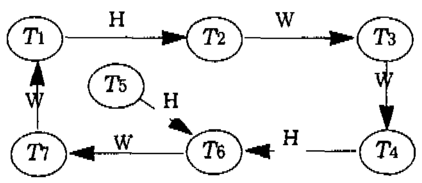


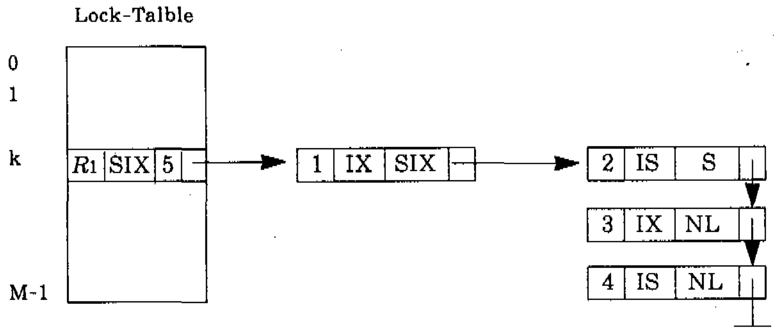
Figure 3.2 H/W-TWBG for example 3.2

We now give the detailed data structures for the implementation of H/W-TWBG as well as the lock table representation. Because those labels used in H/W-TWBG are not explicitly represented and some additional variables are included in its data representation, we will call the latter TWBG to differentiate it from H/W-TWBG. Lock-Table which has an entry for each locked resource is an array [0..M-1] of record with *rid* (record identifier), *tm*(total mode), *queue*(queue pointer), and *holder* (holder list). Each holder of a resource is represented by a record with *tid*(transaction identifier), *gm* (granted mode), *bm*(blocked mode), and *next*(pointer to the next holder). Each transaction has an integer value between 1 and N as its identifier and gets its entry in TWBG according to its identifier. TWBG is an array of size N and contains *visit*, *ancestor*, *pr*(the position of the resource in the Lock-Table), *in-cycle*, *waited*, and *current*. The variable *visit* and *ancestor* are used for the detection of cycles in TWBG and are initialized to 0. During the search of a cycle, once a vertex is probed, visit becomes nonzero and ancestor keeps the returning vertex value to be used in the backtracking of a depth-first-search(DFS)[7]. The variable *in-cycle* is set when the vertex is involved in any cycle, the variable *waited* points to the first edge incident to the vertex and the variable *current*, which is initialized to *waited*, indicates the next edge to be searched in a vertex.

Each edge incident to a vertex is represented by a record with *lock*(lock mode), *tid*(transaction identifier), and *next*(pointer to the next edge). The edge $T_i \rightarrow T_j$ with H-label in H/W-TWBG is represented as an edge(NL, T_j) in TWBG[i]. waited. However, W-labeled edges in H/W-TWBG are implicitly represented at TWBG by maintaining a queue of a resource as follows: Let a resource, say R_m, has its entry in Lock-Table[k].

1) Lock-Table[k]. queue takes 0 or the identifier of the first transaction to be in the queue.
2) For each request(T_i, bm_i) to be in the queue, TWBG[i]. pr is set to k and an edge (bm_i, 0) is put on the list of TWBG[i]. waited. However, if the request is followed by another request, say(T_j, bm_j), an edge(bm_i, T_j) is put on the list of TWBG[i]. waited. Actually, this is the edge $T_i \rightarrow T_j$ with W-label in H/W-TWBG.

For example, for the given status in Example 3.1, the corresponding Lock-Table and TWBG are shown in Figure 3.3.



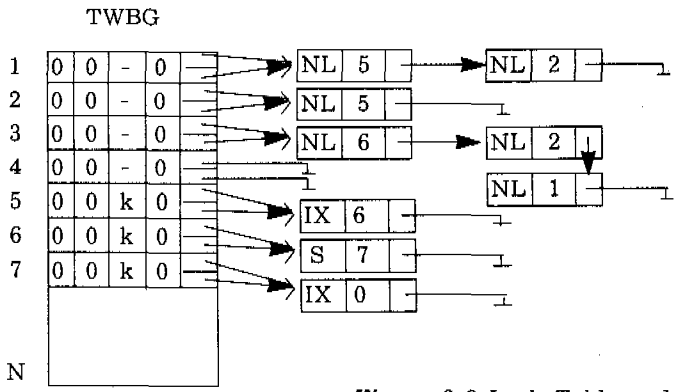


Figure 3.3 Lock-Table and TWBG for example 3.1

For maintaining the queue of a resource easily, when multiple edges are put on an entry of TWBG, the W-labeled edge, if any, i.e. the edge whose lock is not NL, is put at the front of the list. Based on the construction of Lock-Table and TWBG, in the next section, our deadlock detection and resolution algorithm is presented.

4. Deadlock Detection and Resolution

In continuous deadlock detection and resolution, whenever a lock request can not be granted immediately, the existence of a deadlock is checked and resolved[9]. When a deadlock is found, abortion of the current blocker might be a reasonable solution if the size of every transaction is so small[13]. Compared to its simplicity, if the size of each transaction varies in large, selection of victims may render a big difference in database performance and user's waiting time variance[2]. Another mechanism for deadlock resolution is the so called one-cycle-at-a-time[9]. In this method, whenever a cycle is found, a transaction which has minimal cost among those transactions involved in the cycle is selected as a victim and aborted.

There can be several criteria for deciding cost of each transaction, for example, number of locks it holds, statring time of it, the amount of CPU and I/O time which it consumed and so on [2],[3],[4]. We assume that the cost of each transaction is determined by some combination of the above methods and it is stored in a cost-table such that Cost(Ti) indicates the cost of transaction *Ti*. In this section, a continuous deadlock detection and victim selection mechanism without sacrificing the time and space complexity is given.

Under the construction mechanism of the deadlock detection graph, called H/W-TWBG, because a cycle in the graph can be defined as a cyclic sequence of two or more TRRPs, guidelines for the identification of victims in a deadlock cycle can be defined by the following observation.

Observation 2. Two consecutive TRRPs in H/W-TWBG, say TRRP1($T_i \rightarrow \cdots \rightarrow T_j$) and TRRP2($T_j \rightarrow \cdots \rightarrow T_k$) where T_j is blocked at resource R_x, can be disconnected by the following *TRRP disconnection rules*:

1) By aborting T_j, *TRRP1* and *TRRP2* are disconnected.
2) If the last edge of *TRRP1* is W-labeled and the blocked mode of T_j is compatible with the total mode of resource R_x, let the set of requests which precede the request of T_j in the queue of resource Rx and whose blocked modes are not compatible with the total mode of resource R_x be ST. By repositioning those requests in ST right after that of T_j in the queue, *TRRP1* and *TRRP2* are disconnected.

For instance, the cycle in H/W-TWBG in example 3.2 has two TRRPs: one with(T_1, T_2, T_3, T_4) and the other with(T_4, T_6, T_7, T_1). T_1 and T_4 are the tail transactions of the starting edge of each TRRP. According to Rule-1, by aborting either T_1 or T_4, the deadlock can be resolved. Moreover, because the total mode of resource R_2 is compatible with the blocked mode of transaction T_1 at resource R_2, Rule-2 can be applied such that ST becomes {(T_6,X), (T_7,X)} and by repositioning (T_6,X) and (T_7, X) right after (T_1,S) in the queue, the request of T_1 for resource R_2 can be granted and the deadlock is resolved.

In applying TRRP disconnection rule for each cycle found, to select the minimal cost victim and to prevent some requests from repeated repositioning by the application of Rule-2, the victim-cost is calculated as follows: When Rule-1 is applied, victim-cost becomes the cost of the corresponding transaction. However, when Rule-2 is applied, it becomes the summation of the cost of each transaction in ST divided by 2. We select the minimal cost among those and if it came from Rule-2, the cost of each transactin in ST in increased by some constant value.

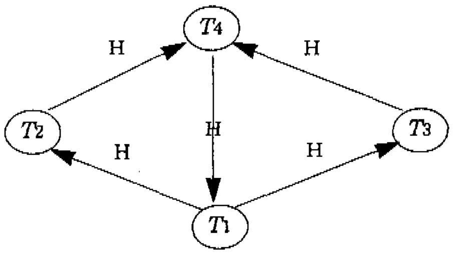
When a blocked lock request causes only one cycle, by applying TRRP disconnection rule, the minimal cost victim can be selected. However, when it causes several cycles to be occurred simultaneously, to get sub-optimal solution, we take the following strategy: The deadlock detection starts from the current blocker and whenever a cycle is found, TRRP disconnection rule is applied, If the victim comes from Rule-2, the corresponding requests are repositioned and their costs are increased by some value. However, when it comes from Rule-1, the victim is added to the *victim set*. When there are no more cycles reachable from the current blocker, the victim set is called as the abortion *set*-0(ABS-0) and the cumulative cost of transactions in ABS-0 is compared with the cost of the following three more abortion sets and the minimal cost set among those is selected and its consituent transactions are aborted.

ABS-1 The current blocker(cb).

ABS-2 The set of the last transactions of each TRRP which starts from cb and is involved in any cycle.

ABS-3 If cb is a lock upgrader of a resource, the set of holders of the resource which made cb to be blocked. Otherwise, the set of holders of the resource which blocks any one of the requests in the queue.

Example 4.1. Assume that H/W-TWBG and Cost-Table are given as in Figure 4.1.



H/W-TWBG

	T_1	T_2	T_3	T_4
Case 1	3	5	2	4
Case 2	10	4	4	5

Cast-Table

Figure 4.1 H/W-TWBG and Cost-Table for example 4.1

There are two deadlock cycles: one with $\{T_1, T_2, T_4\}$ and the other with $\{T_1, T_3, T_4\}$. Assume that T_1 is the current blocker. For the first case of Cost-Table, each abortion set is defined as follows: ABS-0= $\{T_1, T_3\}$, ABS-1= $\{T_1\}$, ABS-2= $\{T_2, T_3\}$ and ABS-3= $\{T_4\}$. Among these four sets, ABS-1 takes minimal cost and T_1 which is the current blocker is aborted. For the second case of Cost-Table, each abortion set is defined as follows: ABS-0= $\{T_2, T_3\}$, ABS-1= $\{T_1\}$, ABS-2= $\{T_2, T_3\}$ and ABS-3= $\{T_4\}$. Among these four sets, ABS-3 takes minimal cost and T_4 is aborted.

We now give the details of our continuous deadlock detection and resolution algorithm which is a kind of a depth-first-search (DFS) method.

```
Algorithm Continuous-Detection-Resolution
/*A transaction cb requests a lock Lcb on a resource Rcb */
Input: cb, Lcb, Rcb, Lock-Table and TWBG.
Output: abortion-list of transactions to be aborted and
        grant-list of transactions to be granted.
begin
Step 0/* edge construction */
  modify Lock-Table and TWBG.
  if cb is granted, return.
```

```
Step 1/* search cycles */
  v:=cb; TWBG(v).ancestor:=-1;TWBG(v).visit:=-1;
  visit-top:=v; abortion-cost:=0;
  abortion-list:={}; change-list:={};grant-list:={};
  repeat
    if TWBG(v).current=nil then begin/* backtracking */
       w:=v; v:=TWBG(w).ancestor;TWBG(w).ancestor:=0;
       if(v≠-1) and (TWBG(w).in-cycle)
       then TWBG(v).in-cycle:=true;
       if(v≠-1)
       then TWBG(v).current=TWBG(v).current^.link
    end else begin
       w:=TWBG(v).current^.tid;
       if w=0 /* v is at the end of a queue*/
       then TWBG(v).current:=TWBG(v).current^.link
       else if w=cb /* a cycle is found */
         then Victim-Selection(v)
         else if TWBG(w).current=nil then begin
           if TWBG(w).in-cycle
           then TWBG(v).in-cycle:=true;
           TWBG(v).current:=TWBG(v).current^.link
         end else begin /* forward search */
           if TWBG(w).visit=0 then begin
              TWBG(w).visit:=visit-top; visit-top:=w
          end;
          TWBG(w).ancestor:=v; v:=w
         end
    end
  until v=-1;
  TWBG(cb).ancestor:=0;
  if(abortion-cost=0) or (abortion-list={cb})
  then goto step 3.

Step 2/* Consideration of TRRPs of the current blocker*/
  Find ABS-1,2 and 3.
  Replace abortion-list as the minimal cost set
  among those and the one found at Step 1.

Step 3/* table clearance and grant-list decision*/
  for each transaction v in abortion-list do
    if v is in grant-list
    then delete v from abortion-list
    else for each resource Rv which v holds/is waiting for do
       Delete v from holder-list or queue of Rv while
       modifying Lock-Table, TWBG and constructing
       grant-list according to our scheduling policy and ECR.
  for each resource in change-list do
    modify Lock-Tabel and TWBG, and construct
    grant-list according to our scheduling policy and ECR.
w:=visit-top;
while w≠-1 do begin
    v:=TWBG[w].visit; TWBG[w].visit:=0;
    TWBG[w].in-cycle:=false;
    TWBG[w].current:=TWBG[w].waited; w:=v
  end
end;/* end of algorithm Continuous-Detection-Resolution*/

Procedure Victim-Selection(var s:integer;)
/*The calling value of s is the vertex which found a cycle.*/
/*The return value of s is the new search starting vertex.*/
begin
  v:=s; min-cost:=cost(cb); min-node:=cb;
  /*Apply TRRP disconnection rule*/
  new-trrp:=true; rule-2:=false;
  repeat
    TWBG[v].in-cycle:=true;
    if TWBG[v].current^.lock=NL then begin
      if cost(v)<min-cost then begin
        min-node:=v; min-cost:=cost(v);
        rule-2:=false
      end;
      new-trrp:=true
    end else/*W-labeled edge*/
      if new-trrp then begin/* a new TRRP*/
         w:=TWBG[v].current^.tid; res:=TWBG[w].pr;
         if Comp[Lock-Table[res].tm,TWBG[w].waited^.lock]
         then begin/* apply Rule-2*/
           /*Change-Cost-Calculation*/
                 next:=Lock-Table[res].queue;
                 t-cost:=0; t-point:=0;
                while next≠w do begin
                   if not Comp[Lock-Table[res].tm,
                                  TWBG(next).waited^.lock)
        then begin
          t-cost:=t-cost+cost(next); t-point:=next
        end;
        next:=TWBG(next).waited^.tid
      end;/* end of while loop*/
      t-cost:=t-cost div 2;
      if t-cost≤min-cost then begin
        min-node:=w; min-cost:=t-cost;
        L-point:=t-point: rule-2:=true
        end
      end;/* end of apply Rule-2*/
      new-trrp:=false
    end;/* end of a new TRRP */
    w:=v; v:=TWBG(v).ancestor
  until(w=cb);

  /*the victim is decided. starting from s, modify ancestors*/
  v:=s;
  if rule-2=false then begin/* victim from Rule-1*/
    while v≠min-node do begin
      w:=TWBG(v).ancestor; TWBG(v).ancestor;=0; v:=w
    end;
    w:=TWBG(v).ancestor;
    TWBG(v).current:=nil: TWBG(v).ancestor:=0,
    abortion-list:=abortion-list+{v};
    abortion-cost:=abortion-cost+min-cost:
    s:=w /* victim's ancestor is a new search starting vertex */
  end else begin /* victim from Rule-2*/
    while v≠L-point do begin
      w:=TWBG(v).ancestor; TWBG(v).ancestor:=0; v:=w
    end;
    s:=L-point;
    /*the last transaction in ST is a new search starting vertex*/
    change-list:=change-list+{TWBG(min-node).pr};
    /*Repositioning while setting ancestors of vertices*/
    new-ancestor:=0; pre:=0; res:=TWBG(min-node).pr;
    next:=Lock-Table(res).queue;i-point:=min-node;
    while i-point≠L-point do begin
      if not Comp(Lock-Table(res).gm, TWBG(next).waited^.lock)
      then begin/*rearrange "next" right after "u-point"*/
    if TWBG[next].ancestor≠0 then begin
      if new-ancestor≠0
      then TWBG[next].ancestor:=new-ancestor:
      new-ancestor:=next
  end;
  cost(next):=cost(next)+1;
  temp:=TWBG[next].waited^.tid;
  TWBG[next].waited^.tid:=TWBG[i-point].waited^.tid;
  TWBG[i-point].waited^.tid:=next;
  i-point:=next;
  if Lock-Table[res].queue=next
  then Lock-table[res].queue:=temp
  else TWBG[ore].waited^.tid:=temp;
  next:=temp
end else begin
  if(new-ancestor=0) and (TWBG[next].ancestor≠0)
  then new-ancestor:=TWBG[next].ancestor;
  TWBG[next].ancestor:=0;
       pre:=next; next:=TWBG[next].waited^.tid
      end
    end/* end of while loop*/
  end;/*end of the case when rule-2=true*/
  if s≠-1 then TWBG[s].current:=TWBG[s].current^.link
end;/*end of procedure Victim-Selection*/
```

Space complexity of our algorithm becomes O($(n+e)$), where n is the number of transactions in TWBG and e is the number of edges in it. When there is not any cycle in the graph, O((e)) time is required to search all the vertices which are reachable from the current blocker. Different from listing all the elementary cycles in a directed graph as in Johnson's algorithm [26], when a cycle is found at Step 1, it is searched again to find the minimal cost victim and search resumes at the ancestor of the aborted one with setting the variable *current* of aborted one as nil or at the last transaction of ST with setting the variable *current* of the transaction as next edge of it. By doing this way, the same cycle can not be searched again and total number of cycles searched(c') can not exceed the number of elementary cycles (c) in the graph and also can not be greater than the number of transactions (n) in the graph. Because procedure Victim-Selection can be done in 0((n)) time, the total time complexity of the algorithm becomes $O(e+n*c')$.

5. Simulation Results for

5. Simulation Results for the Proposed Scheme

Our simulator for studying the performance of the proposed continuous deadlock detection and resolution algorithm is based on the closed queueing model. Whenever a transaction leaves the system, a new one is generated and added to the ready queue to bound the maximum number of transactions, called the *multiprogramming level*(MPL), in the system. However, note that there might be fewer than MPL transactions that are ready to run when a set of transactions block themselves. To simplify our simulation, one level hierarchy for the locking is assumed; each resource is a granule and there is not any data hierarchy which implies that only S locks, X locks, and lock conversions from S locks to X locks are supported.

Initially, all transactions are generated and put into the ready queue. A random number generator is used during the transaction creation process to define the transaction size within certain limits, the resources to be accessed, and the lock request mode for each resource. Each transaction accesses the database randomly, in other words, each resource has an equal chance to be referenced. A FIFO serice descipline is employed for the ready queue. Each transaction makes its lock request one at a time. If its request is grantable by our locking policy, its request is set and it accesses its requested object. If there is more than one object left to be accessed, the transaction re-enters the ready queue and waits for its turn to request the next resource. If a transaction finishes, it leaves the system by releasing all the locks it holds. These locks are deleted from the Lock-Table and some blocked transactions waiting for the related resources might be granted. When a lock request cannot be granted, the requesting transaction is blocked and the existence of a deadlock is checked.

We model the environment as 1000 resources and each trnsaction requests at least 4 resources, at most 12 resources, and its mean value is 8 resources. We simulated six different MPLs: 5, 25, 50, 100, 200 and 400 transactions. In any MPL, simulation terminates when a pedefined number of transactions, in our case 1000 transactions, commit. Transactions are allowed to request S locks, X locks, and lock conversions, where 80% of the requesting modes are S locks and 20% of them are X locks and among those S locks, 6.25% are converted to X locks. In this experiment, source requests are uniformly distributed such that each resource has the same probability of being requested.

We first examined how those abortion sets defined for continuous deadlock detection and resolution algorithm affects to each other. To see the exact portion of each abortion set, ABS-0 is classified into two sets: set-0 and set-4, where set-0 indicates the set of victims to be aborted for each cycle found and set-4 indicates the set of transactions which are repositioned according to the TRRP disconnection rules. ABS1, 2, and 3 are represented as set-1, 2, and 3 respectively. We also compared the following three different deadlock resolution stategies:

1. Resolve deadlock according to our scheme.
2. Resolve deadlock by aborting the minimal cost transaction for each cycle.
3. Resolve deadlock by aborting the current blocker.

We included strategy 2 because it gives the best results for continuous deadlock detection and resolution in Agrawal et al's report[2]. In our graphical representations, we use ps, min, and cb referring to strategy 1, 2, and 3 respectively. To compare the relative performance of each strategy, two ratios, called blocking ratio and restart ratio, are calculated. The blocking ratio is defined as the number of blocked transactions per committed transaction and the restart ratio is used to describe how many transactions are restarted on the average for every committing transaction. Initiallty, the variable tr-blocked (tr-aborted) is set to 0. Whenever a transaction is blocked (aborted), tr-blocked (tr-aborted) is incremented by one. After 1000 transactions are committed, blocking ratio (restart ratio) is set to tr-blocked/1000(tr-aborted/1000). In the experiment, every strategy was tested by using six different random number generators. The best and the worst outcomes were not considered. Therefore, our results are showing the average of four simulation runs.

The percentage of each abortion sets, the blocking ratio, and the restart ratio for the experiment are listed in figure 5.1, 5.2, and 5.3 respectively.

At lower MPLs, set-1, 2, and 3 are never selected because deadlocks have a single cycle. At the hightest MPL (400 transactions), they are selected 14.35% of the time. Set-4 is selected more frequently when higher number of transactions are running in the model. If there are enough deadlocks, the effects of choosing set-4 can be easily seen. As we said earlier, when set-4 is selected, no transactions are actually aborted. Instead, their positions in the queue are repositioned. Moreover, their costs are increased to avoid the live-lock.

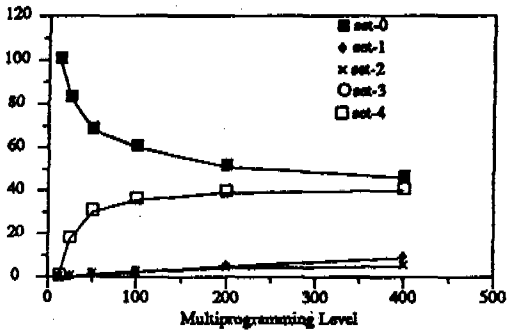


Figure 5.1 Selection percentage of abortion sets vs. MPL

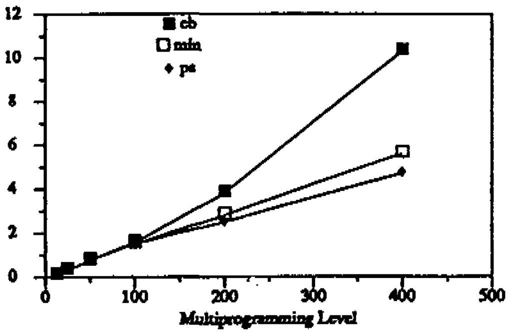


Figure 5.2 Blocking ratio vs. MPL

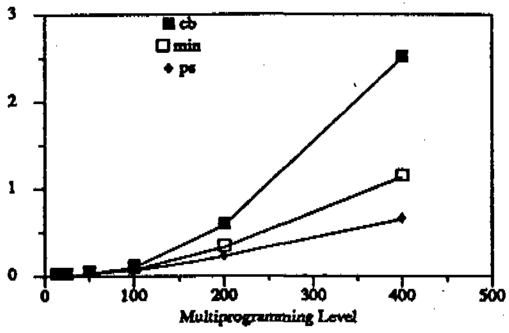


Figure 5.3 Restart ratio vs. MPL

As shown in figure 5.2 and 5.3, unless a higher MPL is reached we cannot distinguish one strategy from another because there are less lock conflicts, and consequently less number of deadlocks. Although the restart ratio of strategy 1 is 27.27% better than strategy 2 when 100 transactions are in the system, this does not affect the blocking ratio considerably because the total number of deadlocks is small. At 200 transactions and above, we experienced remarkable differences. For example, at 200 multiprogramming level, stategy 1 outperforms strategy 2 by 6% in blocking ratio and 30% in restart ratio. For 400 transactions, these figures are 13.03% and 36.80% respectively. Therefore, the use of strategy 1 should be considered when the database is heavily used.

Restarting the current blocker to resolve deadlocks does not hurt much until multiprogramming level of 100 transactions is reached. After that point, it should not be the strategy of choice. To be more specific, at the 400 multiprogramming level, strategy 1 defeats strategy 3 by 53.49% in blocking ratio and 72.37% in restart ratio.

6. Conclusion

In this work, we have developed an efficient deadlock detection and resolution algorithm in the environment of sequential transaction processing with multiple locking modes and lock conversions based on a new deadlock detection graph and two types of deadlock resolution strategies: one by aborting a transaction in a cycle and the other by switching the order of lock requests in the queue of a resource without having the risk of live-lock.

To get sub-optimal solution for the victim selection without sacrificing time and storage complexity, four different sets of possible victims to resolve all deadlock cycles in the system are defined such that we got

O(n+e) storage space and O(e+n*c') time complexity algorithm, where e is the number of waited-by edges, n is the number of transactions in the database system and c' is the number of cycles which were actually searched such that c' is not greater than c, the number of elementary cycles in the graph, and c' is not greater than n.

We also compared our resolution strategy with others which showed that our one gives less blocking and restart ratios especially in the higher multiprogramming level.

References

1. R. Agrawal, M.J. Carey, and D.J. DeWitt, "Deadlock Detection is Cheap," *ACM SIGMOD RECORD*, Vol.13, No.2, pp. 19-34, January 1983.
2. R. Agrawal, M.J. Carey, and L.W. McVoy, The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Trans. on Sofware Eng.*, Vol.SE-13, No.12, pp. 1348-1363, December 1987.
3. R. Agrawal, M.J. Carey, and M. Linvy, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM TODS*, Vol.12, No.4, December 1987.
4. P.A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.
5. C.J. Date, "An Introduction to Database Systems: VolII," *Addisin-Wesley*, 1983.
6. A.K. Elmagarmid, "Deadlock Detection and Resolution in Distributed Processing Systems," Ph. D. Dissertation, The Ohio State University, 1985.
7. S. Even, "Graph Algorithms," *Computer Science Precess*, 1979.
8. J.F. Garza and W. Kim, "Transaction Management in an Object-Oriented Database System," in *Proc. of 1988 ACM SIGMOD Intl. Conf.*, Chicago, IL., 1988.
9. J.Gray, "Notes on Database Operating Systems," in Lecture Notes in Computer Science 60, Advanced Course on Operating Systems, ed. G. Seegmuller, Spirnger Verlag, New York, 1978.
10. J.Gray and A.Reuter, "Transaction Processing," Version I of the Slides, Summer 1987.
11. B.Jiang, "Deadlock Detection is Really Cheap," *ACM SIGMOD RECORED*, Vol.17, No.2, June 1988.
12. D.B.Johnson, "Finding all the Elementary Circuits of a Directed Graph," *SIAM J.Computing*, Vol.4, No.1, March 1975.
13. K.H. Pun and G.G. Belford, "Performance Study of Two Phase Locking in Single-Site Database Systems," *IEEE* Trans. *Software Eng.*, Vol.SE-13, No.12, pp. 1313-1328, December 1987.