Concurrency Control of Bulk Access Transactions on Shared Nothing Parallel Database Machines

Tadashi OHMORI Masaru KITSUREGAWA Hidehiko TANAKA

Department of Electrical Engineering, The University of Tokyo
7-3-1 Hongo, Bunkyo-Ku, Tokyo 113, Japan

ABSTRACT

This paper proposes new concurrency control schemes for *Bulk Access Transactions* (BAT) on shared nothing database machines. A BAT is a transaction accessing large bulk of data, such as a transaction updating a whole file. BATs are used for batch jobs in database services, and many BATs should be finished in a short time. Thus it is necessary to execute BATs concurrently on a database machine.

When scheduling BATs, the performance is degraded by very high contentions of both data and resources. Therefore our strategy for scheduling BATs is to reduce the contentions as much as possible. We propose a *Weighted Transaction Precedence Graph* (WTPG) and two schedulers using it. A WTPG is used to estimate the degree of the contentions in a serializable schedule. Using a WTPG, the proposed schedulers reduce the contentions by optimization. In our simulation, both the schedulers achieve from 1.2 to 1.8 times higher throughput than Atomic Static Lock and Two Phase Lock.

1 Introduction

In database services, a batch job is usually given as a *Bulk Access Transaction* (BAT). A BAT is a transaction accessing large bulk of data, such as a transaction scanning a large file. For instance, a BAT in a banking system reads history-files for statistic analysis, and then updates master-files according to this analysis. Today, database services have to run many BATs for updating files as well as analyzing data. Thus concurrency control of BATs is necessary to keep a consistent database. In this paper, we propose new concurrency control schemes for BATs on 'shared nothing' database machines.

A database service is usually divided into the on-line service and the off-line one. The on-line service mainly executes short term transactions like debit-credit transactions [1]. On the other hand, most of BATs are executed in the off-line service. Today, a long period of time is used for the on-line service, and so the off-line service needs to finish many BATs in a much shorter time. If BATs have only read-operations, they can be executed in the on-line service. But, in the off-line service, many BATs update files as well as read large files. In order to finish these BATs in a short time, they should run concurrently on a database machine. This paper discusses scheduling of these BATs ('BAT processing') in the off-line service.

A 'shared nothing' database machine consists of computer-nodes interconnected by a network [2] [3] [4]. In this architecture, a bulk data processing is executed on the nodes storing necessary data. Then, when executing a BAT on this machine, load is unbalanced among all the nodes. Therefore concurrent execution of BATs is necessary for balancing the load and achieving high performance.

A BAT is categorized as a Long Lived Transaction (LLT) in [5]. Previous studies [6] [7] [8] [9] [1] [2] have intensively discussed concurrency control of short term transactions. But concurrency control of BATs has not been well discussed so far. A feature in the processing of BATs is very high contentions of both data and resources. Data contention means the contention over logical access to a data-granule [9], and resource contention means congestion of resources.

Data contention is very high in the BAT processing because coarse granules of locking are used. For instance, a whole file is locked before starting to scan it. Then a BAT is often blocked by one after another BAT. This 'chain of blocking' reduces the number of active transactions and degrades the overall performance [8] [9]. Furthermore master-files are often updated, and so these 'hot' files prevent high concurrency. Resource contention is also high owing to a bulk-operation such as scanning a file. In addition, a bulk-operation is too expensive to abort. Therefore the BAT processing is quite different from the short term transaction processing, and schedulers for BATs should avoid chains of blocking without aborting transactions.

Our strategy for scheduling BATs is to reduce data/resource contentions in the output schedule. We propose a *Weighted Transaction Precedence Graph* (WTPG) for estimating the degree of the contentions. This graph represents the cost in a serializable schedule of transactions. We propose two schedulers using a WTPG: 'Chain-WTPG scheduler' and 'K-conflict WTPG scheduler'. For reducing the contentions in the output schedule, these schedulers use a global optimization and a local one respectively. In both the schedulers, a transaction must declare its sequence of read/write steps and their I/O demands at its start. This information is used to build up a WTPG. Our simulation tests the performance of these schedulers in the BAT processing. We test them also when transactions declare erroneous I/O demands.

CH2840-7/0000/0476$01.00 © 1990 IEEE

The rest of this paper is organized as follows: Section 2 describes model and assumptions. In Section 3, we propose the WTPG and the two schedulers. Section 4 discusses the simulation results. The paper is concluded in Section 5.

2 Model and Assumptions

2.1 Target environment

Our target environment for the BAT processing is a 'shared nothing' database machine. This architecture consists of computer-nodes interconnected by a network [2] [3] [4]. Each node is a computer with disks storing a part of database, and these nodes execute overall database processing in parallel. As for file placements on these nodes, we assume the placements reducing traffic of messages. These placements are: range partitioning [4], partial declustering [3], and the placement to cluster files of high affinity at a node [3].

When scheduling BATs, these placements cause unbalance of load among the nodes: The range partitioning divides a relation horizontally by ranges of a fixed attribute, and each partition is located at a node [4]. Then a range-query of a BAT hits partitions on a few nodes, and a bulk data processing is executed there. As the result, the load is unbalanced when executing a single BAT. In the partial declustering, a file is partitioned over a subset of all the nodes [3]. This placement also causes the unbalance of load when scanning a file. This is true when placing files of high affinity at a node.

If a BAT runs on these file placements, this unbalance of load degrades the overall performance greatly. Therefore concurrent execution of BATs is necessary for balancing the load.

On a shared nothing architecture, these placements are desirable both in the short term transaction processing and in the mixed transaction processing. This is because the overhead of messages are reduced. In the range-partitioning or the placement clustering files of affinity, each node stores most of the data accessed by a short term transaction. Consequently traffic of messages are reduced among the nodes. The partial declustering also reduces messages when sending data from nodes to nodes. This reduction alleviates overhead of processors, and so the performance of the short term transaction processing is greatly improved [2] [3].

For simplicity, the rest of the paper assumes each relation is range-partitioned on all nodes.

2.2 Transaction model

This subsection defines the model of a BAT.

A transaction T is modeled as a sequential execution of steps, such that each step reads or writes only one partition at a node. This assumes each step has a range-selection hitting one partition.

A read (or write) step to a partition must hold a shared(S) (or exclusive(X)) lock on it before executing the step. A X-lock conflicts with either a S-lock or a X-lock.

$T1$: `r1(A:1)` → `r1(B:3)` → `w1(A:1)`.
$T2$: `r2(C:1)` → `w2(A:1)`.
$T3$: `w3(C:1)` → `r3(D:3)`.

Figure 1: transaction model

A transaction T declares all the data to read and those to write at its start. These declared data are named 'lock-declarations'. A shared (or exclusive) lock-declaration on a locking-granule `d` represents 'a transaction will read (or write) `d` in the future'. A lock-declaration is replaced by a lock-request when T requests to hold this lock. All the locks are held until the commitment of T for recovery, and they are released at its commitment.

We use a partition as a locking-granule. A lock on a partition `d` expresses a predicate lock to the partitioned range of `d`. We do not use a record-level X-lock when updating a partition, because a BAT is supposed to update a major part of the partition.

A size of a partition is given by the number of *objects* in it. One *object* is a unit of data for bulk data processing. e.g. an object for scanning files is the fixed number of tracks in a disk, such as 50 tracks.

Then the cost model of a BAT is defined as follows:

The cost of a read/write-step s is given by $costof(s)$. This is the number of objects accessed by s. $costof(s)$ is the estimated I/O demand of the step. When s reads a% of data in a partition P, $costof(s)$ is set to $a|P|$. ($|P|$ is the size of P.) When updating a% of P, $costof(s)$ is set to $2a|P|$. This is because a bulk update-operation must read data before writing them. Notice that a step can access a part of a partition if indices can be used.

Bulk-updated data are written back to disks immediately, and so we ignore I/O demand from the commitment of a transaction to its completion.

We use a centralized concurrency control owing to the partition level locking. The centralized control node (CN) manages a lock table of partition granules. When a lock is granted to a step of a transaction T, CN sends T to the node storing this partition.

Example2.1: Figure 1 illustrates three transactions $T1$, $T2$, and $T3$. '*step1* → *step2*' means a sequential execution of the steps. In the rest of the paper, $\mathtt{r}_i(P:C)$ (or $\mathtt{w}_i(P:C)$) refers to a read (or write) step of cost C to a partition P by a transaction T_i. (C is the number of objects to be accessed) This figure assumes that 100% of a partition is read and a half of it is updated. The commitment step at the last of each read/write-sequence is not displayed in the rest of the paper. □

3 Schedulers for BAT

3.1 Weighted transaction graph

Our strategy for scheduling BATs is to reduce data/resource contentions in the output schedule. For estimating the degree of the contentions, we attach weights to edges of a

transaction precedence graph. These weights represent the costs for executing transactions.

Definition 1 Weighted Transaction Precedence Graph (WTPG) is a graph $< N, C, E, w >$ such that:

1. N: the set of nodes representing transactions. ($T0$: the initial transaction. Tf: the final transaction. Ti and Tj are general ones.)
2. C: the set of *conflicting-edges* (Ti, Tj). (Ti, Tj) is a pair of edges $Ti \rightarrow Tj$ and $Tj \rightarrow Ti$. (Ti, Tj) means "Ti conflicts with Tj in the serializable order". This edge is generated only when both Ti and Tj have issued conflicting lock-declarations on a locking-granule. When it has been determined that Ti precedes Tj in the serializable order, (Ti, Tj) is replaced by a *precedence-edge* $Ti \rightarrow Tj$. This operation is called 'resolving (Ti, Tj) into $Ti \rightarrow Tj$'.
3. E: the set of *precedence-edges* $Ti \rightarrow Tj$. $Ti \rightarrow Tj$ means "Ti precedes Tj in the serializable order". $Ti \rightarrow Tj$ is generated only by resolving (Ti, Tj). (for all Ti, there exist the edges $T0 \rightarrow Ti$ and $Ti \rightarrow Tf$).
4. w gives a weight $w(Ti \rightarrow Tj)$ for all $Ti \rightarrow Tj$ in $E \cup C$. This weight is the number of accessed objects:

- "$w(T0 \rightarrow Ti) = k$" means "At the present schedule, Ti must access k objects before Ti commits".
- "$w(Ti \rightarrow Tf) = k$" means "After Ti has committed, Ti must access k objects before Ti completes".
- "$w(Ti \rightarrow Tj) = k$" means "After Ti has committed, Tj must access k objects before Tj commits".

□

Example3.1: Figure 2-(a) shows the WTPG where all the transactions in Figure 1 have just started. In the rest of the paper, a WTPG is depicted as follows: A precedence-edge is depicted by a solid arrow. A conflicting-edge is represented by a pair of shaded arrows between nodes. A weight of an edge is depicted beside the edge. "$Tj \leftarrow Ti$" also expresses "$Ti \rightarrow Tj$". e.g. In Figure 2-(a), the conflicting-edge $(T2, T3)$ is a pair of edges $T2 \rightarrow T3$ of weight 4 and $T2 \leftarrow T3$ of weight 2.

In Figure 2-(a), the weights are set as follows:

$w(T1 \rightarrow T2)$ is set to 1: If $T1$ precedes $T2$, $T2$ can start `w2(A:1)` only after $T1$ has committed and has released the X-lock on `A`. Therefore, after $T1$ has committed, $T2$ must access 1 object before its commitment.

Since $T1$ has just started, $T1$ must access 5 objects before its commitment from now. Thus $w(T0 \rightarrow T1)$ is set to 5.

From the cost model in Section2, $w(Ti \rightarrow Tf)$ is set to 0 for all Ti. Tf and its edges are not depicted in the rest of the paper. □

In Definition 1, only $w(T0 \rightarrow Ti)$ depends on the present state of the scheduling. As a schedule proceeds, $w(T0 \rightarrow Ti)$ is adjusted as follows: When a step of Ti ends a bulk data processing of 1 object, Ti issues a message to the control node and decrements $w(T0 \rightarrow Ti)$.

Since the transaction model in Section 2.2 accesses objects sequentially, each weight in a WTPG represents the shortest possible time between two events in a schedule.

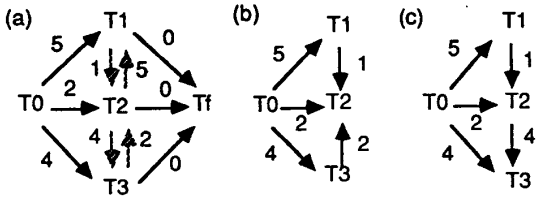

Figure 2: examples of WTPG

When a new transaction starts, new weights in a WTPG are set as follows:

At the start of a transaction Ti, Ti declares its sequence of steps $\{s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_N\}$ and their I/O demands $costof(s_i)$, as defined in Section2.2. Let due(s_i) be the number of objects as follows: 'due(s_i)=k' means "After the start of s_i, Ti must access k objects before its commitment". due(s_i) is defined by the formulae:

due(s_N) = $costof(s_N)$.

due(s_i) = $costof(s_i)$ + due(s_{i+1}). $(i < N)$

Then, at the start of Ti, $w(T0 \rightarrow Ti)$ is set as follows:

$w(T0 \rightarrow Ti)$ = due(s_0).

When a lock-declaration of a step s_i of Ti conflicts with that of a step s_j of Tj, the weights on (Ti, Tj) are set as follows:

$w(Tj \rightarrow Ti)$ = due(s_i).

$w(Ti \rightarrow Tj)$ = due(s_j).

$w(Ti \rightarrow Tj)$ and $w(Tj \rightarrow Ti)$ are set to the largest values among the values in the above formulae.

For all steps s_j of a declared transaction, due(s_j) is attached to the lock-declaration of s_j in the lock table. Thus all the weights on edges of Ti are computed when Ti starts.

3.2 Scheduler using global optimization

Using a WTPG, the degree of data/resource contentions in a schedule is estimated as follows:

Suppose that all the conflicting-edges of a WTPG have been resolved as specified in a serializable order S. e.g. when S=$\{T1 \rightarrow T2, T3 \rightarrow T2\}$ is given, the WTPG of Figure 2-(a) is changed into the WTPG of Figure 2-(b).

Then, in this WTPG, the length of its critical path from $T0$ to Tf is the earliest possible completion time of a total schedule. (S is the serializable order of this schedule.) The shorter this critical path is, the less contentions of both data and resources occur.

In the rest of the paper, a '*full* SR-order' refers to the serializable order resolving all the conflicting-edges of a WTPG. A 'WTPG resolved by a full SR-order S' is the WTPG whose conflicting-edges has been resolved as specified in S. A 'critical path' is the longest path from $T0$ to Tf in a WTPG.

Example 3.2: In Figure 2-(a), a full SR-order W = $\{T1 \rightarrow T2, T3 \rightarrow T2\}$ resolves all the conflicting-edges of the WTPG so that it has the shortest critical path. Figure 2-(b) shows the WTPG resolved by W. Its critical path is $T0 \rightarrow T1 \rightarrow T2$ of length 6.

When the WTPG of Figure 2-(a) is resolved by another full SR-order $\{T1 \rightarrow T2 \rightarrow T3\}$, its critical path has the length of 10 as shown in Figure 2-(c). Clearly a chain of blocking $\{T1 \rightarrow T2 \rightarrow T3\}$ occurs in Figure 2-(c). □
Then our scheduling strategy is:

> The serializable order of the output schedule should be the full SR-order W, where the WTPG resolved by W has the shortest critical path.

This strategy uses a global optimization for reducing the contentions in a schedule, because the future contentions must be predicted. The contentions are reduced by enforcing this globally optimized serializable order W to the output schedule.

It is NP-hard to compute the above W in any WTPG (see the theorem3 in the appendix). Thus we restrict a form of a WTPG into a 'chain-form WTPG'. In a given 'chain-form WTPG', W is computed by $O(N^2)$. (N is the number of nodes in the chain-form WTPG). This algorithm is described in the appendix.

Definition 2 A *chain-form WTPG* is the WTPG such that all the nodes except both $T0$ and Tf are labeled $1, \ldots, N$ as follows: for all k in $2, 3, \ldots, N-1$, $n[k]$ conflicts only with either $n[k-1]$ or $n[k+1]$ in the serializable order. ($n[k]$ is the node of label k.) □

Figure 3 shows a chain-form WTPG. ($T0$ is labeled 0. Tf and its edges are not depicted.) WTPGs in Figure 2 are also chain-form WTPGs.

Using the above scheduling strategy, the scheduler CC1 behaves as follows when a lock-request q is issued on a locking-granule d:

CC1(INPUT q: lock-request on d: data)
Step0: If q is the start step of a new transaction T, CC1 adds T into the WTPG and tests if it is still a chain-form WTPG. If this WTPG is not a chain-form, T is aborted.

Step1: If q conflicts with the lock holding d, q is blocked.

Step2: Compute a full SR-order W, such that the WTPG resolved by W has the shortest critical path.

Step3: If the schedule gets inconsistent with W by granting q, q is delayed. Otherwise q is granted.
□

The delayed lock-requests or the aborted ones are resubmitted to CC1 after a fixed delay. The *Step0* of CC1 keeps a WTPG in a chain-form. This 'chain-form' constraint is test by the depth first traverse. By the *Step3*, the output schedule is kept consistent with W. The CC1 is named the 'Chain-WTPG scheduler' or 'CHAIN'.

Example 3.3: In the WTPG of Figure 2-(a), $W = \{T1 \rightarrow T2, T3 \rightarrow T2\}$ makes the shortest critical path. Suppose that the step `r2(C:1)` of $T2$ in Figure 1 is submitted to CHAIN. If this step is granted, $(T2, T3)$ is resolved into $\{T2 \rightarrow T3\}$. This is inconsistent with W. Thus CHAIN delays `r2(C:1)`. □

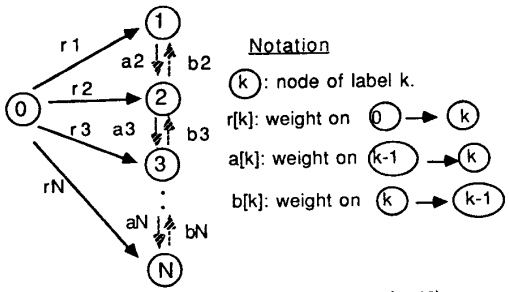

Figure 3: A chain-form WTPG $G(1, N)$

3.3 Scheduler using local optimization

When conflicting lock-requests are often issued to a part of database (namely the hot set), CHAIN cannot start many transactions owing to its chain-form constraint. This case also degrades the performance of BATs. In the BAT processing, master-files are very 'hot' files. Thus schedulers should accept any form of a WTPG without using a global optimization.

When a lock-request q has been just granted, the next function $\mathcal{E}(q)$ estimates the degree of the contentions in the present schedule.

$\mathcal{E}(q$: lock-request of a transaction $T)$
Step1: Make the WTPG where q has been just granted. Identify $before(T)$ and $after(T)$ in this WTPG. They are a set of transactions which precede T in the serializable order and a set of ones which T precedes respectively. If q causes a deadlock, return $\mathcal{E}(q) = \infty$.

Step2: For all the conflicting-edges (Ti, Tj) such that $Ti \in before(T)$ and $Tj \in after(T)$, resolve it into $Ti \rightarrow Tj$.

Step3: Delete all the remaining conflicting-edges in the WTPG. Then $\mathcal{E}(q)$ is the length of the critical path from $T0$ to Tf in the WTPG.
□

Example 3.4: Figure 4 illustrates the above procedure. For simplicity, $w(T0 \rightarrow Ti)$ is set to 0 for all Ti. $T0$ and its edges are not displayed in the figure. In the WTPG of Figure 4-(a), suppose that $T5$ now issues a lock-request q which conflicts with $T6$. Then, if q has been granted, $\{T5 \rightarrow T6\}$ is generated. Therefore $before(T5) = \{T4\}$ and $after(T5) = \{T6\}$. Then $(T4, T6)$ is resolved into $T4 \rightarrow T6$ as shown in Figure 4-(b). The critical path in Figure 4-(b) is $T4 \rightarrow T6$ of length 10. Thus $\mathcal{E}(q) = 10$. □

$\mathcal{E}(q)$ is computed by $O(max(n, e))$, (n: the number of nodes, e: the number of edges in a WTPG): the *Step2* of $\mathcal{E}(q)$ is computed by the depth-first traverse, and its *Step3* is by the topological sort.

Let $C(q)$ be the set of lock-declarations conflicting with a lock-request q. $C(q)$ is found in the lock-table. Then our scheduling strategy is:

> a lock-request q is granted only when q has the smallest value of $\mathcal{E}(q)$ in $C(q)$.

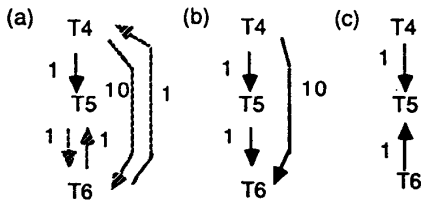

Figure 4: examples of K-conflict WTPG

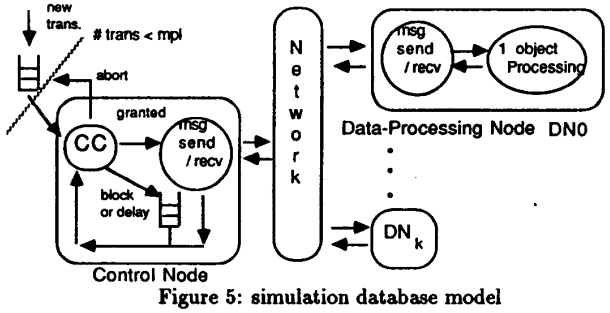

Figure 5: simulation database model

This strategy reduces the contentions by a local optimization, because it estimates only the degree of the contentions occurring at the present schedule.

Using this strategy, the scheduler CC2 behaves as follows when q is issued to a locking-granule d:

CC2(INPUT q: lock-request on d: data)
Step1: If q conflicts with the lock holding d, q is blocked.

Step2: Compute $\mathcal{E}(q)$. If q causes a deadlock, q is delayed.

Step3: If q has the smallest value of $\mathcal{E}(q)$ in $C(q)$, q is granted. Otherwise q is delayed.

□

Example 3.5: In *Example 3.4*, let q' be a lock-request of $T6$ such that q' conflicts with q of $T5$. Figure 4-(c) shows the WTPG where q' has been granted in the WTPG of Figure 4-(a). $(T4, T6)$ has been deleted by the *Step3* of $\mathcal{E}(q')$ in this figure. Since $\mathcal{E}(q) = 10 > \mathcal{E}(q') = 1$, CC2 delays the lock-request q of $T5$ when q is submitted.

□

For alleviating the complexity at the *Step3* of CC2, we limit the size of $C(q)$ to K $(= 0, 1, 2, \ldots)$ by the next constraint:

> Each lock-declaration may conflict with K lock-declarations at most.

At the start of a new transaction T, CC2 tests this 'K-conflict' constraint. If this test fails, T is aborted.

When K is fixed, the *Step3* of CC2 is computed by $O(K \times max(n, e))$. The CC2 under this constraint is named 'K-conflict WTPG scheduler' or 'K-WTPG'. Even K-WTPG of $K = 1$ accepts a WTPG which is not a chain-form.

3.4 Reducing control overhead

For reducing the control overhead, CHAIN and K-WTPG estimate the degree of the contentions again only when
1) a fixed period has elapsed after the last computation, or
2) a transaction commits or starts after the last computation.

Furthermore K-WTPG recomputes $\mathcal{E}(q)$ if
3) a new precedence-edge is generated.

If none of the above conditions hold, CHAIN uses the full SR-order W which was most recently computed at the *Step2* of CC1. In K-WTPG, $\mathcal{E}(q)$ is set to its most recently computed value if it exists.

4 Performance Evaluation

4.1 Simulation model

This section tests the performance of four schedulers in the BAT processing. Schedulers we test are: Chain-WTPG scheduler (CHAIN), K-conflict WTPG scheduler of K=2 (K2), Atomic Static Lock (ASL) in [9], Cautious Two Phase Lock (C2PL) in [10], and NODC (NO Data Contention). NODC grants any lock at any time for clarifying the upper bound of performance. C2PL is a variant of the strict two-phase lock in cautious schedulers [10]. C2PL has a transaction precedence graph (i.e. the WTPG without weights in Section3.1) for predicting a deadlock. In C2PL, a lock-request q is granted if and only if q is not blocked and does not cause a deadlock. C2PL delays q if q causes a deadlock. ASL starts a transaction T if and only if T can hold all the necessary locks at its start. Notice that these schedulers have no deadlock.

Figure 5 displays the simulation model of a shared nothing database machine. Its parameters are listed in Table 1. This model has one control node and several data-processing nodes. *NumNodes* is the number of data-processing nodes. Partitions are located at data-processing nodes, where node's ID = (partition's ID modulo *NumNodes*). Among *NumParts* partitions, *NumHots* partitions are designated as a hot set if necessary.

A new transaction T arrives at the control node with arrival rate λ in the exponential distribution. In each experiment, a new transaction is given its sequence of steps by "*Pattern: step1* → ... → *stepN*". This means a sequential execution from *step1* to *stepN*. Each step is an access to a partition, and a step is expressed as defined in *Example 2.1* of Section 2.2. This model causes the unbalance of load when executing a single BAT on a shared nothing architecture. Since our interest is the scheduling of BATs updating files, each transaction have update-operations to files.

The centralized control-node (CN) is modeled as a *CPUspeed* MIPS processor. CN manages the concurrency control of partition locking-granules. When a lock of a partition is granted to T, T is sent to the data-processing node storing this partition. When T commits (or starts), CN spends *committime* (or *startuptime*) as a coordinator of two-phase commitment.

The model of a data-processing node (DN) is given by

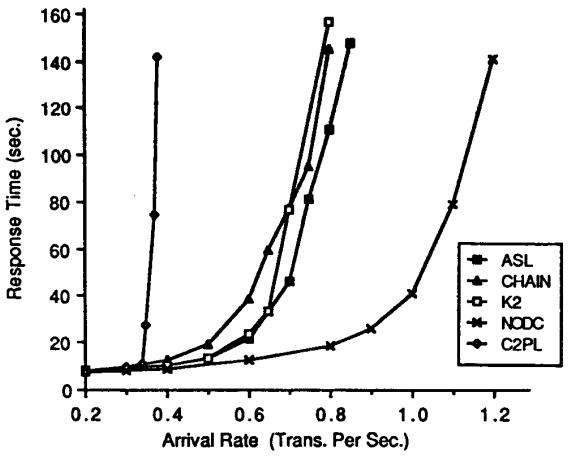

Figure6. Experiment1: Arrival Rate vs. Response Time.

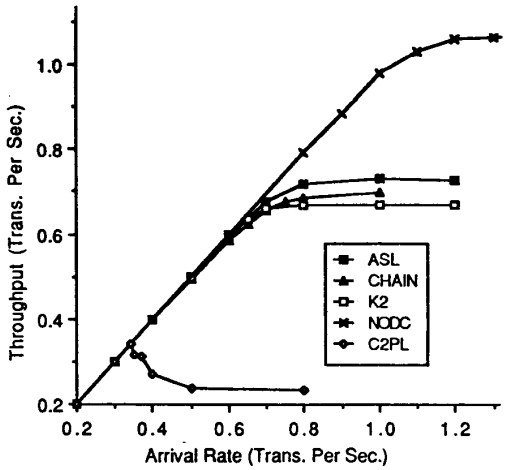

Figure7. Experiment1: Arrival Rate vs. Throughput

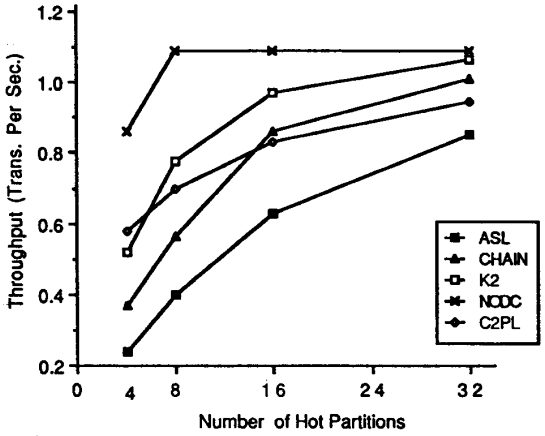

Figure8. Experiment2:
Num.of Hot Partitions vs. Throughput at Resp.Time = 70 sec.

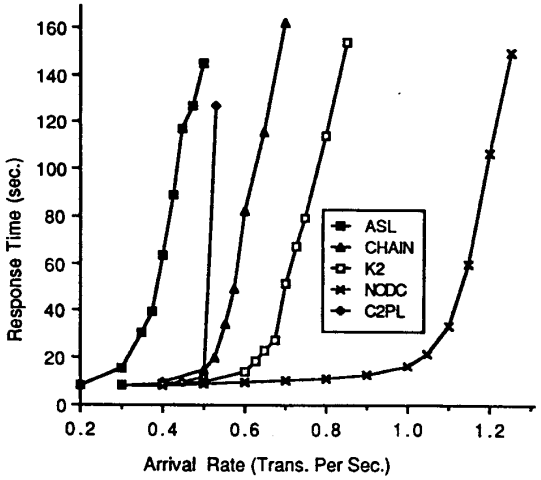

Figure9. Experiment3: Arrival Rate vs. Response Time

Table1: table of parameters

Parameter	Meaning	Value
NumNodes	number of data-processing nodes	8
NumParts	number of partitions	16 ~ 40
NumHots	number of hot partitions	4 ~ 32
mpl	multi programming level	infinite
λ	transaction arrival rate	0 ~ 1.5 TPS
CPUspeed	CPU speed of the control node	4 MIPS
netdelay	network delay time	negligible
msgtime	message send/receive CPUtime	2 ms
startuptime	transaction startup CPUtime	2 ms
committime	commitment CPUtime	7 ms
ddtime	CPUtime of deadlock detection in C2PL	1 ms
kwtpgtime	CPUtime of computing $\mathcal{E}(q)$ in K2	10 ms
chaintime	CPUtime of computing the globally optimized SR-order in CHAIN	30ms
toptime	CPUtime of chain-form test	5 ms
ObjTime	1 object processing time at a node	1000 ms
keeptime	period of control-saving	5000 ms

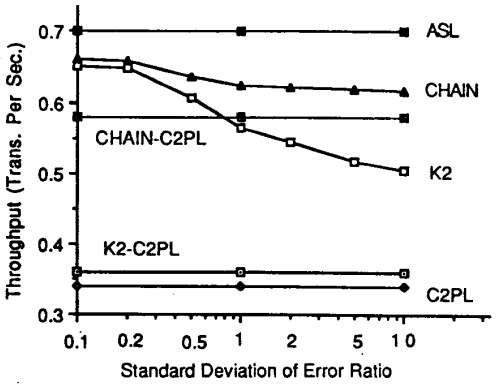

Figure10. Experiment4:
Error Ratio vs. Throughput at Resp.Time = 70 sec.

ObjTime. *ObjTime* is the time for processing 1 object at a DN. (One object is a unit of data for bulk data processing, as defined in Section 2.2.) A DN executes transactions in a round robin manner: i.e. when a transaction T ends a bulk data processing of 1 object, the DN stops T and starts the next waiting transaction. At this time T sends a message to CN for adjusting the weight of the WTPG, as described in Section3.1. T is sent back to CN after ending one step.

This simple model for a DN is justified because a bulk-operation is implemented by pipeline between processors and disks. In addition, a bulk-operation can be executed in I/O bound region when most of the accessed data is filtered out [11]. Control overheads of nodes such as initiating a cohort is ignored because they are far smaller than *ObjTime*.

In Table1, the values of concurrency control parameters *ddtime*, *chaintime*, and *kwtpgtime* have been determined by instruction counts of the control programs. *ObjTime* is set to 1 second. This corresponds to scanning about 60 tracks (2.5 mega byte) per disk in FDS-R. Since *ObjTime* is much larger in practical cases, this setting makes us overestimate the overhead of control.

NumNodes is set to 8, and *NumParts* are chosen among the values in Table 1. Every 8 partitions represents a range-partitioned relation on 8 nodes. The values of *NumParts* in Table 1 are far smaller than those of the simulation studies for short term transactions in [6] [8] [2]. This is because a database has only tens of files scanned by BATs, while these files have thousands of fine locking-granules for short term transactions.

As performance metrics, we use mean response time *RT* (the time from the creation of a transaction to its completion) and throughput (Number of completed Transactions Per Second, TPS). At each measured point with a fixed arrival rate, Our simulation has run during 2,000,000 clocks (1 clock = 1ms) with multi programming level of infinity.

4.2 The case of blocking

In the BAT processing, high data contention degrades performance owing to blocking and a hot set. The first experiment evaluates how well the schedulers avoid chains of blocking.

Experiment1:
Pattern1: `r(F1:1)`→ `r(F2:5)` → `w(F1:0.2)` → `w(F2:1)`, with $NumParts = 16$. The two partitions F1 and F2 are randomly chosen among $NumParts = 16$ partitions, each of which has the size of 5 objects.

This *Pattern1* is a model of the following BATs: 'join the selected result of F1 with F2, and update these partitions depending on the joined result'. The first two steps of *Pattern1* require X-locks, and cause chains of blocking. The first step `r(F1:1)` reads 20% of data in F1 using its indices, and `r(F2:5)` scans F2 without indices. The last two steps update 10% of the read data. The smaller operand in the join operation is accessed at first for hashing algorithms [4].

Figure 6 and 7 show the mean response times and the throughputs as a function of the arrival rate respectively.

In Figure 6, we compare the schedulers by the throughputs at the mean response time of 70 seconds. Looking at this figure, ASL has the best throughput because ASL has no blocking. In contrast, chains of blocking has degraded the throughput of C2PL greatly. ASL, CHAIN, and K2 have from 1.9 to 2.0 times higher throughput than C2PL. Comparing ASL with CHAIN and K2, we can see that CHAIN and K2 avoid the chains of blocking as successfully as ASL does. Furthermore CHAIN does not outperform K2 largely. This implies the scheduling strategy of K2 works as successfully as that of CHAIN.

In Figure 6, resources are saturated at the arrival rate of $\lambda_S = 1.08$ TPS. (because the mean response time of NODC is 70 seconds at λ_S). λ_S is much greater than the arrival rates where the response time of the other schedulers is 70 seconds. This clarifies a feature in the BAT processing where BATs update files. That is, high data contention degrades performance at much smaller arrival rate than saturation of resources does. In the short term transaction processing, this thrashing by data contention ('DC thrashing' in [9]) occurs near or over the arrival rate where resources are saturated.

This high data contention limits useful utilization of resources. In Figure 7, the mean useful resource-utilization ratio of each scheduler is expressed by the ratio of its throughput to that of NODC. (the 'useful' utilization is the utilization except that used by aborted transactions). In this figure, the utilization ratio of ASL, CHAIN, or K2 is not very high (about 64 % = 0.7 TPS/1.1 TPS).

Clearly resources of a node are saturated when this node executes a bulk-operation. The useful utilization is low in Experiment 1 because the number of concurrently running BATs is limited by high data contention.

4.3 The case of a hot set

The second experiment tests the schedulers when high data contention occurs on a hot set.

Experiment2:
Pattern2: `r(B:5)` → `w(F1:1)` → `w(F2:1).`, $NumHots =$ 4, 8, 16, or 32. The partition B is chosen randomly among 8 'read-only' partitions, and each node stores one 'read-only' partition of size 5 objects. The two partitions F1 and F2 are chosen randomly among the other *NumHots* partitions, each of which has the size of 1 object. The read/write steps request S/X-lock respectively. With smaller *NumHots*, higher contention occurs on the hot set.

Figure 8 shows the throughputs at $RT = 70$ seconds as a function of *NumHots*.

ASL keeps a WTPG to be a set of isolated points. Thus ASL can start the smallest number of transactions, and achieves the lowest throughput in this figure. At *NumHot* = 4 or 8, the chain-form constraint of CHAIN also degrades its throughput.

K2 performs best because it has no constraint on the form of a WTPG, and because it starts the highest number of transactions. Although C2PL also permits any WTPG, C2PL is outperformed by both K2 and CHAIN at *NumHots*

= 16 or 32. This is explained as follows: C2PL has higher congestion of resources (about 70% in this figure) at these *NumHots*, and so the blocking time of a transaction is longer than that at *NumHots* = 4 or 8. As the result, C2PL has chains of blocking at the last two steps of *Pattern2*, and is outperformed.

The next pattern of a transaction *Pattern3* is the same as *Pattern2* except that the blocking time of a transaction is longer than that in Experiment2.

Experiment3:
Pattern3: `r(B:4)→w(F1:1)→w(F2:2).`, with *NumHots*=8.
The three partitions B, F1, and F2 are chosen in the same way as that in Experiment2. But the first step and the last step in *Pattern3* have different I/O demands from those in Experiment2. Consequently the blocking time is longer than that in Experiment2.

Figure 9 displays the mean response times as a function of the arrival rate. The throughput of C2PL at $RT = 70$ seconds is 0.5 TPS, 30% lower than 0.7 TPS at $NumHots = 8$ in Experiment2. From this unstable property, we can see that C2PL is very sensitive to the blocking time. In contrast, CHAIN and K2 keep from 1.2 to 1.8 times higher throughput than ASL and C2PL in this experiment. This stable performance are desirable in the BAT processing, because the BAT processing does not have a fixed workload: it has large changes in both the blocking ratio and the frequency of access to hot sets.

From these three experiments, we can see that high data contention limits the inter-transaction parallelism of BATs when they update files. Therefore the intra-transaction parallelism are necessary for high (greater than 90%) useful utilization of resources. An solution is to distribute files randomly on all the nodes of a shared nothing database machine. This file placement benefits parallelism of BATs [7], but the message overhead degrades performance of the short term transaction processing, as described in Section 2.1.

4.4 Test of sensitivity

The fourth experiment tests CHAIN and K2 when transactions declare erroneous I/O demands.

Experiment4:
Pattern1 used in Experiment1, with $NumParts$=16.

The I/O demand of each step is estimated at C by the formula: $C = C_0 \times (1+x)$, where C_0 is the exact I/O demand of the step, and x is the error ratio. x is given by the normal distribution where the mean is 0 and the standard deviation is σ. ($C = 0$ when $x \leq -1$). With greater σ, CHAIN and K2 estimate the degree of data/resource contentions incorrectly, and so chains of blocking cannot be avoided.

As the lower bound of CHAIN (or K2), we use the scheduler CHAIN-C2PL (or K2-C2PL). CHAIN-C2PL (or K2-C2PL) is C2PL except that a WTPG must keep the constraint of 'chain-form' in Section3.2 (or 'K-conflict' in Section3.3). If a new transaction does not keep these constraints, these schedulers delay its start.

Figure 10 displays the throughputs at RT=70 seconds as a function of σ. At $\sigma = 1$, CHAIN (or K2) achieves 4.6% (or 13.8%) lower throughput than it does at $\sigma = 0$. At this high error ratio of $\sigma = 1$, they still keep much higher throughput than C2PL.

K2 is more sensitive to the error ratio than CHAIN. Since CHAIN-C2PL has the high throughput of 0.58 TPS, the insensitivity of CHAIN is due to its chain-form constraint. In contrast, K2-C2PL has the low throughput of 0.36 TPS. Thus we can see that K2 avoids chains of blocking mainly by using weights in a WTPG.

5 Conclusion

This paper has proposed new concurrency control schemes for Bulk Access Transaction (BAT) on shared nothing database machines. As file placements, we have assumed the placements reducing overhead of messages, such as range-partitioning or partial declustering.

We have proposed Weighted Transaction Precedence Graph (WTPG) and two schedulers using it: Chain-WTPG scheduler (CHAIN) and K-conflict WTPG scheduler (K-WTPG). Both schedule BATs so that data/resource contentions are reduced. For building up a WTPG, a transaction must predeclare its sequence of steps and their I/O demands.

CHAIN predicts the globally optimized serializable order W at first. CHAIN grants a lock-request only if it is consistent with W. A WTPG in CHAIN must be a 'chain-form' for computing this W in a polynomial time. On the other hand, K-WTPG grants a lock-request q only if q causes the lowest contentions at the present schedule.

K-WTPG does not predict the contentions occurring in the future, while CHAIN does. Thus the scheduling strategy of CHAIN is more strict than that of K-WTPG. But K-WTPG accepts any form of a WTPG, while CHAIN does not.

Our simulation has clarified the following results:

1: When the blocking ratio of a lock-request is very high, CHAIN and K-WTPG avoid chains of blocking as perfectly as Atomic Static Lock (ASL in [9]) does. In this case, CHAIN is a little better than K-WTPG, which in turn outperforms Cautious Two Phase Lock (C2PL in [10]) greatly.

2: When high data contention occurs on a hot set like master-files, K-WTPG is better than both C2PL and CHAIN, which in turn outperform ASL largely.

3: K-WTPG is more sensitive to erroneous I/O demands than CHAIN is. But both avoid chains of blocking successfully even when very erroneous I/O demands are declared.

4: In the BAT processing, high data contention limits the inter-transaction parallelism of BATs. Therefore the intra-transaction parallelism are necessary for utilizing resources very usefully.

In our simulation, CHAIN and K-WTPG achieve from 1.2 to 1.8 times higher throughput than ASL and C2PL do. This superiority depends on file placements of shared nothing architectures. Thus the impact of the intra-transaction parallelism should be analyzed both in the BAT processing and in mixed workloads. In mixed transaction processing,

different schedulers are necessary for different classes of jobs. CHAIN and K-WTPG are desirable for the class of BATs when shared nothing database machines use the file placements reducing the message overhead.

References

[1] Tandem Performance Group. A Benchmark of NonStop SQL on the Debit Credit Transaction. In *Proc. ACM-SIGMOD '88*, pages 337–341, 1988.

[2] Bhide,A. An Analysis of Three Transaction Processing Architectures. In *Proc. 14th Int'l Conf. Very Large Data Bases*, pages 339–350, 1988.

[3] Copeland,G. et al. Data placement in bubba. In *Proc. ACM-SIGMOD'88*, pages 99–108, 1988.

[4] DeWitt,D.J. et al. Gamma - High Performance Dataflow Database Machine. In *Proc. 12th Int'l Conf. Very Large Data Bases*, pages 228–237, 1986.

[5] Gray,J. The Transaction Concept: Virtues and Limitations. In *Proc. 7th Int'l Conf. Very Large Data Bases*, 1981.

[6] 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*, 1984.

[7] Carey,M.J. et al. Parallelism and Concurrency Control Performance in Distributed Database Machines. In *Proc. ACM-SIGMOD '89*, 1989.

[8] Agrawal,R. et al. Models for Studying Concurrency Control Performance: Alternatives and Implications. In *Proc. ACM-SIGMOD '85*, pages 108–121, 1985.

[9] Tay,Y.C. Locking Performance in Centralized Databases. *ACM Trans. Database Syst.*, 10(4):415–462, 1985.

[10] Nishio,S. et al. Performance Evaluation on Several Cautious Schedulers for Database Concurrency Control. In *Proc. 5th Int'l Workshop Database Machines*, pages 212–225, 1987.

[11] Kitsuregawa,M et al. Query Execution for Large Relations on Functional Disk System. In *IEEE Proc. 5th Int'l Conf. Data Engineering*, 1989.

[12] Graham,R. et al. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In *Annals of Discrete Mathematics 5, North Holland*, pages 287–326, 1979.

Appendix:
Optimization algorithm of Chain-WTPG scheduler

This appendix describes the algorithm to compute the full SR-order W, such that W makes the shortest critical path in any chain-form WTPG $G(1,N)$ in Figure 3 in Section 3.2.

The following notations are used:

1. with k fixed, $n[k]$ is abbreviated to nk.
2. $G(i,j)$ refers to the subgraph of $G(1,N)$ which consists of $n0$ and $\{n[i], n[i+1], .., n[j]\}$.
3. '$(n[i], n[i+1])$ is set upwards' means that this edge is resolved into $n[i] \leftarrow n[i+1]$. '$(n[i], n[i+1])$ is set downwards' means that it is resolved into $n[i] \rightarrow n[i+1]$.
4. $min(A,B)$ (or $max(A,B)$) is the minimum (or the maximum) between A and B.

We can find the shortest critical path in $G(k,N)$ from the following structural parameters. Each member of these parameters is referred to by *parameter.member*.

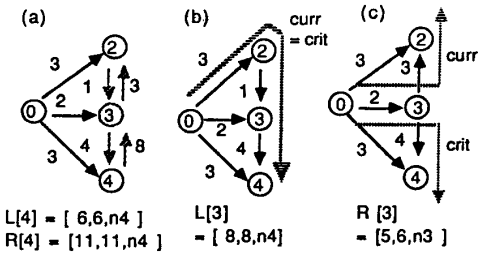

Figure 11: examples of $L[k]$ and $R[k]$

Definition 3 $L[k]$ and $R[k]$ are the triplets $[curr, crit, rev]$ defined below:

Suppose that $(n[k-1], n[k])$ has been set downwards in $G(k-1,N)$. Let $S1(k-1,N)$ be the full SR-order to make the shortest critical path in this $G(k-1,N)$. Then $L[k]$ is defined on the edge $n[k-1] \rightarrow n[k]$ as follows:

1. $L[k].crit$ = the length of the shortest critical path in the $G(k-1,N)$ resolved by $S1(k-1,N)$.
2. $L[k].rev$ = the smallest label among the labels i, such that $(n[i], n[i+1])$ is set upwards in $S1(k-1,N)$. $L[k].rev = N$ if such the label does not exist.
3. $L[k].curr$: the length of the path $n0 \rightarrow n[k-1] \rightarrow n[k] \rightarrow .. \rightarrow n[L[k].rev]$.

Suppose that $(n[k-1], n[k])$ has been set upwards in $G(k-1,N)$. Let $S2(k-1,N)$ be the full SR-order to make the shortest critical path in this $G(k-1,N)$. Then $R[k]$ is defined on the edge $n[k-1] \leftarrow n[k]$ as follows:

4. $R[k].crit$ = the length of the shortest critical path in the $G(k-1,N)$ resolved by $S2(k-1,N)$.
5. $R[k].rev$ = the smallest label among the labels i such that $(n[i], n[i+1])$ is set downwards in $S2(k-1,N)$. $R[k].rev = N$ if such the label does not exist.
6. $R[k].curr$ = the length of the critical path from $n0$ to $n[k-1]$ in the $G(k-1, R[k].rev)$ resolved by $S2(k-1,N)$.

□

Example 4.1: Figure 11-(a) illustrates $G(2,4)$, $L[4]$, and $R[4]$. Figure 11-(b) shows the case of computing $L[3]$: Suppose that $(n2, n3)$ has been set downwards. Then, between $n3 \rightarrow n4$ and $n4 \rightarrow n3$, the former makes the shorter critical path of length 8. So $S1(2,4) = \{n2 \rightarrow n3 \rightarrow n4\}$. In $S1(2,4)$, there is no $n[i]$ such that $(n[i], n[i+1])$ is set upwards. The critical path in Figure 11-(b) is $n0 \rightarrow n2 \rightarrow n3 \rightarrow n4$. Thus we set $L[3] = [8, 8, n4]$. $R[3]$ is similarly computed in Figure 11-(c).

Theorem 1 Suppose that $L[i]$ and $R[i]$ for all $i = k+1$ to N are given in $G(k,N)$. Then the following formulae compute $S1(k,N)$, $S2(k,N)$ in Definition 3 and the full SR-order $S(k,N)$ to make the shortest critical path P in $G(k,N)$:

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

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

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

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

if $L[k+1].crit \leq R[k+1].crit$ then
  $S(k,N) = S1(k,N)$.
else
  $S(k,N) = S2(k,N)$.

□

Example 4.2: In Example 4.1, $L[3].crit = 8 > R[3].crit = 6$. Hence $S(2,4) = S2(2,4) = \{n2 \leftarrow n3\} \cup S1(3,4) = \{n2 \leftarrow n3 \rightarrow n4\}$. □

Theorem 2 Suppose that $G(k-1,N)$ and all the parameters $L[i]$ and $R[i]$ from $i = k+1$ to N are given. Then $L[k]$ and $R[k]$ are computed by $O(N-k)$. □

Outline of the proof:
We show the algorithm Lcomp() for computing $L[k]$, in the C language-like notation. The variables a[k], b[k] and r[k] are the weights defined in Figure 3.

```
Lcomp() {
        /* procedure for L1[k] */
temp = L[k+1].curr - r[k] + r[k-1] + a[k];
if ( temp <= L[k+1].crit )
  L1[k] = [ temp, L[k+1].crit, L[k+1].rev ];
else {
  L1[k].crit
  = min( max(V(h), R[h+1].crit)); /* EXPR1 */
    for all h = k+1 to L[k+1].rev

  L1[k].rev  = h0;
  L1[k].curr = C(h0);
/*    h= h0 takes the minimum in EXPR1.
 *    C(h) and V(h) are defined as follows:
 *  V(k-1) = C(k-1)  = r[k-1];
 *  V(h) = max(r[h], V(h-1)+a[h]);(k =< h)
 *  C(h) = C(h-1) + a[h]; ( k =< h )
 */
} /* end of L1[k] */
    /* procedure for 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] */
if  ( L1[k].crit <= L2[k].crit )
    L[k] = L1[k];
else
    L[k] = L2[k];
}
```

In Lcomp(), $L[k]$ is set to either $L1[k]$ or $L2[k]$ according to their values of *crit*. $L1[k]$ represents $L[k]$ such that $(n[k], n[k+1])$ has been set downwards. $L2[k]$ does when this edge has been set upwards. $L2[k]$ assumes only the case of resolving $G(k-1,N)$ by $\{n[k-1] \rightarrow n[k]\} \cup S2(k,N)$.

$L1[k]$ is computed for the case where $G(k-1,N)$ is resolved at first by $\{n[k-1] \rightarrow n[k]\} \cup S1(k,N)$. In this case, a new path $P0 = \{ n0 \rightarrow n[k-1] \rightarrow n[k] \rightarrow n[k+1] \rightarrow ... \rightarrow n(L[k+1].rev)$ } is generated. The variable temp in Lcomp() is the length of $P0$.

If $P0$ is longer than the shortest critical path of $G(k,N)$, $P0$ is the critical path when using $\{n[k-1] \rightarrow n[k]\} \cup S1(k,N)$. But $P0$ may get shortened by setting upwards $(n[h], n[h+1])$ in $P0$. i.e. using $S(h) = \{n[k-1] \rightarrow n[k] \rightarrow \ldots \rightarrow n[h]\} \cup S2(h,N)$ where $k+1 \leq h \leq L(k+1).rev$. The expression EXPR1 in Lcomp() computes $L1[k]$ in this case.

In EXPR1, $C(h)$ is the length of the path $P(h) = n0 \rightarrow n[k-1] \rightarrow n[k] \rightarrow \ldots \rightarrow n[h]$. $V(h)$ is the length of the critical path in the $G(k-1,h)$ resolved by $P(h)$. $S(h)$ has the critical path of length $max(V(h), R[h+1].crit)$ in $G(k-1,N)$. So EXPR1 correctly finds $h = h0$ such that $S(h0)$ makes the shortest critical path in $G(k-1,N)$. Thus $L1[k]$ is computed by $O(N-k)$.

The other cases for $L1[k]$ are trivial and they are omitted here. The algorithm of $R[k]$ is described in the lemma 1.
□

Computing $L[k]$ and $R[k]$ from $k = N$ to 2, $L[2]$ and $R[2]$ are computed. Then Corollary 1 holds from theorem 1 and theorem 2.

Corollary 1 In a chain-form WTPG $G(1,N)$ of Figure 3, the full SR-order making the shortest critical path is computed by $O(N^2)$. □

Lemma 1 In theorem 2, $R[k]$ is computed by $O(N-k)$.

This proof is similar to that of theorem 2 and omitted. The following algorithm Rcomp() computes $R[k]$ from $G(k,N)$ with $R[i]$ and $L[i]$ $(k+1 \leq i \leq N)$.

```
Rcomp() {
      /* procedure for R1[k]  */
temp = R[k+1].curr + b[k];
if ( max( r[k-1], temp ) <= R[k+1].crit )
    R1[k] = [ temp, R[k+1].crit, R[k+1].rev ];
else if ( max(r[k-1], temp ) == r[k-1] )
    R1[k] = [ r[k-1], r[k-1], R[k+1].rev ];
else {
    R1[k].crit
     = min( max( V(h), L[h+1].crit) ); /* EXPR2 */
       for all h = k+1 to R[k+1].rev

    R1[k].rev  = h0;
    R1[k].curr = V(h0);
       /*  h = h0 takes the minimum in EXPR2. */
}
      /*  procedure for 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;
if  ( R1[k].crit <= R2[k].crit )
    R[k] = R1[k];
else
    R[k] = R2[k];
}
```

In Rcomp(), $R1[k]$ represents $R[k]$ such that $(n[k], n[k+1])$ has been set upwards. $R2[k]$ does when setting this edge downwards.

The expression EXPR2 assumes $S(h) = \{n[k-1] \leftarrow n[k] \leftarrow .. \leftarrow n[h]\} \cup S1(h,N)$. It finds $h = h0$, where $S(h0)$ makes the shortest critical path in $G(k-1,N)$.

The variable V(h) is the length of the critical path in the $G(k-1,h)$ resolved by $S(h)$. C(h) is the length of the path $n0 \rightarrow n[h] \rightarrow n[h-1] \rightarrow \ldots \rightarrow n[k-1]$. They are computed by the formulae:

```
V(k-1) = C(k-1) = r[k-1];
C(h) = C(h-1) - r[h-1] + r[h] + b[h];  (k ≤ h)
V(h) = max( C(h), V(h-1) );  (k ≤ h)
```

□

Theorem 3 It is NP-hard to compute the full SR-order which makes the shortest critical path in any WTPG.

Outline of the proof: The static general job-shop scheduling problem *JS* is represented by a disjunctive graph [12]. This graph is the WTPG where all the outgoing edges of a node have the cost of the node. Since *JS* is NP-hard, this theorem holds. □