submitted to ACM-SIGMOD '89.

Concurrency Control of Bulk Access Transactions on a Parallel Disk Database Machine

OHMORI, Tadashi KITSUREGAWA, Masaru
TANAKA, Hidehiko
Dept. Electrical Engineering, The University of Tokyo

Abstract

This paper proposes the concurrency control scheduler for Bulk Access Transactions (BAT): the transactions to access bulk of data on the database-disks. The target environment is the database machine composed of parallel disk modules with the range-partitioned file placement. For the concurrent processing of BATs, 'Weighted Transaction Precedence Graph' (WTPG) and the cautious scheduler using it are proposed. WTPG is a transaction precedence graph with weighted edges. By enforcing the serializable order to make the shortest critical path in the WTPG, the scheduler generates a schedule of less data/resource contention. Simulation results show the scheduler using WTPG outperforms the others based on the two-phase lock, the atomic lock, and the optimistic lock by 30% to 100% in the throughput.

1 Introduction

The increase of 24-hours database service expands the need to compute more batch jobs in much shorter time. A batch job is usually the transaction to read or write as large bulk data as hundreds of mega byte on the database-disks. For instance, the jobs in the statistic processing read the major part of files and update the major part of another file. e.g. "take the join operation of two relations after the selection on their non-indexed attributes and update them depending on the joined result". This category of transactions have the long lifetime because of the large amount of data-access to the disks. In order to execute such the transactions in a short time, it is required to execute them concurrently on a dedicated database machine. This paper refers to the transactions as *Bulk Access Transactions* (BAT) and proposes the serializable scheduler for the concurrent processing of BATs on a database machine.

The BAT is categorized as the Long Lived Transaction (LLT) in [Gr81], while a cooperative (CAD-oriented) transaction [Ban85] is also a LLT. Previous works in the concurrency control discussed the short term transaction (STT) processing [Gr78, Car84, Bh88], the LLT processing with many STTs [Bay86,Mo87], and the LLT processing for cooperative transactions [Ban85, Kl85]. The concurrent processing of many BATs is not, however, discussed so far in itself.

In the STT processing, the data/resource contentions are critical in the performance of the well-known locking protocols [Tay85, Ag85]. The data contention refers to the chain of blocked transactions. The resource contention refers to the congestion of the disks. In the STT processing such as a debit-credit system, either of the contentions is usually very low. The conventional locking protocols work in such the environment. In contrast, the BAT processing has the high data/resource contentions inherently: Because the locking granule must be very coarse such as a half of a relation, and all the disks are often congested by accessing bulk of data. Hence the conventional locking protocols for the STT processing will not work in the BAT processing. In the LLT processing for cooperative transactions, most of previous works decompose a LLT into a set of STTs by relaxing the serializability [Mo87, Ban85, Lyn83]. Since they go towards the CAD/CAM application, it is hard to apply their methods straight to the BAT processing under the serializability.

We develop a new concurrency controller for the BAT processing. Our target environment is the database machine composed of the parallel disk modules with the range-partitioned file placement [DeW86].

In the range partitioned file placement, each relation is horizontally partitioned by the range of values on the specified attribute. e.g. In storing a relation `emp(Id,Name,Age)` on three disk modules, it is divided into three partitions by the ranges on `Id`: `Id` $<$ 10, 10 $\leq$ `Id` $<$ 20, and 20 <= `Id`. Each disk module stores one partition per relation and has a clustered index on any one attribute: e.g. `Name`. Consequently the range query on the partitioned attribute hits the partitions on a few disk modules among all the disks. A BAT accesses the major part of the hit partition. e.g. in the above example, the query

"`retrieve emp where Id < 10 and Account > 1000`" accesses the whole partition of the range '`Id < 10`'. Thus only one disk module storing it executes the bulk data processing for the query. It causes the load imbalance among the disk modules.

Under the high data/resource contention, the imbalance gets greater and prevents the parallel disk modules from running in parallel. Therefore the concurrency controller for the BAT processing should generate the serializable schedule of less data/resource contention so that all the disk modules execute the bulk data processing in parallel as much as possible.

For the purpose, we propose *Weighted Transaction Precedence Graph* (WTPG) and a cautious scheduler [Kat85] using it. The WTPG is a transaction precedence graph with the weighted edges. The weights represent the 'cost' for executing the transactions. By enforcing the serializable order to make the shortest critical path in the WTPG, the scheduler generates a schedule of less data/resource contention. It is NP-hard to find this 'desirable' serializable order in a given WTPG. We use a chain-form WTPG, where the 'desirable' order is found in $O(n^2)$ (n: the number of nodes in WTPG).

In the scheduler using WTPG, every transaction must predeclare all its read/write dataset and its 'cost' information until the commitment. By using them, the scheduler has no deadlock, less chain of blocking, and need not hold the locks until necessary. The simulation results show the proposed scheduler achieves the higher throughput than the others using the two phase lock, the atomic lock, and the optimistic lock by 30% to 100%.

The protocols in [Ab88] are the first that uses the cost information of transactions. Our scheduler using WTPG enforces the 'desirable' order of transactions by the prediction, while the 'undesirable' blocked transactions are aborted in [Ab88].

The rest of the paper is organized as follows: Section 2 describes our model and assumptions. In Section 3, Weighted Transaction Precedence Graph and a scheduler using it are proposed. Section 4 presents the $O(n^2)$ time algorithm to find the serializable order of the shortest critical path in a chain-form WTPG. Some implementation issues are also described. Section 5 gives simulation results and discussions. Finally Section 6 summarizes the paper.

2 Model and Assumptions

2.1 target environment

This paper assumes the 'Shared Nothing' database machine and the 'range-partitioning' file placement on it.

The 'Shared Nothing' database machine [Bh88] is composed of parallel disk modules interconnected by a network. Each Disk Module (DM) is a computer with disks for the database. It executes basic data processing such as filtering, clustering, sorting, etc. This category includes many database machines [DeW86, DeW87b].

All the relations are stored by the 'range partitioned' file placement on multiple DMs [DeW86]. Each relation is divided into fragments called 'partition' by the range of values on the specified attribute. Each DM stores one partition per relation, and has a clustered index on any one attribute.

Consequently a range query on the partitioned attribute hits the partitions on a few DMs. It causes the load imbalance among the DMs and degrades the throughput of BAT. A simple solution for the imbalance is to use 'random partitioned' or 'hash partitioned' file placement: a file is distributed randomly or by hash to multiple disk modules [DeW86]. Then a range-query will access the uniform amount of data to each DM. Thus all the disk modules run the bulk data processing in parallel. These schemes cannot, however, store the sorted files or VSAM-clustered files in their own rights. Since these file organizations are important for the short-term transaction processing, we assume the range partitioning.

2.2 transaction model

A BAT is modeled as a serial sequence of read/write step to a partition. The read/write step is abbreviated as *step* in the rest of the paper. Each step is assumed to have the range selection on the partitioned attribute, where the range is that of the accessed partition. In the locking protocols, a read/write step to a partition d requires a shared(S)/exclusive(X) lock on d. The step is executed only when it holds the lock on d. Lock mode escalation is allowed. All the locks are held until the commitment because of the recovery. All the

read/write dataset of a BAT can be predeclared at its start.

We adopt a partition as a locking granule. A lock on a partition represents a predicate lock on the range-value of the partition in the partitioned attribute. When updating data in a partition, we don't use a record-level lock but adopt a X-lock on the whole partition. Because a BAT is supposed to update a major part of the partition and the record-level X-lock will almost always conflict with a shared predicate lock on the partition.

A cost model of a BAT is defined only by data-accessing to the disks. The network delay, the control overhead, and the internal computing time in transactions are ignored. Because the lifetime of a BAT is dominated by the time for accessing bulk of data to disks. Even in the join operation, the assumption is justified by the stream oriented database machine and the hash join algorithms [DeW86, Nak88].

A read/write step is given a cost in proportion to the size of the accessed data. The unit of size for a partition is represented by *unit* itself. We assume that one *unit* of size is large enough to ignore any other overhead except the data access on the disks.

In read-accessing a% of a partition P, the read step `r(P)` is given the cost $a|P|$, s.t. $|P|$ is the size of P. e.g. if P is 10MB and a unit of size is 1MB, P is given the size 10 unit and the 50% read access `r(P)` is given the cost 5 unit.

Note that the read step `r(P)` has a shared lock on the whole partition P but may access only a part of it by using the clustered index on another attribute of P.

In write-accessing a% of P, the write step `w(P)` is given the cost $2a|P|$. The factor 2 is given because it must read the data to be updated from the disks again.

All the updates are not deferred but written back to disks immediately. Because it is expensive to keep all the bulk-updated data in main memory until commitment. Hence the cost from the commitment to the completion is negligible.

Each step is executed without interruption. Once a step starts on a disk module, its execution is not interrupted by that of the other steps. We adopt the un-interruptable model because, in the BAT processing, it matters how quickly a large bulk of data is retrieved from the disks. One answer is to store a partition in the contiguous disk-blocks and retrieve it without interruption [Kit86].

Figure 1: transaction model

Example: Fig.1 illustrates the four transactions T1 to T4, the five partitions `A`, `C`, `D`, `E`, `F` with their sizes, and their placement on the two disk modules DM1 and DM2. `r`$i(P)$ (or `w`$i(P)$) is the read (or write) step to the partition P by the transaction Ti. Each read step is assumed to access 100% of the partition. Each write step is assumed to access 50% of the partition. Thus the cost of a step to a partition is the size of the partition itself. The `commit` step comes at the last of each transaction. It is omitted in the figure. []

2.3 scheduler model

We assume the centerized concurrency controller (CC) which is invoked when one of DMs gets idle and requests a new step to run on it. CC runs as follows:

Step0: invoked when a disk module DMi ends the execution of one step and gets idle.

Step1: find a step q from RQ[DMi] where q must obey the concurrency control protocol of CC. RQ[DMi] is the ready queue for the steps to access a partition on DM_i.

Step2: If q is found, grant its lock and run it on DMi.

Step3: If not found, DMi sleeps in a unit of time. After the sleeping, DMi invokes CC again. []

First-Come-First-Serve (FCFS) strategy is assumed in all the Ready Queues. By invoking CC when a disk module gets idle, we can partially avoid the situation where a transaction holds a lock but cannot be executed because of the disk-congestion. In the situation, the locks are uselessly held so long that another transaction cannot run on another non-congested disk. Under the high data contention in the BAT processing, the 'useless' locking should be avoided as much as possible.

2.4 scheduling example

Suppose that the four transactions in Fig.1 are ordered in the sequence T1, T2, T3, T4 in the ready queue (RQ) of the concurrency controller (CC) at the start. Fig.2-a is a Gantt

Figure 2: Gantt chart of the transactions in Fig.1

chart where CC uses Cautious 2 Phase Lock protocol (C2PL) [Nis87]. In Fig.2, one clock is the time for accessing a data of size 1 unit on one disk module.

In C2PL, every transaction must predeclare its read/write data set at its start. C2PL grants a step q to a data d iff the lock of q is not blocked by the current lock held on d AND it causes no deadlock to grant q in the future. For the deadlock detection, C2PL uses the acyclic test in the transaction precedence graph. It is implemented on the Shared Nothing database machine as follows: each DM has its own ready queue RQ[DMi] and the lock manager for its partitions. When getting idle, each DM selects a non-blocked step in RQ[DMi] and ask the CC if the precedence constraint caused by granting the step makes no deadlock.

The C2PL scheduler generates the serializable order (SR-order) { T1, T2 → T3 → T4 }: (T2 → T3 says 'T2 precedes T3 in the serializable order'.) In Fig.2-a, by the FCFS discipline, CC grants `r1(D)` for `DM1`, `r2(A)` and `r3(C)` for `DM2` until clock2. At clock2, `w4(A)` is blocked by `r2(A)`. The chain of blocked transactions { T2 → T3 → T4 } makes DM1 idle after the clock 7. `r1(D)` makes DM1 busy till clock 4 and delays the commitment of T2. These two phenomena make the large idle time on both the disk-modules.

The phenomena are the 'data-contention thrashing' and the 'resource-contention thrashing' in [Tay85]. Both the thrashings are inherent in the BAT processing because of the very coarse locking granule and bulk data accessing. In the range-partitioned file placement, the thrashings prevent the parallel DMs from executing the bulk data processing in parallel. It degrades the throughput of BAT.

Our solution is to predict the possible thrashing by data/resource contentions and avoid it. Fig.2-b is the case where CC runs T3's step `r3(C)` at clock 0 and `w3(A)` at clock 1 on DM2. The generated SR-order is { T1, T3 → T2, T3 → T4 }. Under the order, both the DMs execute the bulk data processing in parallel. Thus the scheduler for the BAT processing should generate such the serializable schedule of less data/resource contention.

3 Approach

3.1 scheduling strategy

In the formal theory of scheduling, our problem is a dynamic scheduling of general job-shop with serializability constraint. The static general job-shop scheduling problem is known to contain NP-hard class [Gra79]. Hence a strategic scheduling must be used. Our strategy is to make a scheduler predict the serializable order (SR-order) of active transactions s.t. less data/resource contentions will occur. Then the scheduler grants the lock-request which obeys such the 'desirable' SR-order, while delaying the lock-requests to conflict with the order.

For the prediction, we use a transaction precedence graph with the weight on each edge. The weight represents the remaining cost to be executed in transactions. The 'Weighted' Transaction Precedence Graph (WTPG) is defined as follows:

Definition 1 Weighted Transaction Precedence Graph is a transaction precedence graph $< N, E, C, w >$ such that: (T0 is the initial transaction. Tf is the final transaction. Ti,Tj are the other general transactions.)

1. N is the set of nodes. A node represents a transaction.

2. E is the set of directed edges Ti $\rightarrow$ Tj. It says "Ti is determined to precede Tj in the serializable order". For each transaction Ti, T0 precedes Ti and Ti precedes Tf.

3. C is the set of *choice edges.* A *choice edge* (Ti,Tj) is a pair of the directed edges Ti$\rightarrow$Tj and Tj$\rightarrow$Ti. It says 'Ti conflicts with Tj'. If Ti is determined to precede Tj, the edge Tj$\rightarrow$Ti disappears and the edge Ti$\rightarrow$Tj is regarded as a member of E. This operation is named *resolution of choice edge.*

4. w is the weight function of all the directed edges in $E \cup C$. The edge $Ti \rightarrow Tj$ in E or the directed edges $Ti \rightarrow Tj$ and $Tj \rightarrow Ti$ in a choice-edge (Ti, Tj) in C have the weights $w(Ti \rightarrow Tj)$ (or $w(Tj \rightarrow Ti)$) defined below:

- $w(T0 \rightarrow Ti)$ is the rest of the time from the current to Ti's commitment, assuming that the remaining steps of Ti are executed without any blocking by any other transactions. It is the earliest possible commitment time (or the ready time) of Ti.

- $w(Ti \rightarrow Tf)$ is the cost from Ti's commitment to its completion.
- $w(Ti \rightarrow Tj)$ is the cost of the remaining steps Tj must execute until its commitment after Ti's commitment.

□

Note that the edge (Ti, Tj) and $Ti \rightarrow Tj$ exist only when Ti and Tj have the conflicted access to a data d. Since all the locks are held until the commitment, Tj must start the conflicted step after Ti commits and releases the lock on d.

Example: Fig.3 is the WTPG for the situation in Fig.2-a at clock 0 after `r1(D)` has started. (For simplicity, T1 is omitted.) The edge T0 → T2 has the weight 8 as T2's earliest possible commitment time: T2 can commit at clock 8 at earliest, because T2's step `r2(E)` on DM1 can start at clock 4 after `r1(D)` ends. The edge T2 → T3 has the weight 2 for the cost of `w3(A)` and `w3(C)`: T3 must run these steps before its commitment after T2 commits and releases the X-lock on the conflicting data A. By the assumption on the cost model, the edge $Ti \rightarrow Tf$ has the weight 0.[]

The SR-order of all the transactions in a given WTPG resolutes all the choice-edges in the graph. The order is named a *full* SR-order. The WTPG resoluted by a full SR-order is a directed acyclic graph with the weighted edges.

As the 'desirable' SR-order, We use the full SR-order to make the shortest critical path from T0 to Tf in the WTPG. In Fig.3, the SR-order W making the shortest critical path is { T3 → T4, T3 → T5 }. By enforcing the SR-order W, T3's step `r3(C)` and `w3(A)` are granted in priori to `r2(A)` and `w4(C)`.

Give a full SR-order, then the length of the critical path in the WTPG resoluted by the SR-order is the earliest possible completion time of the total schedule under the SR-order. By enforcing the full SR-order to make the shortest critical path, less data/resource contention is achieved. Because a chain of blocked transactions makes the long critical path and a congested disk makes the earliest possible commitment time of some transactions long.

Figure 3: example of WTPG

Figure 4: WTPG of Fig.2-a at clock 0

As shown in Fig.2-a, the C2PL-scheduler in section2.4 makes the full SR-order { T1, T2 → T3 → T4}. Fig.4-a is the WTPG resoluted by this order. Its critical path is $T0 \rightarrow T2 \rightarrow T3 \rightarrow T4$ of length 14. Fig.4-b shows the WTPG resoluted by the SR-order W of the shortest critical path. The path is $\{T0 \rightarrow T3 \rightarrow T2\}$ of length 8.

3.2 scheduler using WTPG

When a new transaction T arrives, its choice-edges and their weights must be built up in the WTPG. For the purpose, T must declare all its read/write dataset at its start. Moreover each lock-declaration by T must include the 'duetime' of the lock: The 'duetime' of the lock is the cost of T's remaining steps to be executed after holding the lock until its commitment. By the duetime, the weights on the choice-edges are computed locally in each disk module.

As described in Section2, the Concurrency Controller (CC) using WTPG is invoked when DMi gets idle and runs as follows:

Step0. invoked when DMi gets idle.

Step1. determine the full SR-order W of the active transactions, s.t. W makes the shortest critical path in the current WTPG.

Step2. find a step q in RQ[DMi] such that q is not blocked by the lock held on d and the precedence constraint caused by granting q does not conflict with W. (RQ[DMi] is that in section2. d is the data accessed by q.)

Step3. If q is found, grant the lock of q, run q on DMi, and update all the weights on the edge $T0 \rightarrow Ti$.

Step4. If q is not found, DMi sleeps in the unit of time. After the sleeping, DMi invokes CC again. []

Note that q holds the lock on the accessed data if it is granted in the above *Step3.* All

Figure 5: chain WTPG

the locks are released at the commitment.

Example: Fig.2-b is the schedule generated by the above scheduler. In Fig.2-b at clock 0 after r1(D) starts, the SR-order of the shortest critical path is $W = \{T3 \rightarrow T4, T3 \rightarrow T5\}$ as shown in Fig.4-b. The scheduler delays r2(A) for DM2 at clock0 because the constraint $\{T2 \rightarrow T3\}$ caused by the step conflicts with W. Instead of r2(A), r3(C) is granted because the constraint {T3 → T4 } caused by the step obeys W. At clock1, the ready time of T3 is updated as shown in Fig.4-c. The SR-order of the shortest critical path is still W at clock1. Thus r2(A) is delayed again and w3(A) is granted for DM2.[]

The CC is implemented in the same way as that in C2PL scheduler in section2.4. In the above *Step2*, the blocking test is done in each DM and the remaining part is test in the centerized CC.

4 Scheduler using chain WTPG

4.1 to find the shortest critical path in chain WTPG

Our scheduler must find the full SR-order which makes the shortest critical path in a given WTPG. (In the rest of the paper, "path" refers to the path from T0 to Tf. In depicting WTPG, Tf is omitted.) The problem is referred to as 'the Shortest Critical Path finding problem' (SCP) in this paper. SCP for a given WTPG is NP-hard, as shown in the appendix. This section restricts the form of WTPG to a set of chains and presents the $O(n^2)$ time algorithm for SCP. 'chain WTPG' means that all the edges except $T0 \rightarrow Ti$ and $Ti \rightarrow Tf$ constructs a set of chains:

Definition 2 Chain WTPG is the WTPG where all the transactions except T0 and Tf are labelled in the linear order { 1, 2, ..., N } and the transaction of label k conflicts at most with the two adjacent ones of label $(k-1)$ and $(k+1)$. []

Fig.5 is a chain WTPG with node $n0, n1, .., nN$. n0 is the initial transaction T0. The other nodes are general transactions. (Tf is omitted.)

As the notations, $n(k)$ refers to the node of label k. $G(1,N)$ refers to the graph in Fig.5. $G(i,j)$ $(1 \leq i \leq j \leq N)$ refers to the subgraph of $G(1,N)$, where $G(i,j)$ is composed of the node $n0$ and $\{n(i), n(i+1), .., n(j)\}$. $n(i) \rightarrow n(j)$ is the directed edge from $n(i)$ to $n(j)$ in the graph, and says "$n(i)$ precedes $n(j)$ in the serializable order".

Then, to the given $G(1,N)$, we compute the following pair of the parameter $L(k)$ and $R(k)$ on the choice-edge $(n(k-1), n(k))$ from $k = N$ to $k = 2$. They represent the shortest critical path information in $G(k-1,N)$. In the following, we say "the choice edge $(n(i), n(i+1))$ is set upwards" when the edge is resoluted to $n(i+1) \rightarrow n(i)$, and "set downwards" when it is resoluted to $n(i) \rightarrow n(i+1)$.

Definition 3 The parameter $L(k)$ and $R(k)$ are the triplets (curr, crit, rev). They are defined on the edge $n(k-1) \rightarrow n(k)$ and $n(k) \rightarrow n(k-1)$ respectively as follows;

In $G(k-1,N)$ where the choice-edge $(n(k-1), n(k))$ is set downwards, let $S1(k-1,N)$ be the SR-order of $\{ n(k-1), n(k), ..., nN \}$ which makes the shortest critical path in $G(k-1,N)$. Then $L(k)$ is defined on the edge $n(k-1) \rightarrow n(k)$ as follows:

- $L(k).crit$: the length of the shortest critical path in $G(k-1,N)$ under $S1(k-1,N)$.
- $L(k).rev$: the smallest label among the labels i such that the choice-edge $(n(i), n(i+1))$ is set upwards under $S1(k-1,N)$. If it does not exist, $L(k).rev$ is set to N.
- $L(k).curr$: the length of the path $n0 \rightarrow n(k-1) \rightarrow n(k) \rightarrow .. \rightarrow n(L(k).rev)$.

In $G(k-1,N)$ where the choice-edge $(n(k-1), n(k))$ is set upwards, let $S2(k-1,N)$ be the SR-order of $\{ n(k-1), n(k), ..., nN \}$ which makes the shortest critical path in $G(k-1,N)$. Then $R(k)$ is defined on the edge $n(k) \rightarrow n(k-1)$ as follows:

- $R(k).crit$: the length of the shortest critical path in $G(k-1,N)$ under $S2(k-1,N)$.
- $R(k).rev$: the smallest label among the labels i such that $(n(i), n(i+1))$ is set downwards under $S2(k-1,N)$. If the node does not exist, $R(k).rev$ is set to N.
- $R(k).curr$: the length of the critical path from $n0$ to $n(k-1)$ in $G(k-1, R(k).rev)$ under $S2(k-1,N)$.

Figure 6: example of L(k) and R(k)

[]

Note that, in the above definition, all the choice edges in $G(k-1, L(k).rev)$ under $S1(k-1, N)$ are set downwards by the definition of $L(k).rev$. Under $S2(k-1, N)$, all the choice edges in $G(k-1, R(k).rev)$ are set upwards by the definition of $R(k).rev$. In the rest of the paper, $L(k)$ and $R(k)$ are notated by the triplet $(curr, crit, rev)$ and rev is notated by $n(rev)$.

Example1: Fig.6-a illustrates a chain-WTPG $G(2,4)$ composed of three nodes $n2, n3, n4$ with $n0$, and $L(k), R(k)$ for $k = 3, 4$. Apparently $L(4)$ is (6,6,n4) and $R(4)$ is (11,11,n4). For computing $L(3)$ in $G(2,4)$, the edge $(n2, n3)$ is set downwards $n2 \rightarrow n3$ at first. Fig.6-b shows this case. The possible SR-order under the constraint $\{n2 \rightarrow n3\}$ is { $n3 \rightarrow n4$ } or { $n4 \rightarrow n3$ }. Between the two cases, { $n3 \rightarrow n4$ } makes the shortest critical path $n0 \rightarrow n2 \rightarrow n3 \rightarrow n4$ of length 8 in $G(2,4)$. (the critical path by $\{n4 \rightarrow n3\}$ is $n0 \rightarrow n4 \rightarrow n3$ of length 11.) Hence $L(3)$ is (8,8,n4). Fig.6-c shows the case for $R(3)$. Give the constraint $\{n3 \rightarrow n2\}$, then the SR-order $S2(2,4)$ of the shortest critical path is $\{n3 \rightarrow n2, n3 \rightarrow n4\}$, not $\{n4 \rightarrow n3 \rightarrow n2\}$. Consequently $R(3)$ is (5,6,n3).[]

Theorem 1 Suppose that $L(i)$ and $R(i)$ for all $i = k+1$ to N are given in $G(k, N)$. Then the SR-order $S1(k, N)$, $S2(k, N)$ in the definition3 and the SR-order $S(k, N)$ which makes the shortest critical path P in $G(k, N)$ are computed by the formulae:

$S1(k, N) = \{k \rightarrow (k+1) \rightarrow .. \rightarrow L(k+1).rev\} \cup S2(L(k+1).rev, N)$

$S2(k, N) = \{k \leftarrow (k+1) \leftarrow .. \leftarrow R(k+1).rev\} \cup S1(R(k+1).rev, N)$

such that $S1(k, k) = S2(k, k) = \phi$ for all k.

The length of $P = min(L(k+1).crit, R(k+1).crit)$.

$S(k, N) = S1(k, N)$; if $L(k+1).crit \leq R(k+1).crit$

$S(k, N) = S2(k, N)$; otherwise

[]

Proof: see the appendix.[]

Figure 7: the algorithm for $L(k)$

Example2: In the Example1, $L(3).crit = 8 > R(3).crit = 6$. Hence $S(2,4) = S2(2,4) = \{n2 \leftarrow n3\} \cup S1(3,4) = \{n2 \leftarrow n3 \rightarrow n4\} \cup S2(4,4) = \{n2 \leftarrow n3 \rightarrow n4\}$. []

By computing $L(2)$ and $R(2)$ in polynomial time, the SR-order $S(1,N)$ to make the shortest critical path in $G(1,N)$ is found in polynomial time. The algorithm for $L(k)$ and its proof are described here. The algorithm for $R(k)$ and its proof are presented in the appendix.

Theorem 2 Suppose that $G(k-1,N)$ and all the parameter $L(i)$ and $R(i)$ s.t. $i = k+1$ to N are given. Then the algorithm in Fig.7 and that in Fig.12 in the appendix compute $L(k)$ and $R(k)$ in $O(N-k)$ respectively.[]

For the proof of the theorem 2, the following lemma is used (Its proof is given in the appendix.)

lemma 1 Suppose that, in $G(k-1,N)$, $G(k-1,h)$ has set all its choice-edges downwards and the edge $(n(h),n(h+1))$ is set upwards $(k-1 \leq h \leq N)$. Then the SR-order which sets all the choice-edges in $G(h,N)$ by $S2(h,N)$ has the shortest critical path P in $G(k-1,N)$. And the length of P is $max(V(h),R(h+1).crit)$, s.t. $V(h)$ is the length of the critical path in the above $G(k-1,h)$.[]

Proof of theorem 2: In the algorithm for $L(k)$, $L(k)$ is the minimum of $L1(k)$ and $L2(k)$. $L1(k)$ is the value of $L(k)$ when the edge $(n(k),n(k+1))$ is set downwards. $L2(k)$ is that when the edge is set upwards.

In the procedure for $L2(k)$, it is enough to assume that $G(k,N)$ has set all its choice-edges by $S2(k,N)$ by lemma1. By adding $n(k-1)$ and the edge $n(k-1) \rightarrow n(k)$ to $G(k,N)$, the new generated path in $G(k-1,N)$ is $n0 \rightarrow n(k-1) \rightarrow n(k)$. It is compared with the shortest critical path in $G(k,N)$ under $S2(k,N)$, whose length is $R(k+1).crit$. Thus the procedure computes $L2(k)$ correctly.

Figure 8: $G(k-1,N)$ in computing $L1(k)$

In the procedure for $L1(k)$, $G(k,N)$ has the shortest critical path under $S1(k,N)$. By adding $n(k-1)$ and the edge $n(k-1) \rightarrow n(k)$ to $G(k,N)$ under $S1(k,N)$, the new generated path $P0$ is $n0 \rightarrow n(k-1) \rightarrow n(k) \rightarrow n(k+1) \rightarrow ... \rightarrow n(L(k+1).rev)$. Fig.8-a shows $G(k-1,N)$ in this case. All its choice-edges are set by the SR-order $S3 = S1(k,N) \cup \{n(k-1) \rightarrow n(k)\}$. The variable temp in the algorithm is the length of $P0$. If $P0$ is shorter than the shortest critical path in $G(k,N)$, $S3$ makes the shortest critical path in $G(k-1,N)$. (Because if not, when the edge $(n(k-1),n(k))$ and $(n(k),n(k+1))$ are set downwards, $G(k-1,N)$ has the critical path shorter than that in $G(k,N)$. Thus $G(k,N)$ has the shorter critical path than its shortest critical path. It is the contradiction.)

If $P0$ is longer than the shortest critical path in $G(k,N)$, $P0$ may become the new critical path in $G(k-1,N)$. However, $P0$ may get shortened by setting upwards an edge in $P0$. Let the edge be $(n(h),\ n(h+1))$ and let $P(h)$ be the path $n0 \rightarrow n(k-1) \rightarrow n(k) \rightarrow n(k+1) \rightarrow ... \rightarrow n(h)$. Fig.8-b illustrates $G(k-1,N)$ where all the choice-edges in $G(k-1,h)$ are set downwards and the edge $(n(h),n(h+1))$ is set upwards. h must only vary from $k+1$ to $L(k+1).rev$. Because if h gets greater than $L(k+1).rev$, $P(h)$ gets longer than the original $P0 = P(h)$ s.t. $h = L(k+1).rev$. By the lemma1, the expression (1) in Fig.7 finds the shortest critical path for $L1(k)$ in this case. $V(h)$ in lemma1 is computed by the previous value $V(h-1)$ in $O(1)$ by the formulae as shown in Fig.7. Since the expression (1) includes the original new critical path $P0$, L1(k) is correctly computed.

Hence the algorithm in Fig.7 computes L(k) correctly. As for the complexity, the expression (1) in the procedure L1(k) in Fig.7 runs the loop at most $N-k$ times. $V(h)$ can be also computed in $O(1)$ in each loop. Thus $L(k)$ takes $O(N-k)$. []

corollary 1 SCP for a given chain-WTPG is solved in $O(n^2)$. s.t. n is the number of nodes in the WTPG.

Proof: By computing $L(k)$ and $R(k)$ from $k = N$ to 2, L(2) and R(2) are computed. This procedure must call the algorithm for $L(k)$ in Fig.7 and $R(k)$ in Fig.12 at $(N-1)$ times.

Figure 9: example of the algorithm for $L(k)$

By the theorem 2, the whole procedure takes $O(N^2)$. []

Example3: Fig.9-a shows the chain WTPG that adds the node n1 and the edge $n1 \rightarrow n2$ to the graph in Fig.6-a. Fig.9-b shows the L2(2), s.t. the edge (n2,n3) is set upwards. By adding $n1$, the new generated path P in Fig.9-b is $n0 \rightarrow n1 \rightarrow n2$ of length 15, greater than the old critical path $R(3).crit = 6$. Thus P becomes the new critical path and L2(2) is (15,15,n2). Fig.9-c shows the case for computing $L1(2)$. When adding $n1 \rightarrow n2$ to $G(2,4)$ under $S1(2,4) = \{n2 \rightarrow n3 \rightarrow n4\}$, the new path P is $n0 \rightarrow n1 \rightarrow n2 \rightarrow n3 \rightarrow n4$ of length $(15 + L(3).curr - r2) = 20$, greater than $L(3).crit = 8$. But P can be shortened by setting the edge (n3,n4) upwards as in Fig.9-d. In Fig.9-d, the critical path has the length $max(n0 \rightarrow n1 \rightarrow n2 \rightarrow n3, R(4).crit) = |n0 \rightarrow n1 \rightarrow n2 \rightarrow n3| = 16$. Hence L1(2) is (16,16,n3). Since $L1(2).crit = 16$ is greater than $L2(2).crit = 15$, $L(2)$ is set to L2(2) = (15,15,n2). By the theorem 1, the SR-order $S(1,4)$ to make the shortest critical path in the chain-WTPG is: $S(1,4) = S1(1,4) = \{n1 \rightarrow n2\} \cup S2(2,4) = \{n1 \rightarrow n2 \leftarrow n3 \rightarrow n4\}$. []

4.2 implementation issues

This subsection describes some implementation issues for the scheduler using the chain-WTPG.

1. At the start of the transaction T, the scheduler using the chain-WTPG must test if the new WTPG by adding T is still a set of chains. It is computed by the depth-first traversal in the WTPG. If it fails, T is delayed until one active transaction commits or aborts.

2. When invoked, the concurrency controller (CC) does not compute the SR-order to make the shortest critical path every time. CC uses the previously computed SR-order if neither of these conditions is satisfied.

a) the specified period P has elapsed after the previous computation, or

b) a transaction commits/ aborts/ starts. or

c) a disk module starts a step which takes the larger lifetime than P.

The last condition is necessary because the start of a non-interruptable step may delay the commitment of another transaction significantly.

3. As the schedule proceeds, the ready time in the WTPG must be adjusted. Let $r(i,t)$ be the ready time of Ti at the clock t. $r(i,t)$ should be updated when the clock proceeds from t to $t+1$, or a new step starts at the clock t. For the purpose, each transaction T has the list $due(T) = [d_0, d_1, .., d_n]$ for all the disk modules DM0 to DMn. d_j is defined below.

Let *step(DMj)* be the first step among the remaining steps T must execute on DMj. Then, $d_j = costof(step(DMj)) + dueof(step(DMj))$, s.t. $costof(S)$ is the remaining cost of the step S itself, and $dueof(S)$ is the cost after ending the step S until T's commitment. When a step of T starts or it is currently running, $due(T)$ is updated. Let l_j be the remaining cost of the step ST, s.t. ST is running at clock t on DMj. $load = [l0, l1, ..., ln]$ represents the load status of all the disk modules at clock t. Then $r(i,t)$ is $max(X(k))$ for all $k = 0$ to n such that $X(k) = d_k$ if ST is the step of Ti and $X(k) = l_k + d_k$ otherwise.

$r(i,t)$ should be updated locally in each disk module and informed of to the centerized concurrency controller (CC).

For the purpose, each transaction issues the above due-information d_j to the disk module DMj and updates it if necessary. Suppose that all the ready time $r(i,t)$ has been computed at the start of the clock t. Then $r(i,t)$ is updated as follows:

Step0: If T_m's step of cost C starts on DM_j at clock t, $r(m,t)$ is not updated. For all Ti such that DMj has Ti's due-information d_j now, DMj updates $r(i,t)$ $(i \neq m)$ by $r(i,t) := max(r(i,t), d_j + C)$. If a unit of idle time is inserted to DMi, this formulae is applied s.t. C is the specified idle time. After the update, DMj informs CC of the new $r(i,t)$.

Step1: At the next clock $t+1$, $r(i,t+1)$ is that by decrementing $r(i,t)$.[]

Note that the above *Step0* is computed in each disk module locally.

5 Simulation and Results

5.1 simulation model

The simulation model is shown in Fig.10. It makes the Gantt chart for the input sequence of transactions as follows:

The *Transaction Generater* generates transactions at the arrival rate λ and enqueues it to Start Queue (SQ). Let the *active* transaction be that in Ready Queue (RQ), Block Queue (BQ), or Access Queue (AQ). A new transaction is enqueued into RQ from SQ if the number of active transactions is smaller than the multi programming level *mpl*. RQ represents all the ready queues RQ[DMi] in section2. If a transaction T is blocked, it goes into BQ until the lock is released. If granted, T goes to AQi for DMi which stores a partition for T. DMi executes the step of the transaction in AQi. When ending the step, T releases all its locks and completes if T commits. Otherwise T goes into RQ again.

Numdisk is the number of the disk modules. *Numdata* is the number of the partitions. The partition i is located on the disk module *(i modulo Numdisk)*. The size of a partition and the cost of a read/write step are given in each experiment. As the unit of size for a partition, we don't define a specified value but use the *unit* itself. One unit of size is large enough to ignore any other overhead except the data-access on the disk modules. One clock in the simulation is the time to access the data of a unit size on a disk module.

The protocols in the simulation are: Cautious Two Phase Lock (C2PL) [Nis87], Atomic Static Lock (ASL) [Tay85], Optimistic Lock (OPT) [Kun81] and chain-WTPG (WTPG).

In the chain-WTPG scheduler, the test for chain-form is executed when a new transaction goes into RQ. If the test fails, it goes into Topology Queue (TQ). When an active transaction commits or aborts, all the transactions in TQ go into SQ again. C2PL is used instead of dynamic two phase lock (D2PL) because D2PL suffers from deadlock and abortion. Since WTPG is deadlock-free, it is not fair to compare D2PL with it. In ASL, a transaction T requests all its locks at the start. If it fails, T is enqueued into SQ again. OPT is implemented according to [Kun81]. If the validation fails, the transaction is aborted and enqueued into SQ again. Note that C2PL, WTPG, and ASL are

Figure 10: simulation model

all deadlock-free. As the upper bound of the performance, we use the protocol *NONE*. It grants any lock-request anytime.

5.2 experiments

As performance metrics, we use the mean number of the committed transaction per clock (the throughput) and the mean available disk utilization. They are measured under *mpl* = *infinite* with varying the arrival rate λ. The saturation point is the value of λ such that the throughput is 90 % of λ. The simulations are run in 1000 clocks.

Numdisk=8 and *Numdata*=24 are used. (One relation is represented by eight partitions.) These values cause the high data/ resource contentions. They are much smaller than those used in the short term transaction (STT) processing [Car84, Ag85].

We adopt these small values because the size of BAT is assumed to increase as the size of database does. The 'size' refers to the amount of data accessed by the transaction. In the case where *Numdisk* is 8 and each read/write step is to access one partition, one range-selection of a BAT to a relation is assumed to hit the 1/8 region of the relation. Even if a database machine has 256 disk modules and each relation is divided into 256 partitions, the range-selection of a BAT is to run on the $256 \times 1/8 = 32$ disk modules in parallel and the other 224 disk modules are free for the other BATs. Consequently the effective values of *Numdisk* and *Numdata* are the same as those we used here. On the other hand, the size of STT does not increase as the database does. Hence the data/resource contentions are very low in the STT processing, while both the contentions are still very high in the BAT processing.

Since there are no benchmarks for the BAT processing, we use the three typical transaction patterns. The high data contention is classified as the chain of blocking and the long-duration lock on the 'hot spotted' relations. These three patterns have either of them. As the notations, each step is notated by <r(*read*)/w(*write*)> (*partitionID*, *cost of the step*). Each step has the range selection whose range is that of the accessed partition

on the partitioned attribute.

Experiment1: the read/write sequence: { r(F1:1)-r(F2:5)-w(F1:0.2)-w(F2:1) }. Its sequence of S/X-lock is { X(F1:1)- X(F2:5)- D(F2:1) }. D(F2:1) is the no-lock operation of a unit cost for the write step w(F2:1). The step for w(F1:0.2) is omitted. It models the transaction "to compute the join operation between the two partitions F1 and F2 and update them according to the joined result". F1 and F2 are randomly chosen among $Numdata = 24$ partitions of size *5 unit*. Both the write-step updates 10% size of the read data. The model suffers from the chain of blocking in X(F1) and X(F2).

The read/write sequence is determined as follows: Hybrid hash join in [DeW86] is assumed. Let A1 be the partitioned attribute. Each partition has a clustered index on another attribute A2 and a dense index on A1. r(F1) and r(F2) issue the range-selection on A1 whose range is that of the partition F1 and F2 respectively. Furthermore r(F1) has the range-selection on A2 which reads 20% of F1 by the clustered index on A2. Because of hybrid hash join, the smaller read step r(F1) is executed before r(F2).

Experiment2: The read/write sequence { r(B1:1) - r(B2:2) - r(B3:2) - w(F1:1) - w(F2:1) }. Its S/X lock sequence is the above sequence where r/w is replaced by S/X-lock. It models "to compute the join between the three partitions B1, B2 and B3, and update the other two partitions F1 and F2 according to the joined result".

B1, B2, and B3 are chosen randomly in the 8 partitions of size 2 unit representing one read-only relation. F1 and F2 are chosen randomly in the other $NumHot = 16$ partitions of size 1 unit, which represent the other two relations. The first step r(B1) has the range selection on the clustered attribute of B1 and reads 50% of it. Both the write step update 50% of the operand. In this case, the $NumHot = 16$ partitions become the hot spot. Thus the long-duration lock on them degrades the throughput.

Experiment3: The read/write sequence { r(B:4)-w(F1:1)-w(F2:4) }. Its lock sequence is the sequence s.t. r/w is replaced by S/X-lock. B is chosen randomly in the 8 partitions of size 4 *unit*, which represent a read-only relation. F1 and F2 are chosen in the other $NumHot = 16$ partitions of size 4 *unit* which represent the two hot-spotted relations. w(F1) accesses the 1/8 size of F1 by its clustered index. In this case, the last two

Figure 11: simulation results

Table 1: throughput (# committed transaction/clock) at the saturated points.

	NONE	ASL	C2PL	WTPG	OPT
Experiment1	1.01	0.81	0.39	0.80	0.29
Experiment2	1.06	0.66	0.89	0.90	0.69
Experiment3	0.82	0.46	0.40	0.63	0.40

update-steps are the access to the hot spotted relations and causes the chain of blocking at the same time.

5.3 results and discussions

The results in the experiments are given in Fig.11. Table1 shows their throughputs at the saturated points.

In Experiment1, Fig.11-a shows the throughput and Fig.11-b show the disk utilization. ASL and WTPG outperform OPT and C2PL in the throughput at the saturated points by about $(0.8/0.39 - 1) = 100\%$. ASL has the best throughput because it has no chain of blocking. WTPG has almost the same throughput as that in ASL. It implies WTPG avoids the thrashing by chains of blocking. C2PL has the low throughput by chains of blocking. OPT achieves almost the same throughput as that in C2PL.

Our result seems to conflict with the result in [Tay85, Ag85] that "in the high DC/ low RC case, OPT outperforms the blocking protocols", because the disk utilization at the saturated points is not so high in OPT and C2PL (about 30%) in Fig.11-b. It does not, however, conflict because the high abort ratio in OPT brings the higher non-available disk utilization as in Fig.11-b. Thus the resource contention in OPT is so high that the abortion is expensive. Consequently OPT has the low available disk utilization.

In Experiment2, the throughput is shown in Fig.11-c. C2PL and WTPG outperforms ASL by $(0.89/0.66 - 1) = 36\%$ and OPT by $(0.89/0.69 - 1) = 30\%$ in throughput. Since ASL has all its locks at the start, it must hold the locks on the hot spotted relations for a long time. It restricts the number of active transactions and degrades the throughput

significantly. C2PL and WTPG don't hold the locks until the locks are necessary.

Fig.11-d shows the throughput in Experiment3. The chain of blocking also occurrs in addition to the contention on the hot spot. Hence C2PL and ASL suffers from either of the high data contentions. WTPG avoids both of them. Consequently WTPG outperforms C2PL by $(0.63/0.4 - 1) = 58\%$ and ASL by $(0.63/0.46 - 1) = 37\%$.

Those results show the following: Under the high data/ resource contention in the BAT processing, WTPG has the low thrashing by both the chain of blocking and the contention on the hot spot, and has no abortion. The abortion protocol OPT has the low throughput because of the expensive abortion in the high resource contention. Under the high data contention, the blocking protocol C2PL suffers from the chains of blocking and the atomic lock does from the contention on the hot spots.

Moreover when λ increases over the saturated points, ASL and WTPG have the stable throughput while C2PL and OPT has the much lower throughputs than their peak values. Because C2PL and OPT have no guard to limit the increasing data contention without the multi programming level. On the other hand ASL and (chain-)WTPG has restricted the number of active transactions automatically. This stability helps to set the 'optimal' multi programming level.

Among the protocols except WTPG, ASL is the best alternative. It has no abortion, no blocking, but has the low number of active transactions. ASL is a subclass of WTPG by the topological constraint of 'the set of isolated points'. Intuitively the chain-form WTPG achieves twice as high number of active transactions as that in ASL. Because a chain composed of k nodes has $k/2$ isolated points.

6 Conclusion

This paper proposed a concurrency control scheme for Bulk Access Transactions (BAT): the transactions to access bulk of data on the database-disks. The target environment is the database machine composed of parallel disk modules with the range-partitioned file placement. The BAT processing has the high data/resource contention inherently, and the thrashing by them degrades the throughput of BAT significantly.

For the concurrent processing of many BATs, we proposed a cautious scheduler using the chain-form *Weighted Transaction Precedence Graph* (WTPG). The scheduler (WTPG-scheduler) predicts the serializable order making the shortest critical path in the WTPG. By enforcing this serializable order, it generates a schedule of less data/resource contention. We have also described the $O(n^2)$ algorithm for finding such the 'desirable' order.

In the WTPG-scheduler, every transaction must declare the duetime of its read/write step as well as its read/write dataset. The 'duetime' is the time to execute the transaction after holding the lock of the step until the commitment. By using these informations, the WTPG-scheduler has no abortion by deadlock, avoids the thrashing by chains of blocking, and need not hold the locks especially on the hot spots until necessary.

The simulation results show the WTPG-scheduler outperforms the others based on the two phase lock, the atomic lock, and the optimistic lock by 30% to 100% in some typical examples. The well known locking protocols do not work in the BAT processing: the optimistic lock has the low available disk utilization because of the expensive abortion in the high resource contention. The two phase lock suffers from the chain of blocking. The duration of the lock on the hot spots gets longer by high data/resource contention and degrades the throughput in the atomic lock.

As the future problems in the WTPG-scheduler, the transaction model should be extended to the parallel read/write steps. In this case it matters how precisely the cost of transactions is predicted and how much the performance depends on their precision. Moreover the recovery scheme should be built in the WTPG-scheduler.

In addition to them, we intend to study the BAT processing under the other file placement strategies. Particularly we plan to study the mixed processing of short term transactions and bulk access transactions on a database machine with various file placement strategies.

References

[Ab88] Abbott,R., et al, 'Scheduling Real time Transactions: a Performance Evaluation', In *Proc. 14th Int'l Conf. Very Large Data Bases*, 1-12, 1988.

[Ag85] Agrawal,R, et al, 'Models for Studying Concurrency Control Performance: Alternatives

and Implications', In *Proc. ACM-SIGMOD '85*, 108-121, 1985.
[Ban85] Bancilhon,F. et al, 'A Model of CAD Transactions', in *Proc. 11th Int'l Conf. Very Large Data Bases*, 1985.
[Bay86] Bayer,R. 'Consistency of Transactions and Random Batch', *ACM Trans. Database Systems*, vol.11 (No.4):397-404, 1986.
[Bh88] Bhide,A. 'An Analysis of Three Transaction Processing Architectures', In *Proc. 14th Int'l Conf. Very Large Data Bases*, 339-350, 1988.
[Car84] Carey,M.J. et al, 'The Performance of Concurrency Control Algorithms for Database Management Systems', In *Proc. 10th Int'l Conf. Very Large Data Bases*, 107-118, 1984.
[DeW86] DeWitt,D.J. et al. 'Gamma - High Performance Dataflow Database Machine', In *Proc. 12th Int. Conf. Very Large Data Bases*, 228-237, 1986.
[DeW87a] DeWitt,D.J. et al, 'A Single User Evaluation of the Gamma Database Machine', In *Proc. 5th Int. Workshop on Database Machines*, 43-59, 1987.
[DeW87b] DeWitt,D.J. et al, 'A Single User Performance Evaluation of the Teradata Database Machine', Tech. Rep., DB-081-87, MCC, 1987.
[Kat85] Katoh,N. et al, 'Cautious Transaction Schedulers with Admission Control', *ACM Trans. Database Systems*, Vol.10, No.2, 205-229, 1985.
[Kit86] Kitsuregawa, et al, 'Functional Disk System for Relational Database', In *Proc. Int'l Conf. Data Engineering*, 88-95, 1986.
[Kl85] Klahold,P. et al, 'A Transaction Model supporting complex applications in integrated information systems', In *Proc. ACM SIGMOD '85*, 388-401, 1985.
[Kun81] Kung,H. et al., 'On Optimistic Methods for Concurrency Control', *ACM Trans. Database Systems*, vol.6, No.2, June, 1981.
[Gra79] Graham,R.L. et al, 'Optimization and approximation in deterministic sequencing and scheduling: a survey', In *Annals of Discrete Mathematics 5*, 287-326, 1979.
[Gr78] Gray,J., 'Notes on Data Base Operating Systems', In *Operating Systems: An Advanced Course*, Springer-Verlag, 1978.
[Gr81] Gray,J., 'The Transaction Concept: Virtues and Limitations', In*Proc. 7th Int'l Conf. Very Large Data Bases*, 144-154, IEEE, 1981.
[Lyn83] Lynch,N. 'Multi Level Atomicity - A New Correctness Criterion for Database Concurrency Control', *ACM Trans. Database Systems*, vol.8, No.4, 484-502, 1983.
[Mo87] Molina,H.G. et al, 'SAGAS', In *Proc. ACM-SIGMOD '87*, 249-259, 1987.
[Nak88] Nakayama,M. et al, 'Hash Partitioned Join Method Using Dynamic Destaging Strategy', In *Proc. 14th Int'l Conf. Very Large Data Bases*, 468-478, 1988.
[Nis87] Nishio,S. et al. 'Performance Evaluation on Several Cautious Schedulers for Database Concurrency Control', In *Proc. 5th Int. Workshop on Database Machines*, 212-225, 1987.
[Tay85] Tay,Y.C., 'Locking Performance in Centralized Databases', *ACM Trans. Database Systems*, Vol.10, No.4, 415-462, 1985.

Figure 12: the algorithm for $R(k)$

Appendix

[proof of theorem 1] Suppose that $G(k,N)$ has the shortest critical path under a SR-order $S0$. By the definition of $L(k+1).rev$, under $S0$, $G(k, L(k+1).rev)$ has set all the choice-edges downwards and the edge $(n(L(k+1).rev), n(L(k+1).rev+1))$ is set upwards. Thus by lemma1, $S1(k,N)$ in the theorem is correct. $S2(k,N)$ is proved in the same way as $S1(k,N)$. By the definition of $S1(k,N)$ and $S2(k,N)$, $S(k,N)$ in the theorem is correct.[]

[proof of lemma1] Since the edge $(n(h), n(h+1))$ is set upwards, $G0 = G(k-1,h)$ is not connected with $G1 = G(h,N)$. Thus the critical path in $G(k-1,N)$ is either $V(h)$ in $G0$ or the critical path in $G1$. Among the critical path in $G1$, $S2(h,N)$ makes the shortest one. Hence the SR-order which sets all the edges in $G(k-1,h)$ downwards and sets all the edges in $G(h,N)$ by $S2(h,N)$ has the shortest critical path in $G(k,N)$. Its length is $max(V(h), R(h+1).crit)$ by the definition of $R(h+1).crit$. []

[The algorithm for $R(k)$ – see Fig.12]

[Proof of the algorithm for $R(k)$ in the theorem2]

$R1(k)$ is the value of $R(k)$ for the case the edge $(n(k), n(k+1))$ is set upwards. $R2(k)$ is that when it is set downwards. Then $R(k)$ is the minimum of $R1(k)$ and $R2(k)$.

In the procedure for $R2(k)$, the correctness is trivial. In the procedure for $R1(k)$, it is enough to assume that $G(k,N)$ has set all its edges by $S2(k,N)$ at first. By adding $n(k-1)$ and the edge $n(k) \rightarrow n(k-1)$, the two new paths are generated. One is the edge $n0 \rightarrow n(k-1)$. The other is the path P which is composed of the critical path $P2$ from $n0$ to $n(k)$ in $G(k, R(k+1).rev)$ and the edge $n(k) \rightarrow n(k-1)$. The path $P2$ is that of length $L(k+1).curr$ in $G(k,N)$.

If the edge $n0 \rightarrow n(k-1)$ becomes the new critical path in $G(k-1,N)$, it cannot get shortened. The part (3) in Fig.12 computes $R2(k)$ in this case.

Otherwise the new critical path P can get shortened by setting an edge in P downwards. The critical path when the edge $(n(h), n(h+1))$ in P is set downwards is $max(V(h), L(h+1).crit)$. Its proof is similar to that in lemma1 and omitted. $V(h)$ is the length of the critical path in $G(k-1,h)$, where all the choice-edge in it are set upwards. Thus the expression (2) in Fig.12 finds the shortest critical path in this case.

Consequently the algorithm computes $R(k)$ in $O(N-k)$. []

- Partition & its size

  A : 1 unit. C : 1 unit.
  D : 4 unit. E : 3 unit. F : 3 unit.

- Data placement

  E, D on DM1. F, A, C on DM2.

- Transaction Model

  T1: r1(D).
  T2: r2(A) → r2(E) → w2(A).
  T3: r3(C) → w3(A) → w3(C).
  T4: w4(C) → w4(F).

Figure 1: transaction model and data placement

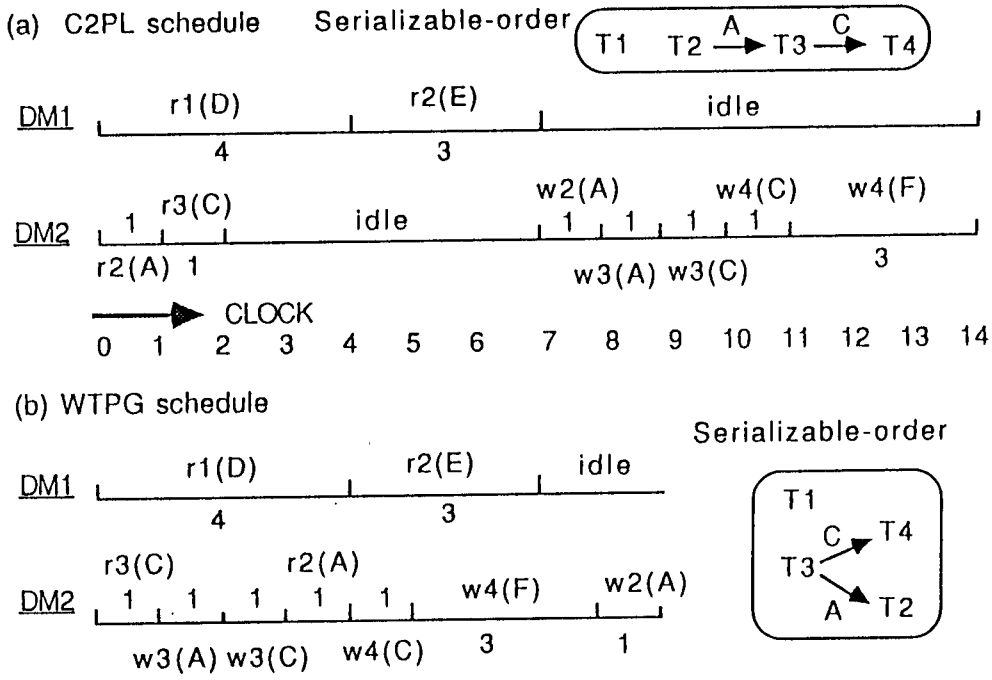


Figure2: Gantt chart of the transactions
in Fig.1 by a) C2PL b) WTPG

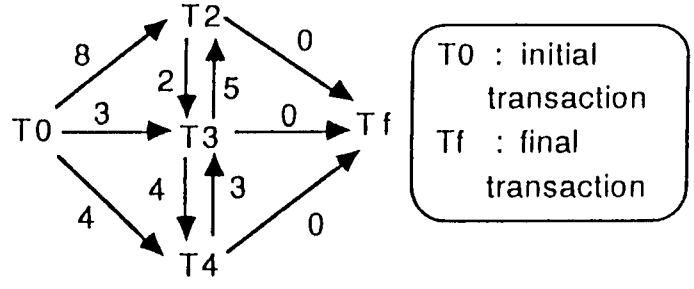


Figure3: example of WTPG

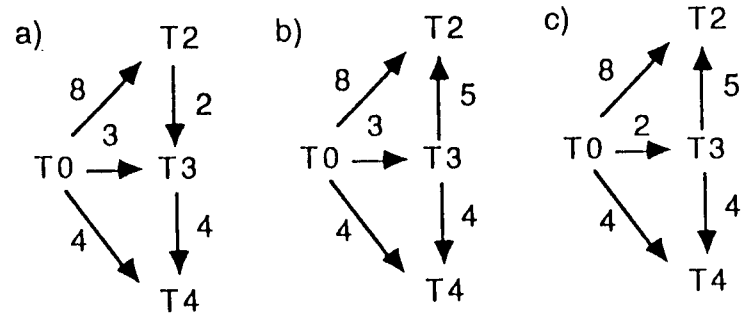


Figure4. WTPG resoluted by a) C2PL at clock 0
b) the shortest critical path at clock0
c) the shortest critical path at clock1

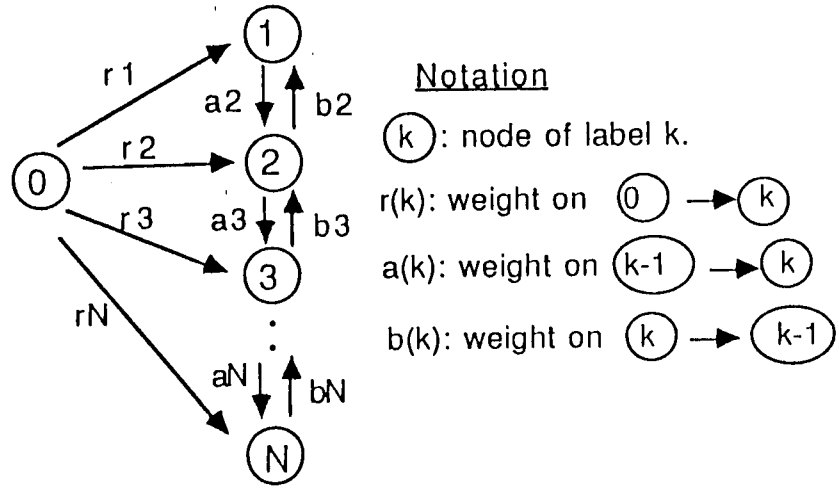
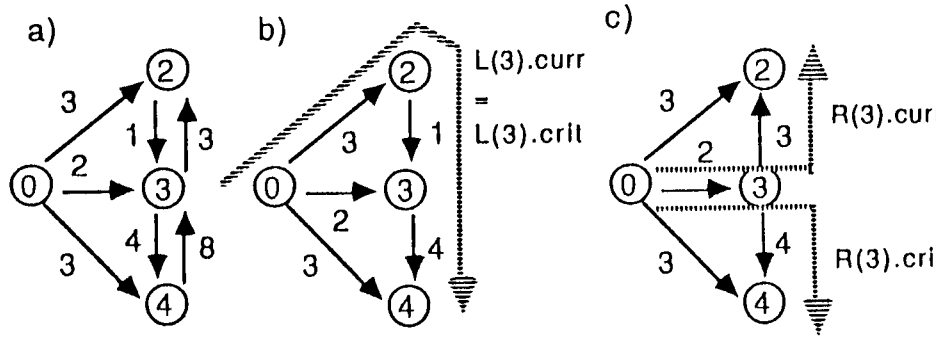


Figure5: chain WTPG G(1,N)



L(4) = (6,6,n4) L(3) = (8,8,n4) R(3) = (5,6,n3)
R(4) = (11,11,n4)

Figure.6: example of L(k) and R(k)

```
The algorithm for L(k)

Input: G(k,N), L(i), R(i) for all i = k+1 to N.
Output: L(k) = (crit, curr, rev)
Notation: n(k) : the node of label k.
        n(i)-n(j) : the directed edge from n(i) to n(j).
          a(k) : the weight on the edge n(k-1)-n(k).
          r(k) : the weight on the edge n0-n(k).

main()
{
        /* procedure L1(k):      */
        temp = L(k+1).curr - r(k) + r(k-1) + a(k);
        if ( temp <= L(k+1).crit ) then {
                L1(k).crit = L(k+1).crit;
                L1(k).curr = temp;
                L1(k).rev  = L(k+1).rev;
        }
        else    {       /* temp > L(k+1).crit */
                L1(k).crit = min( max( V(h), R(h+1).crit ) );   (1)
                                h
                             such that  h = k+1 to L(k+1).rev
        /*
         *      V(h) is the length of the critical path in G(k-1,h)
         *              where all the choice-edges are set downwards.
         *      V(h) = max( r(h), V(h-1)+a(h) ). (k =< h =< L(k+1).crit)
         *      V(k-1) = r(k-1).
         */

                L1(k).rev  = h0;   such that h=h0  takes the minimum in (1).
                L1(k).curr = the length of the path
                             n0-n(k-1)-n(k)-n(k+1)-,..- n( L1(k).rev);
        }
        endif   /* end of L1(k) */

        /*      procedure L2(k):*/
        L2(k).curr = r(k-1) + a(k);
        L2(k).crit = max( L2(k).curr, R(k+1).crit );
        L2(k).rev  = k;
        /*      end of L2(k)    */

        L(k).crit = min( L1(k).crit, L2(k).crit );
        if  (L1(k).crit <= L2(k).crit)  then {
                L(k).curr = L1(k).curr;
                L(k).rev  = L1(k).rev;
        }
        else {
                L(k).curr = L2(k).curr;
                L(k).rev  = L2(k).rev;
        }
        endif
}
```

Figure 7: the algorithm for $L(k)$

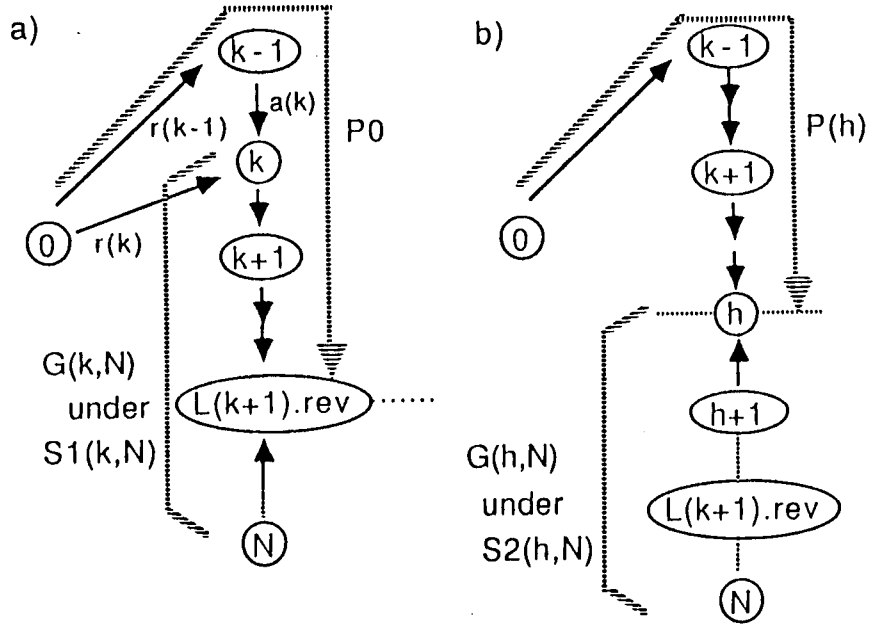


Figure 8: $G(k-1,N)$ in computing $L1(k)$

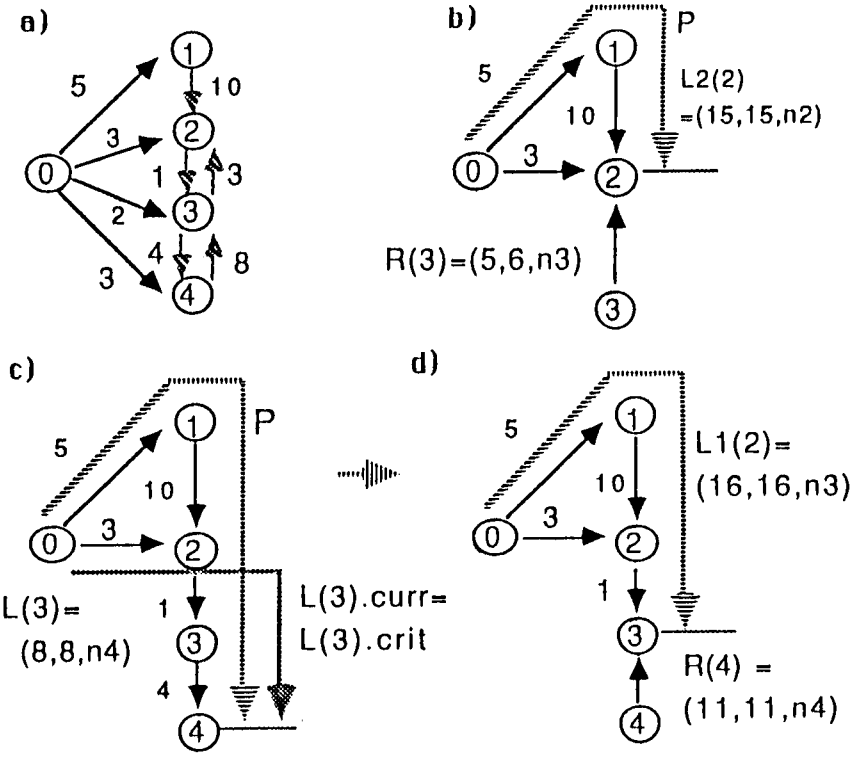


Figure9: example of the algorithm for L(k)
a) $G(1,4)$ b) L2(2) c),d) L1(2)

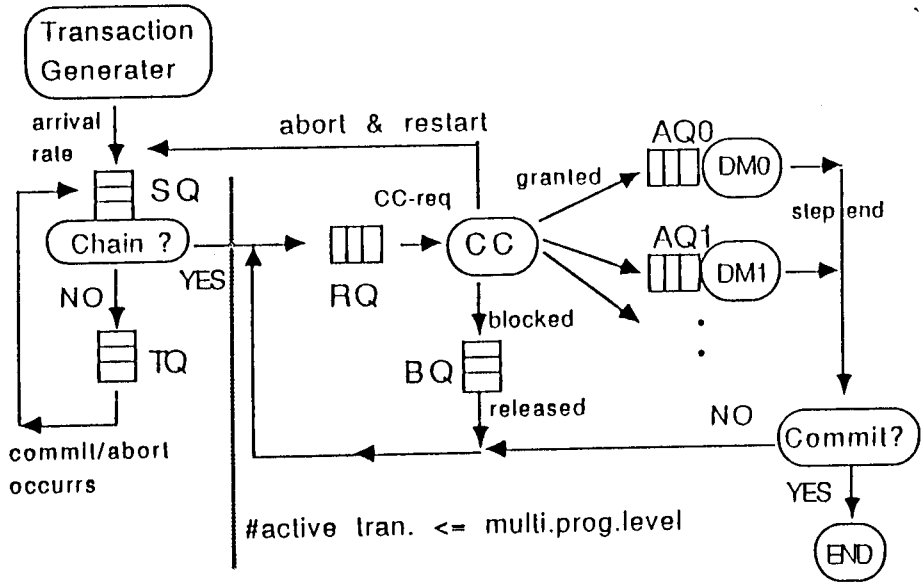


Figure10: the simulation model

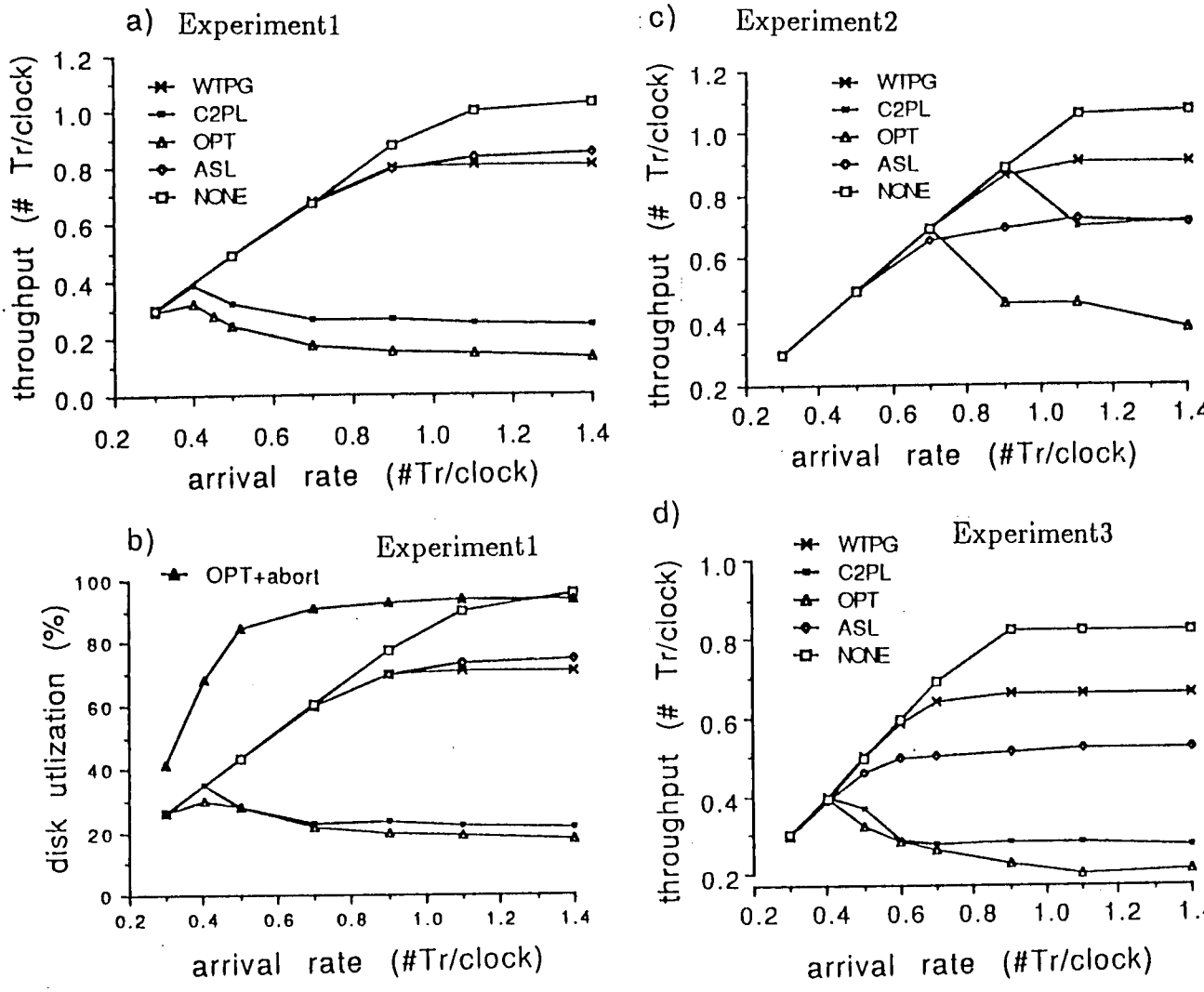


Figure11: simulation results
a), c), d) throughput in Experiment1, 2, 3
b) disk utilization in Experiment1

```
The algorithm for R(k)

Input: G(k,N), L(i), R(i) for all i = k+1 to N.
Output: R(k) = (crit, curr, rev)
Notation: n(k) : the node of label k.
          n(i)-n(j) : the directed edge from n(i) to n(j).
           b(k) : the weight on the edge n(k)-n(k-1).
           r(k) : the weight on the edge n0-n(k).

main()
{
        /*       procedure R1(k)        */
        temp = R(k+1).curr + b(k);
        if ( max( r(k-1), temp ) =< R(k+1).crit ) {
                R1(k).crit = R(k+1).crit;
                R1(k).curr = temp;
                R1(k).rev  = R(k+1).rev;
        }
        else if ( max( r(k-1), temp ) == r(k-1) ) { /* the part (3) */
                R1(k).crit = r(k-1);
                R1(k).curr = r(k-1);
                R1(k).rev  = R(k+1).rev;
        }
        else {  /* temp > R(k+1).crit */
                R1(k).crit = min( max( V(h), L(h+1).crit ) );   /*  (2) */
                                h
                        such that  h = k+1 to R(k+1).rev
                /*
                 *      V(h) :  the length of the critical path in G(k-1,h)
                 *              where all the choice-edges are set upwards.
                 *      V(h) = max( C(h), V(h-1) );  (k =< h =< R(k+1).crit)
                 *      V(k-1) = r(k-1);
                 *
                 *      C(h) :  the length of the path
                 *              n0-n(h)-n(h-1)-...-n(k)-n(k-1).
                 *      C(h) = C(h-1) - r(h-1) + r(h) + b(h);
                 *                                ( k =< h =< R(k+1).rev)
                 *      C(k-1) = r(k-1);
                 */

                R1(k).rev  = h0;  such that h=h0  takes the minimum in (2).
                R1(k).curr = C(h0);
        }
        endif   /*      end of R1(k)    */

        /*      procedure R2(k)         */
        R2(k).curr = max( r(k)+b(k), r(k-1) );
        R2(k).crit = max( R2(k).curr, L(k+1).crit );
        R2(k).rev  = k;
        /*      end of R2(k)    */

        R(k).crit = min( R1(k).crit, R2(k).crit );
        if  (R1(k).crit <= R2(k).crit)  then {
                R(k).curr = R1(k).curr;
                R(k).rev  = R1(k).rev;
        }
        else {
                R(k).curr = R2(k).curr;
                R(k).rev  = R2(k).rev;
        }
        endif
}
```

Figure 12: the algorithm for $R(k)$