A Non-Blocking Locking Method and Performance Evaluation on Network of Workstations

YU Ge (于 戈), WANG Guoren (王国仁), and ZHENG Huaiyuan (郑怀远)

Department of Computer Science, Northeastern University, Shenyang, P.R. China

E-mail: {yuge,wanggr,zhenghy}@mail.neu.edu.cn;

JIN Taiyong (), Kunihiko Kaneko and Akifumi Makinouchi

Department of Intelligent Systems, Kyushu University, Fukuoka, Japan

E-mail: {jty,kaneko,akifumi}@db.is.kyushu-u.ac.jp

Received June 11, 1999; revised June 16, 2000.

Abstract A network of workstation (NOW) can act as a single and scalable powerful computer by building a parallel and distributed computing platform on top of it. WAKASHI is such a platform system that supports persistent object management and makes full use of resources of NOW for high performance transaction processing. One of the main difficulties to overcome is the bottleneck caused by concurrency control mechanism. Therefore, a non-blocking locking method is designed, by adopting several novel techniques to make it outperform the other typical locking methods such as 2PL: 1) an SDG (Semantic Dependency Graph) based non-blocking locking protocol for fast transaction scheduling; 2) a massively virtual memory based backup-page undo algorithm for fast restart; and 3) a multi-processor and multi-thread based transaction manager for fast execution. The new mechanisms have been implemented in WAKASHI and the performance comparison experiments with 2PL and DWDL have been done. The results show that the new method can outperform 2PL and DWDL under certain conditions. This is meaningful for choosing effective concurrency control mechanisms for improving transaction processing performance in NOW environments.

Keywords distributed and parallel database, concurrency control, transaction management, locking mechanism, NOW (network of workstations)

1 Introduction

As low-cost and high-performance workstations and microcomputers such as Sun Ultra Sparc and Pentium III become more and more popular, network of workstations (NOW) that consists of several sites or even hundreds of sites is available. Moreover, the progress of high-speed communication network technique such as ATM and Fast-Ethernet makes it more feasible. One of the most promising features of NOW is to act as a single, scalable powerful computer, which is regarded as a rival of massively-parallel processor (MPP) computers since an NOW will have the same computing capability and transaction processing capability as those of a currently available MPP computer, while it is much cheaper, more flexible and scalable than an MPP computer. This target can be achieved by building a parallel and distributed computing platform on top of it. WAKASHI is such a platform system for distributed and parallel database management[1]. However, one of the main difficulties to overcome is the bottleneck caused by concurrency control mechanism.

As we know, locking mechanism is a general approach for concurrency control. In a memory mapping mechanism supported by most types of workstations, locking is a very

Partially supported by the Excellent Young Teacher Foundation and the Doctoral Program Foundation of the Ministry of Education, China.

efficient way to implement. Therefore, we focus our topic on the lock-based concurrency control methods. Locking methods guarantee database consistency by two different policies: blocking the execution of some transactions (to wait for the release of the requested locks) or restarting some transaction (to abort non-serializable execution). It is believed that blocking policy has better performance than restart policy because the former has lower restart rate, and restart operations take a long time or even form resource contention. Thus the most widely adopted method in commercial products is the blocking policy based method: 2-phase-locking (2PL).

Under lower conflict rate of transactions when few transactions compete the same data resource, 2PL demonstrates better performance than other known methods[2]. However, as conflict rate rises, 2PL can easily lead to data contention, that is, most transactions are blocked into waiting queues and time is wasted by waiting in the queues so that transaction throughput cannot increase with the number of active transactions and thus the system performance degrades[3]. Under NOW environments, there are more conditions that might worsen data contention problems, such as the following. 1) Scalability: NOW might cover the network of an office, a department or an enterprise. NOW might contain large number of computers, and the number of sites is dynamically changeable in terms of available sites, problem size, or load balance consideration. When more sites are involved, the level of concurrency might increase, which will lead to high level of data contention for accessing shared data. 2) Advanced applications: NOW should support parallel transactions or more complex transactions than ordinary systems. There are many new database applications such as multimedia data services, engineering designs, OLAP and data mining, which contain I/O-intensive operations as well as CPU-intensive operations.

Furthermore, the gains in NOW performance require that transaction processing throughput should continue to increase with Multi-Programming Level (MPL) before reaching its resource contention point. This only can be done by increasing the concurrency of transaction executions. However, we cannot use blocking policy because its waiting feature impedes throughput increase at a fixed MPL. Thus we have to pursue a non-blocking approach on the basis of restart policy.

To overcome the limitations of blocking based lock methods, several methods have been developed by combining with the restart policy[4−6]. DWDL (Distributed Wait-Depth Limited) method is an effective one among them[7]. DWDL resolves the data contention problem of 2PL by restarting some transaction whenever a wait-chain becomes longer than one. In fact, DWDL is a hybrid of blocking policy and restart policy, that is, when lock conflicts occur, a transaction must wait if the wait-chain is less than two, otherwise, some transaction in the wait-chain must restart.

In this paper, we propose a new concurrency control method, which is similar to SGT (Serializable Graph Testing Method)[8,9] in the respect that a conflicting transaction continues its execution provided that some standards (SGT method uses serializability) are satisfied. It is known that SGT method provides higher concurrency than other methods such as 2PL method[10] and TSO (Time Stamp Ordering) method[11], because it allows all possible and correct executions unlike those methods that only allow certain kinds of executions (for example, TSO only permits executions of transactions that obey time stamp ordering). However, SGT method has two shortages: 1) it requires much memory space to store Serializable Graph (SG) and much CPU time to certificate SG, which might cause resource contention in ordinary computer environments; and 2) the high concurrency of database operations also risks high restart rates of transactions. These two points might cancel the profits brought by high concurrency. Moreover, new applications require extended transactions that don't necessarily use serializability as the unique correctness criterion. For example, in a video-server system, the half-finished image data are allowed to be read by a browser application.

To implement the new method effectively, we employ several novel techniques: 1) a backup-page based group undo algorithm is exploited for fast abortion; 2) a multi-processor and multi-thread based transaction manager is exploited for fast execution; and 3) a massively virtual memory based server cache is maintained for supporting the prefetch effects of restarted transactions.

The rest of the paper is organized as follows. Section 2 introduces WAKASHI with the two typical locking methods: 2PL and DWDL for later performance comparison. Section 3 presents the principle and design of NBL algorithm. Section 4 discusses the implementation issues. The performance experiments and evaluation are reported in Section 5. Finally, Section 6 concludes the paper.

2 WAKASHI and Typical Locking Methods

WAKASHI is a distributed and parallel object server system, developed at Northeastern University of China and Kyushu University of Japan[1].

2.1 WAKASHI and Locking Mechanism

WAKASHI, running on a distributed UNIX platform, consists of servers that run as daemon processes, and a client library that is linked into user programs. The server performs data access control and transaction management, and the client library provides communication interfaces between clients and servers. In WAKASHI, each server provides a repository of the database and the log file. A database is a collection of persistent heaps and a heap is a basic storage unit. Furthermore, a heap is divided into pages of equal length, and a page is a basic control unit. When manipulating a database, the heap files are mapped into the server virtual memory and the client virtual memory, respectively. Thus, the objects in a heap are loaded into the workspace along with the page mapping. All desirable controls such as locking and dereferencing are automatically performed by the server, thus the application can directly manipulate these objects in a transparent way. WAKASHI also offers client location transparency for remote access by Distributed Shared Memory (DSM) approach. All necessary actions on DSM control (mainly cache coherency control) are automatically performed by servers.

At each site, there are at least one server and any number of clients. An application is launched as a client by linking its application program with the WAKASHI client library. An application may consist of many transactions, moreover, they may be nested. In WAKASHI, both clients and servers are implemented as multi-threaded processes. That is, when a client accesses several database files, a thread for each accessed heap is created in a client, and a thread for each opened heap is created in a server. Thus there is a "multi-threaded" connection between the client and the server when the client accesses more than one heap. Moreover, the control information is distributed on different heaps. Thus, the control on different heaps can be performed in parallel.

In WAKASHI, the actions of a transaction are a partial order of read operations and write operations on pages. Every transaction uses the private memory of the client as its workspace, which is a mapping from the cache of the server. A read operation fetches a page from a persistent heap into a client workspace, and a write operation puts a page from a client workspace into a persistent heap. Since pages can be cached in the workspace, a restarted transaction also does useful work to prefetch some required pages into the cache memory.

Fig.1 shows the system structure of WAKASHI with its control units. In a client, each HeapClient is a thread for accessing on a heap, and TransClient is the coordinator process for the threads of the client and the interface for transaction operations. In a server,

each HeapServer is a thread that performs control and provides services on a heap, and TransServer is the coordinator process for the threads of the server and the transaction manager. The main data structures are also shown in Fig.1. LockT is the lock control table that keeps the information for concurrency control.

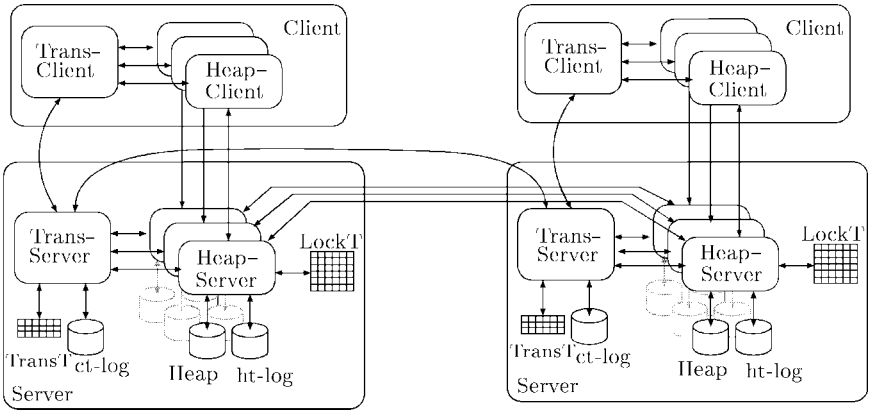

Fig.1. The system structure of WAKASHI.

By using memory mapping mechanism supported by hardware, locking is one of the most effective ways for concurrency control, because the access on conflicted data or invalid data is automatically detected, and then the required control is enforced correctly by the server. In WAKASHI, we use three protection modes for a page P in a working space: (1) none: P is invalid, (2) read: P is read-allowed, and (3) write: P is read-allowed and write-allowed.

2.2 Two-Phase Locking (2PL) Method

The practical implementation of 2PL is the strict 2PL method[9], in which lock conflicts between transactions are resolved by making the lock requester wait for the lock holder in an FIFO queue. The waiting-for relationship can be described by Waiting For Graph (WFG). By combining with 2-Phase-Commit (2PC) protocol, all locks of a transaction are held until the transaction is terminated. When a transaction is to be aborted or committed, all its read locks are released before starting abort or commit. In case of abort, the write lock on a page can be released immediately after finishing the undo operation on the page. In case of commit, all write locks must be released after finishing commit. When a lock is released, the transaction(s) in the waiting queue is (are) scheduled in FCFS (First-Come-First-Served) manner to avoid the risk kn which read lock requests jump ahead thus delay other write lock requests indefinitely.

Deadlock is possible when two or more transactions are waiting for the lock held by some other transaction and none can continue its execution. In this case, a cycle that contains some of these waiting transactions is formed in the WFG. In this paper, we adopt WFG-based deadlock detection method, by which, whenever a transaction is blocked, local deadlocks are checked. If a deadlock is found, the transaction with the most recent starting time among those involved in the deadlock cycle, that is, the youngest transaction, is selected as the victim to be restarted. Global deadlock detection is checked by a "Snoop" process, which periodically collects waiting-for information from all sites to form a global WFG and checks for any global deadlock. To balance workload, the "Snoop" is allocated at an idlest site and can migrate dynamically in terms of workload change of the system.

2.3 Distributed Wait-Depth Limit (DWDL) Method

When data contention becomes serious, the blocking feature of 2PL inevitably reduces

system utilization because the blocked transactions will exhaust a lot of system resources like memory and swap space and make data contention more serious. Therefore, DWDL method selects some blocked transactions to restart to reduce data contention[7]. DWDL doesn't allow the waiting-queue of a locking to be longer than one, otherwise, an involved transaction is selected to restart. The rules of selecting the victim transaction are based on transaction maturity, which is defined by an age function. The age of transaction T, denoted as $L(T)$, is defined to be the difference of the current time and the startup time of T.

The implementation of DWDL on WAKASHI is the same as 2PL method, except that the $L(T)$ restart rules are used to check transaction conflict in WFG. With DWDL method, restarting a transaction is triggered by waiting-depth overflow, and the scheduling is free of deadlocks because no path longer than one exists.

3 Non-Blocking Locking (NBL) Method

Essentially, 2PL and DWDL are blocking-based methods. They are hard to provide maximum effective concurrency to take full use of system resources of NOW environments. In order to overcome the shortages caused by the blocking feature, a Non-Blocking Locking (NBL) method is proposed in this paper.

3.1 Transaction Dependency

In the following, we define some concepts needed for our method.

Definition 3.1 (Conflict). *For any two operations O_i and O_j that belong to two different transactions T_i and T_j, respectively, if they operate on the same page P and one of them is a write operation, then T_i and T_j conflict on P.*

Definition 3.2 (Precede). *When two transactions have a conflict on a page, the first coming transaction is said to precede the second one, or the second coming transaction is said to follow the first one.*

Definition 3.3 (Depend). *If T_2 reads or writes a page P that has been written by T_1 most recently, then T_2 is said to be dependent on T_1, denoted as T_2 $\mathcal{D}$ $T1$.*

3.2 Semantic Dependency Graph

It has been proved that the concurrent execution of transactions is correct if there is no cycle in their Serializable Graph (SG)[9]. SG is exploited for scheduling transactions in SGT method[8]. The advantage of SG-based method is supporting maximum concurrency of transaction executions. However, not all concurrent executions are effective, and some might be restarted later due to non-serializability, thus maximum concurrency doesn't mean maximum transaction throughput.

The main problem of SGT is allowing unconditional execution provided that it is serializable. To reduce the potential high restart rate when data contentions become serious, we improved this method in the following aspects:

- Restart a transaction in advance provided that it has high risk of abortion.
- SGT method requires much space to store SG because SG contains active transactions (not terminated yet) and committed transactions that depend on some active transaction(s). In our method, SDG only contains the active transactions. Thus NBL can save storage space for SDG compared with SGT method. This is because SGT allows a transaction to commit before the transaction that it depends on while NBL doesn't allow. NBL requires the depending transaction to wait until the depended transaction is terminated.

• SGT only contains precedence semantics. We add dependency semantics into Serializable Graph. Depended transactions form a *dependency chain*. This semantics can be used to support optimized processing as stated later.

Definition 3.4 (Semantic Dependency Graph). *An SDG is a directed graph* $\langle V, E\rangle$. *V is the node set that contains the active transactions. E is the edge set that contains all* $\langle T_i, T_j\rangle$ $(i \neq j)$ *such that one of T_i's operations conflicts with and follows one of T_j's operations. Each edge has a dependency type* (*Read or Write as defined in Definition* 3.3) *as its label.*

Definition 3.5 (Dependency Chain). *A dependency chain is a directed path that connects any two nodes in an SDG.*

A node may be involved in different dependency chains. To describe features of an *SDG*, three important parameters are defined as follows.

Definition 3.6 (Depth and Width).

(1) *depth*(n)*: The depth of node n is the maximum number of edges in any dependency chain that contains n.*

(2) *inWidth*(n)*: The inWidth of node n is the number of n's incoming edges, that is, the number of transactions that directly depend on this transaction.*

(3) *outWidth*(n)*: The outWidth of node n is the number of n's outgoing edges, that is, the number of transactions that this transaction depends on directly.*

3.3 Restart Rule

Using the results from DWDL, we restrict the depth of dependency chains with the concern of data contention and set it as 1 in this paper. To describe the features of different nodes in SDG with dependency depth 1 (called SDG-1), we classify them into three types.

Definition 3.7 (Node Type). *Three types of node are defined.*

(1) *Independent node, denoted as* n_node. *It has neither incoming edge*(s)*, nor outgoing edge*(s) (*see Fig.*2(a)). n_node *transaction doesn't depend on any other transactions, and no other transaction*(s) *depends on it.*

(2) *Depended node, denoted as* i_node. *It has incoming edge*(s)*, but no outgoing edge*(s) (*see Fig.*2(b)). i_node *transaction doesn't depend on any other transactions, but some transaction*(s) *depends on it,*

(3) *Depending node, denoted as* o_node. *It has outgoing edge*(s)*, but no incoming edge*(s) (*see Fig.*2(c)). o_node *transaction depends on some other transactions, but no other transaction*(s) *depends on it.*

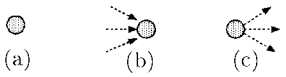

Fig.2. Type of node in SDG. (a) n_node. (b) i_node. (c) o_node.

The three kinds of nodes have the parameters as follows:

1) *depth*(n_node)=0; *inWidth*(n_node)=0, *outWidth*(n_node)=0;

2) *depth*(i_node)=1; *inWidth*(i_node)=m, *outWidth*(i_node)=0;

where m is the number of incoming edges.

3) *depth*(o_node)=1; *inWidth*(o_node)=0, *outWidth*(o_node)=n;

where n is the number of outgoing edges.

Besides restricting the depth like DWDL, we restrict the *outWidth* of a transaction with the concern of probability of cascaded aborting. Suppose the probability of a transaction to be aborted is x, then the probability of the transaction T with $outWidth(T) = n$ is greater than x, since $1 - (1 - x)^n > x$ for $x < 1$. Thus, the larger the *outWidth*, the higher the probability of aborting T. To reduce the risk of cascaded aborting, the transaction with large *outWidth* must be restarted. On the other hand, *inWidth* also should be considered

because of the cost of cascaded aborting. The larger the *inWidth*, the higher the cost of the cascaded aborting. Therefore, to select victim transaction, two factors are used, not just using age function as in DWDL approach.

Next we discuss the restart rules for different combinations of three kinds of nodes. There are 9 possible cases of locking requests, as shown in Fig.3, where long arrows mean the edge to be added into SDG. There are 7 cases (noted as solid arrows) in which we should consider to restart a transaction, and other 2 cases (noted as dash arrows) need not consider restarting.

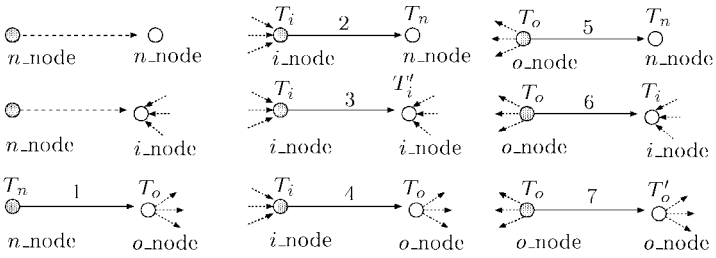

Fig.3. Graph of depend depth and width.

- Case 1: n_node T_n follows o_node T_o (see 1 of Fig.3), then restart T_o.
- Case 2: i_node T_i follows n_node T_n (see 2 of Fig.3), then restart T_i.
- Case 3: i_node T_i follows i_node T_i' (see 3 of Fig.3), if $inWidth(T_i) > inWidth(T_i')$, then restart T_i' else restart T_i.
- Case 4: i_node T_i follows o_node T_o (see 4 of Fig.3), then restart T_o.
- Case 5: o_node T_o follows n_node T_n (see 5 of Fig.3), if $outWidth > maximum\_width$, then restart T_o.
- Case 6: o_node T_o follows i_node T_i (see 6 of Fig.3), then restart T_i.
- Case 7: o_node T_o follows o_node T_o' (see 7 of Fig.3), if $outWidth(T_o) > outWidth(T_o')$, then restart T_o else restart T_o'.

4 Implementation Issues

This section presents the major implementation issues related to the NBL method.

4.1 NBL Algorithm on WAKASHI

In the NBL method, to increase concurrency, a page P locked by transaction T_1 is allowed to be read or written by another transaction T_2. In this case, T_1 has to retain P at first like in the case of nested transactions and let T_2 hold it. This action is called "Lock Delegating", by which the dependency between T_1 and T_2 is still kept and an edge is added into SDG if the edge doesn't exist yet. The special processing for "Lock Delegating" operation is needed in memory-protection mechanism, that is, after receiving DELEGATE signal, T_1's client will set the protection of P as none to disallow T_1 to access P again.

The following is the main part of the NBL algorithm. Suppose transaction T_2 encounters a lock conflict with T_1 on page P when checking against LockT. *WidthLIMIT* is set as 2, and *DepthLIMIT* is set as 1.

if the edge $\langle < T_2, T_1 \rangle$ already exists in SDG then
  if (locking-request = READ) then
    dependency-type($\langle T_2, T_1 \rangle$) := READ; {overriding WRITE by READ}
else
  Add the edge $\langle T_2, T_1 \rangle$ with dependency-type to SDG;
  {re-calculate Depth and Width}

```
    inWidth(T1) := inWidth(T1) + 1;
    outWidth(T2) := outWidth(T2) + 1;
    depth(T1) := maximum(depth(T1), depth(T2) + 1);
    depth(T2) := depth(T1);
    {check whether the depth or the width overflows}
    if (depth(T1) > DepthLIMIT) or (outWidth(T2) > WidthLIMIT) then
        {select victim in terms of inWidth and outWidth rule}
        if (inWidth(T2) ≤ inWidth(T1)) or (outWidth(T2) ≥ outWidth(T1)) then
            throw_signal(T2,RESTART);
        else throw_signal(T1,RESTART);
        return(WAIT);
    end;
end;
throw_signal(T1,DELEGATE,P);
return(SUCC);
```

4.2 Commit and Abort Rule

With the NBL method, the isolation of conflicting data is broken by allowing the result of an un-committed transaction to be read by other transactions. The formed dependency between them will affect the commit or abort of dependent transactions, the semantics of which are similar to the commit-dependency and abort-dependency defined in ACTA model[12]. To distinguish the effects of different kinds of dependency, two kinds of dependency are defined. In fact, more kinds of dependency like dirty-read can be defined for different semantics. We don't discuss them for the limitation of the space.

Definition 4.1 (Read-dependent). *If T_2 reads page P that has been written by T_1 most recently, then T_2 is said read-dependent on T_1 , denoted as T_2 $\mathcal{D}_r$ T_1.*

Definition 4.2 (Write-dependent). *If T_2 writes page P that has been read or written by T_1 most recently, then T_2 is said write-dependent on T_1, denoted as T_2 $\mathcal{D}_w$ T_1.*

In the following, we give some analysis for their effects and special processing to cope with commit and abort. Suppose T_2 is dependent on T_1.

• *Commit.* If T_2 is committed before T_1, two cases exist. (1) If T_2 is read-dependent on T_1, we will lose *recoverability* of T_2 in case that T_1 is aborted. (2) If T_2 is write-dependent on T_1, then T_1 cannot read data written by T_2, and cannot write data read/written by T_2. This means the locks of T_2 should be kept even after T_2 is committed. This will raise complexity of the algorithm and require extra storage spaces. Thus we choose to let T_2 wait for T_1. Hence, T_2 is not allowed to commit before T_1 if T_2 is dependent on T_1.

• *Abort.* If T_1 is aborted before T_2, two cases exist. (1) If T_2 is read-dependent on T_1, then T_2 is to be aborted too because T_2 already reads the results of T_1. This is cascaded abort. (2) If T_2 is write-dependent on T_1, then T_2 does not need to abort because T_1's results do not affect T_2. However, undoing T_1 might affect T_2's results. The special processing is: for page P written by T_1, if P is not re-written by T_2, then P is undone by the before image of T_1; otherwise, P is not undone but the before image of T_2 is changed with T_1's. Hence, undo processing must be performed differently for two cases.

In terms of above analysis, two rules are defined.

• **Commit rule**: T_2 is not allowed to commit before T_1 if T_2 is dependent on T_1, that is, if (T_2 $\mathcal{D}_r$ T_1 or T_2 $\mathcal{D}_w$ T_1), then Commit(T_1) after Commit(T_2).

• **Abort rule**: If T_2 is read-dependent on T_1 and T_1 is aborted, then T_2 must be aborted too, that is, if (T_2 $\mathcal{D}_r$ T_1) and Abort(T_1), then Abort(T_2).

4.3 Fast Undo Processing

As we know, restart based concurrency control methods require more abort operations than blocking based concurrency control methods. If undo operations are performed by using ordinary log files, the overhead will be big because (1) complex log operations are needed for different cases, and (2) a lot of log records are to be read to abort each dependent transaction. Moreover, dependency chains of NBL method lead to cascaded abort. To reduce overheads caused by normal undo operations and cascaded abort, we can take use of the redo-only recovery mechanism of WAKASHI[1]. By this method, the before image of each modified page P is saved in the virtual memory in the form of backup page. The backup page of P is attached to the entry of P in the lock table LockT.

When a lock is delegated from one transaction to another transaction, the original page is saved in the manner of stack on behalf of the writer transaction. When a transaction T is aborted, each page entry of the LockT is checked and the page must be undone if it has the backup page of T.

With backup page approach, we can process the undo operations on the same page for all dependent transactions in one step, instead of undoing them for every transaction as ordinary systems. We call this method as "group abort". When a transaction T is aborted, a dependency tree whose root is T is processed by using "group abort" method. The algorithm to abort a transaction T is as follows:

for each page P_i that has a backup page of T, do
  get the backup page P_i' of T;
  if T_k is write-dependent on T, then
    replace the backup page of T_k with P_i';
  else Undo P_i with P_i';
  remove P_i' and all backup pages of read-dependent transactions of T;
end;
for each read-dependent transaction T_i of T, **do**
  abort(T_i);

5 Performance Evaluation and Comparison

This section shows the impacts of different lock managers on the system performance in NOW environments. Our experiments are based on the multi-user OO7 (M-OO7) benchmark[13], which was designed by the University of Wisconsin. With M-OO7, we investigate the behaviors of three locking algorithms with different workloads: update-bounded type and query-bounded type. At each site, a workload generator invoking M-OO7 transactions is implemented to increase multiple programming level (MPL) by 2, 4, 8, 16, ..., until the thrashing point is found. The transaction throughput rate is measured for performance comparison.

5.1 Overview of Multiuser OO7 Benchmark

M-OO7 defines two types of testing databases: 1) PrivateDB accessible only by one client; 2) SharedDB accessible by all clients. The size is scalable with the number of clients. In our experiments, the sizes of small PrivateDB and small SharedDB are 4.6MB and 4.3MB, respectively, and the sizes of medium PrivateDB and medium SharedDB are 49MB and 43MB, respectively. The sample database structure for three clients is shown in Fig.4.

The workload of a transaction is generated in terms of three parameters as follows.

• *Operation Configuration Vector* (OCV) is a 4-tuple $< Pr, Sr, Pw, Sw >$, which represents the probability of read or update operations on the PrivateDB or the SharedDB. The sum of the four probabilities is 1.

• *RepeatCount* defines the total number of operations. A small number generates a short transaction, and a large number generates a long one.

• *ThinkingTime* defines the time intervals in every operation. A short time generates a computing-intensive transaction, and a long time generates an interactive one. In our experiments, ThinkingTime is set as 0 to increase conflicting rate.

For each client, it runs a transaction as follows.

```
begin_transaction();
for RepeatCount do
    get a random pair (DB,OP) in terms of OCV;
    select PrivateDB or SharedDB in terms of DB;
    follow a single random path in the assembly tree and locate a composite part;
    for each node of depth-first traversal do
        perform the read or update operations on each atomic part in terms of OP;
    end;
    sleep(ThinkingTime);
end;
commit_transaction();
```

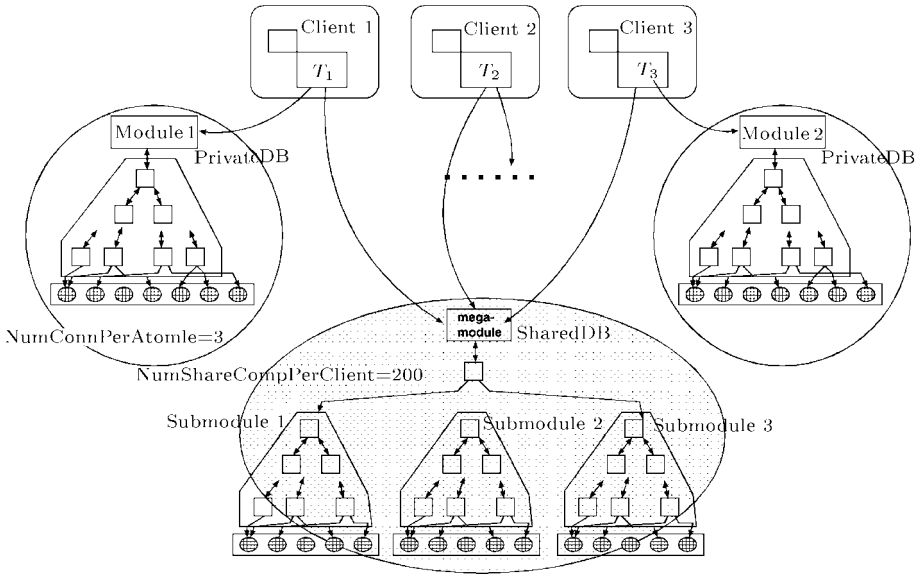

Fig.4. Structure of multiuser OO7 database.

In the experiments, each transaction accessed 45927 different objects and 439587 different objects, in small database and medium database, respectively.

5.2 Testbed Configuration

The experiments were made at different MPLs. We selected the average throughput rate at one site for local testing, and at two remote sites for distributed testing. In each experiment, the clients were allocated on three machines evenly. The site where Shared database and Private databases were created was called Local site, and the other site was called Remote site. We got the results for the 2PL, DWDL and the NBL locking managers that were implemented in WAKASHI, respectively. The results were obtained by repeatedly running the same testing. The throughput was computed by the UNIX function gettimeofday() and the final results were the average of 4 runs of the benchmark operations.

We used three symmetric Sun Sparc/20 workstations that were connected by an isolated Ethernet. The three machines have the same configuration with 4 SuperSparc+ processors

(50MHz), 64 MB main memory, and a Segate ST31200N 1.05GB disk driver. The swapping space is 180 MB and the page size is 4KB. The operating system is SUN OS 5.5. The testing programs were coded with C++ and OML C++ binding language of WAKASHI.

5.3 Checkin/Checkout Experiment

In the Checkin/Checkout experiments which simulate CAD applications, half of the transactions act as designers who perform Checkin operations, and the other half of the transactions act as designers who perform Checkout operations. The parameters of the experiments are set to be update-bounded as follows.

$Checkout\_OCV = \langle 0, 30, 70, 0 \rangle$; $Checkin\_OCV = \langle 30, 0, 0, 70 \rangle$

The results are shown in Fig.5, Fig.6, and Fig.7. We do not include the test using 2PL for Medium DB because transactions managed by 2PL often deadlock and we could not get correct result.

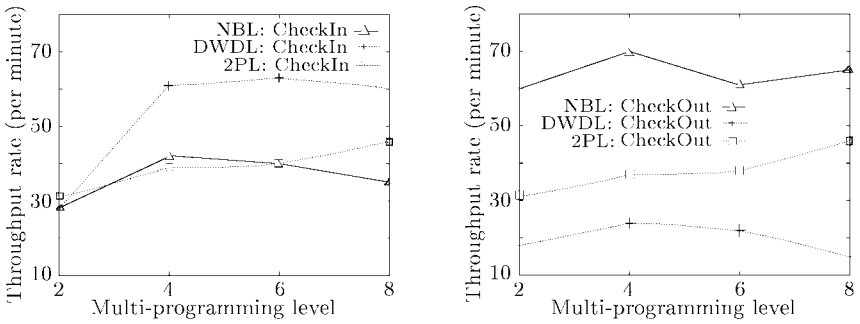

Fig.5. Update-bounded, local (SmallDB, RepeatCnt=2000).

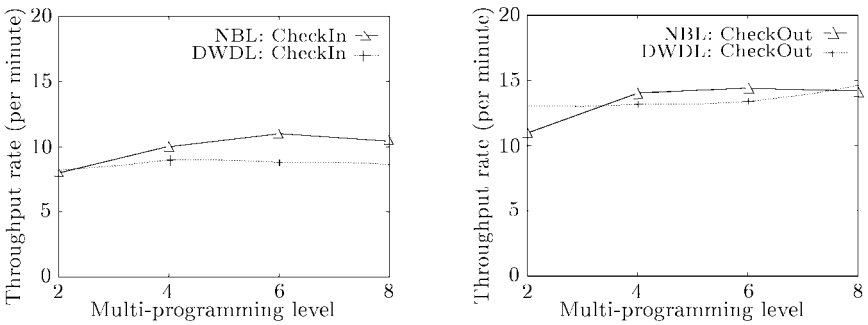

Fig.6. Update-bounded, local (Medium DB, RepeatCnt=10).

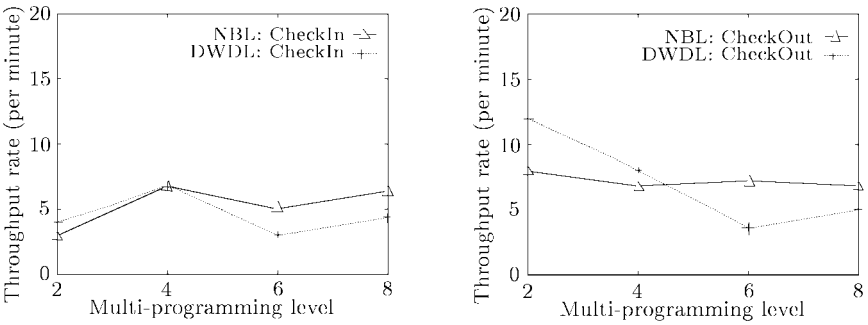

Fig.7. Update-bounded, remote (Medium DB, RepeatCnt=10).

• Local operations: Fig.5 shows that concurrency control policy is not the main factor affecting the throughput of the operations for Small DB. Because the size of Small DB is smaller than the size of main memory and so the most of Small DB is in main memory during operations, the operations for Small DB are CPU-intensive. CPUs can continue to process some CPU-intensive short-duration transactions while some other transactions are blocked. In this case, 2PL is not so bad. In contrast, NBL increases the throughput of the operation for Medium DB by 10%–20% compared with DWDL as shown in Fig.6. This is because the operations for Medium DB tend to be I/O-intensive, and the update-bounded feature causes more blocked transactions that have to wait.

• Distributed operations: Fig.7 shows NBL outperforms DWDL for larger MPL. This is because larger MPL leads to more blocked transactions for DWDL than for NBL.

5.4 Producer/Consumer Experiment

In the producer/consumer experiment, which simulates the video-on-demand applications, one producer transaction acts as a video camera that generates video data, other consumer transactions act as displayers to read and display the video data.

For this experiment, the parameters are defined to be query-bounded as follows:

$producer\_OCV = \langle 70, 0, 0, 30 \rangle$; $consumer\_OCV = \langle 0, 70, 30, 0 \rangle$

The results are shown in Fig.8, Fig.9, and Fig.10. We do not include the test using 2PL for Medium DB either.

• Local operations: For the same reason as in the Checkin/Checkout test, the operations producer and consumer are also CPU intensive in Fig.8. Then, concurrency control policy is not the main factor affecting the throughput of the operations for the Small DB. In case of Medium DB as shown in Fig.9, DWDL outperforms NBL and the maximum throughput of DWDL is 30% larger than NBL. This is because query-bounded feature causes few blocking operations so that transactions don't have to wait.

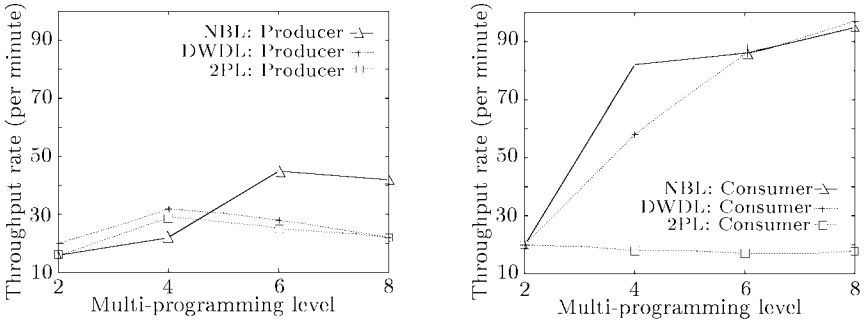

Fig.8. Query-bounded, local (small DB, RepeatCnt=2000).

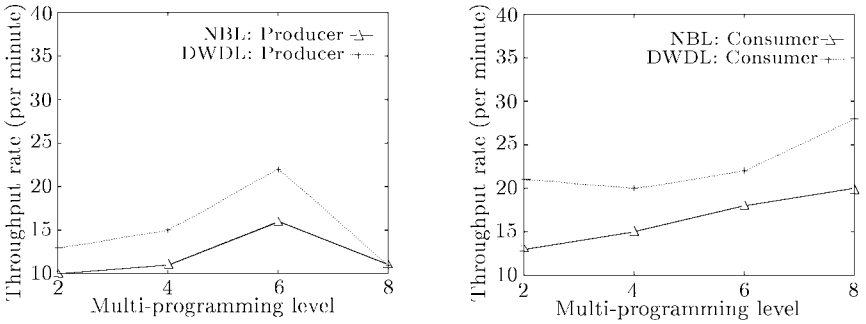

Fig.9. Query-bounded, local (Medium DB, RepeatCnt=10).

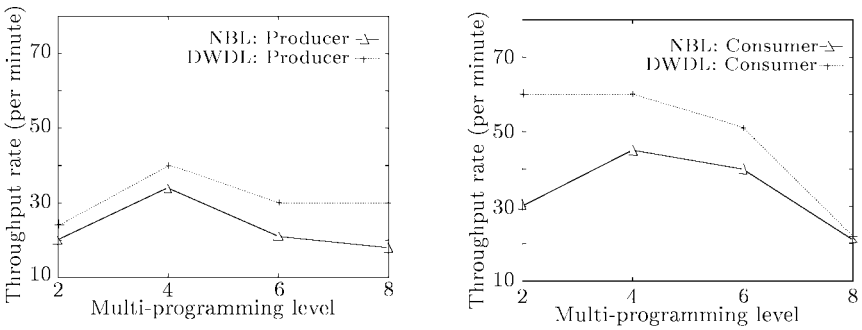

Fig.10. Query-bounded, remote (Medium DB, RepeatCnt=10).

• Distributed operations: Since the number of messages of NBL concurrency controller is larger than that of DWDL, Fig.9 shows that DWDL increases total throughput of the operation by 10%–50% compared with NBL for distributed environments.

6 Conclusions

In this paper, we have studied several alternatives for synchronizing accesses to databases by multiple transactions in NOW environments. They are 2PL (a conventional blocking-based method), DWDL (an improved blocking-based method), and NBL (a restart-based method proposed by us).

The results of experiments with multiuser OO7 benchmark in our environment show that: 1) The update-bounded transactions with NBL scheduling can increase more throughput than those with DWDL methods in local and distributed environments; 2) Query-bounded transactions with DWDL can increase more throughput than those with NBL in local and distributed environments; 3) Both NBL and DWDL outperform 2PL for medium size database.

Using high-bandwidth network such as ATM, the throughput in NOW environments is expected to become near to that in multi-processor environment, which is to be proved in our next research topic. Therefore, we conclude that both NBL and DWDL are suitable for advanced applications which are usually with long duration and heavy workload, especially in NOW environments with high-bandwidth network.

References

[1] Yu G, Kaneko H, Bai G *et al.* Transaction management for a distributed object storage system WAKASHI — design, implementation and performance. In *Proc. 12th ICDE,* New Orleans, Feb. 1996, pp.380–389.

[2] Agrawal R, Carey M J, Livny M. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Systems,* Dec. 1987, 12(4): 609–654.

[3] Thomasian A. Two-phase locking performances and its trashing behavior. *ACM Trans. Database Systems,* Dec. 1993, 18(4): 579–625.

[4] Carey M J, Krishnnamurthy S, Livny M. Load control for locking: The 'Half-and-Half' approach. In *Proc. 9th ACM PODS Conf.,* 1990, pp.72–84.

[5] Franaszek P A, Robinson J T, Thomasian A. Concurrency control for high contention environments. *ACM Trans. Database Systems,* Jun. 1992, 17(2): 304–345.

[6] Moenkeberg A, Weikum G. Conflict-driven load control for the avoidance of data-contention thrashing. In *Proc. IEEE Conf. Data Engineering,* Kobe, 1991, pp.632–639.

[7] Franaszek P A *et al.* Distributed concurrency control based on limited wait-depth. *IEEE Trans. Parallel Distributed Sys.,* 1993, 11: 1246–1264.

[8] Badal D Z. Correctness of concurrency control and implications in distributed databases. In *Proc. IEEE COMPSAC Conf.,* Nov. 1979, pp.588–593.

[9] Bernstein P A, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems. Addison-Wesley, MA, 1987.

[10] Gray J N. Notes on database operating systems. Operating Systems: An Advanced Course, Lecture Notes in Computer Sci., Springer-Verlag, Berlin, 1978, 60: 393–481.
[11] Bernstein P A, Goodman N. Timestamp based algorithms for concurrency control in distributed database systems. In *Proc. 6th VLDB,* Oct. 1980, pp.285–300.
[12] Chrysanthis P K, Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proc. SIGMOD,* May 1990, pp.194–203.
[13] Carey M J, DeWitt D J, Kant C *et al.* A status report on the OO7 OODBMS benchmarking effort. ftp://ftp.cs.wisc.edu/oo7/retrospective.ps

YU Ge is a professor at Northeastern University, China. He received his B.E. degree and M.E. degree from Northeastern University in 1982 and 1986, respectively, and his Ph.D. degree from Kyushu University, Japan in 1996. He is a member of CIMS Expert Group of the National '863' High-Tech Programme of China, a member of IPSJ, ACM, and ACM SIGMOD.

WANG Guoren is a professor at Northeastern University, China. He received his B.E. degree, M.E. degree, and Ph.D. degree from Northeastern University in 1988, 1991 and 1996, respectively. He did post-doctoral work at Kyushu University from 1996 to 1997. He is a member of ACM, and ACM SIGMOD.

ZHENG Huaiyuan graduated from Northeastern University in 1953. He has been a professor at Northeastern University since 1986, and was a member of CIMS Expert Group of the National '863' Programme of China from 1991 to 1996. His research interests include distributed database systems, software engineering, and computer integrated manufacturing (CIM).

JIN Taiyong received his B.E. degree from Northeastern University in 1993, and his M.E. degree from Kyushu University in 1998. Now he is a Ph.D. candidate in Kyushu University. His research interests include parallel transaction processing, object database systems, and distributed shared memory management.

Kunihiko Kaneko is an associate professor at Kyushu University, Japan. He received his B.E. degree, M.E. degree, and Ph.D degree from Kyushu University in 1990, 1992 and 1995, respectively. He is a member of IPSJ, ACM, and IEEE.

Akifumi Makinouchi is a professor at Kyushu University, Japan. He received his B.E. degree from Kyoto University in 1967, Docteur-Ingenieur degree from Universite de Grenoble in 1970, and Ph.D. degree from Kyoto University in 1985. He is a member of IPSJ, ACM, and IEEE.