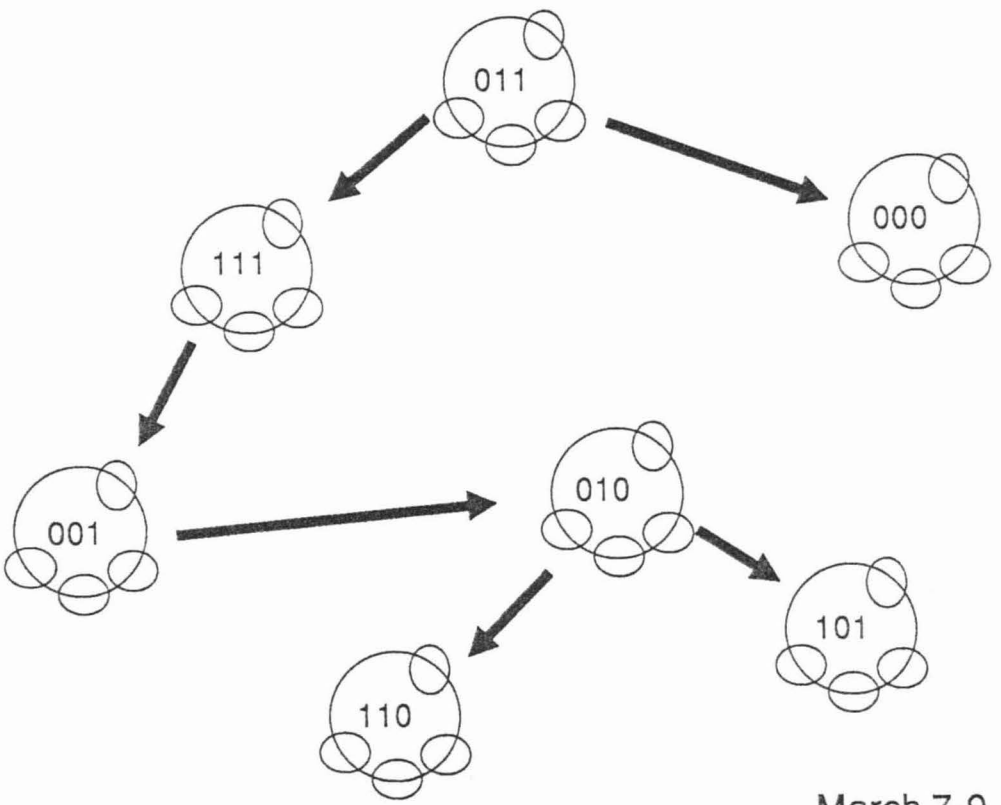
Databases and Parallel Architectures

Proceedings

PARBASE-90

International Conference on Databases, Parallel Architectures, and Their Applications

Edited by N. Rishe, S. Navathe, and D. Tal



March 7-9, 1990
Miami Beach, Florida

Sponsored by
Florida International University
in cooperation with IEEE and Euromicro

IEEE Computer Society Press

Institute of Electrical and Electronics Engineers, Inc.

PERFORMANCE EVALUATION OF A NEW OPTIMISTIC CONCURRENCY CONTROL ALGORITHM

Jonathan Addess* Ehud Gudes* Doron Tal** Naphtali Rishe**

* Department of Computer Science, Ben-Gurion University, Beer Sheva 84105, Israel
** School of Computer Science, Florida International University, Miami, FL 33199

ABSTRACT

A new *optimistic* concurrency control algorithm for a parallel multi-processor database machine is presented. The algorithm is simulated using Flat Concurrent Prolog and it's performance is discussed.

INTRODUCTION

Concurrency control algorithms have received considerable attention since the 1970s when the need for fast and high performance database machines became critical. The traditional approach of locking entails high overhead, especially in highly parallel database machines and therefor optimistic algorithms [1,2] seem more suitable in that environment.

In this report we focus on a modification of the classic Kung & Robinson time-stamp based concurrency control algorithm, proposed in [3]. The algorithm is based primarily on two innovative techniques: query killing notes and weak serializability of transactions. In particular, it prefers long transactions over short queries and thus reduces considerably the number of transaction rollbacks required.

In order to test the validity and evaluate the performance of the proposed algorithm a simulation program was written and run using a realistic set of transactions. This paper first briefly reviews the transactions model and concurrency algorithm presented in [3], and then presents and discusses the simulation results.

The simulation has been performed at the Ben-Gurion University using Flat Concurrent Prolog (FCP) [4]. The advantages of FCP for specifying and implementing parallel algorithms include its refined granularity of parallelism, its declaretiveness and conciseness and it's powerful communication and synchronization primitives. The algorithm maps very naturally into FCP since the concepts of processes and messages map directly into FCP constructs. FCP proved itself as a very useful tool for simulating concurrency control algorithms. Other uses of FCP for simulating parallel database algorithms were reported in [5,6].

THE TRANSACTIONS MODEL

A transaction is a set of interrelated update requests to be performed as one unit [7]. In our model we employ the deferred update scheme. Upon completion of the transaction the DBMS checks its integrity and then physically performs the update. A two-phase commit protocol is used to insure that all updates will be performed at all nodes.

We use here the term *programmatic transaction* to denote the program fragment comprising a transaction. During its execution, a programmatic transaction may issue database queries. Queries which are performed from within a programmatic transaction are called *transactional queries*.

The immediate outcome of a programmatic transaction is an *accumulated transaction*, which is composed of a set of facts to be deleted from the database, a set of facts to be inserted into the database and additional information needed to verify that there is no interference between concurrent transactions. If the verification produces a positive result, then the new instantaneous database is: *((the-old-instantaneous-database) - (the-set-of-facts-to-be-deleted))* U *(the-set-of-facts-to-be-inserted)*.

CH2728-4/90/0000/0522$01.00 © 1990 IEEE

The execution of a transaction statement is composed of four stages:

1) transaction accumulation - The results of the updating instructions are accumulated in the sets D (the set of facts to be deleted from the database), I (the set of facts to be inserted into the database.) and V (the results of the transactional queries which need to be verified). The result of the transaction accumulation, i.e. the outcome (V,D,I) of the run of the program segment, is called the *accumulated transaction*.

2) Integrity validation - at this time the system checks if the resulting database would not violate the integrity constraints.

3) Concurrency validation - using the algorithm below, it is checked that the set V is still consistent, and that this transaction does not interfere with other concurrent accumulated transactions and queries.

4) Physical performance of the accumulated transaction - the new instantaneous database is: *((the-old-instantaneous-database) - D) U I*.

The above model of transactions is mapped onto a parallel semantic database machine described in [3]. Briefly, that machine is a multi-disk multi-processor database machine where all the processors are identical, have access to their own local disks, can process both transactions and queries and may communicate via a high speed communication network. The database is divided among the local disks using a special storage structure described in [3].

In the machine described above, any node may serve as the entry point for transactions and queries, that node is called: the originator node. Queries arrive at the originator node and are divided into sub-queries

which are sent to the appropriate processors. Transactional queries are divided into sub-queries and distributed like any other queries.

Programmatic transactions are executed as part of the host programs. When the program reaches the "end transaction" statement, the accumulated transaction is computed and sent to a transactions originator which distributes them between relevant processors as a set of *sub-transactions* (V1,D1,I1), (V2,D2,I2), ... The above sub-transactions are executed using a *two-phase commit protocol* [8]. As soon as the sub-transaction (Vi,Di,Ii) is examined by its processor i, its components are checked. If verification is successful and there are no conflicts with concurrent accumulated transactions, then a *ready* message is sent to the originator of the accumulated transaction. When the *commit* message arrives, the updates are performed atomically.

THE ALGORITHM

As in [1,2] the algorithm is based on Time-stamps. Time stamps for queries are assigned at the time the query is distributed to the processors. Time stamps for transactions are assigned at the time the accumulated transaction is distributed. The algorithm uses three data structures at each node: two queues and a log. One queue is the waiting queue which contains the transactions or queries which have not been processed yet by this node. The other queue is the hold-queue which contains the sub-transactions which were checked for conflict, found O.K., sent the *ready* message, and are now waiting for a *commit* message. The log contains information on all the sub-transactions and queries which were executed locally during the last reasonable period of time. The log is used during the verification process for checking whether any sub-transaction exists on the log which conflicts with the current transaction or query, and has an earlier time-stamp.

One unique property of the algorithm is the notion of *killing notes*. When a conflict is detected between a transaction and a query, the query is selected to be aborted. However, instead of aborting the query, the node discovering the conflict sends a killing note. If all other nodes have already executed the query, then they may ignore the killing note, and as will be discussed in the simulation part, can save quite a few query aborts. The algorithm is presented in high-level only. See [3] for details.

(A) A sub-transaction T1=(V1,D1,I1) of a transaction T=(V,D,I) arrives at node N1.

1. Perform the verification. If V1 is not O.K. then kill the current transaction and return.
2. If there is a sub-transaction in the hold queue that can interfere with the current sub-transaction then kill the current sub-transaction T1 and return.
3. If there is a younger sub-query Qi in the log which interferes with T1 then send a killing note for Qi.
4. Put the sub-transaction on hold and send an *agree to perform* message to the transaction originator.

(B) A transactional sub-query P1 arrives at node N1.

1. Execute P1.

(C) A sub-query Q1 arrives at node N1 or is popped from the hold queue.

1. If any sub-transaction T1 with a later time-stamp than Q1 which interferes with Q1 has been physically performed or any killing message associated with some sub-transaction T1 has been left for sub-query Q1 and T1 interferes with Q1 then kill the current query and return else: 2. Execute Q1, return the result to the originator and update the log.

(D) A commit message for a sub-transaction T1 arrives at node N1.

1. Remove sub-transaction T1 from the hold queue.
2. If the message is associated with killing notes for sub-queries then for each sub-query Q1 to be killed perform sub-algorithm (C) on it.
3. Perform the physical change in the database and update the log.

Examples of several scenarios which can arise by this algorithm were presented in [3]. We next discuss the simulation in FCP.

SIMULATION DESCRIPTION

Flat Concurrent Prolog (FCP)

A Flat Concurrent Prolog program is a set of guarded Horn clauses of the form: H <--- G1,G2, ... ,Gn | B1,B2, ... ,Bm

Where H is the head predicate, Gs (guard elements) and Bs (body elements) are sets of "procedure" calls. Procedure calls may be solved in any order and thus may be viewed as a system of parallel "processes".

The semantics of FCP are based on non-deterministic process and clause selection. Each state of a computation consists of a multi-set of processes which form the resolvent, a program and the data state. A reduction transition involves non-deterministically selecting a process from the resolvent and a clause from the program, verifying the validity of the guard and reducing the head process to the body processes.

The computation must be "just" i.e. a process, whose reduction is continuously enabled, is eventually taken. No other restrictions on order of process selection exist.

Description of the Simulation Program

A) The network we used is a 3-dimensional cyclic cube of order N, in which each node has 6 neighbors. Therefor the network can have 1,8,27,64 ... etc. nodes.

B) All nodes are the same and each one consists of a few parts (see figure 1). Requests enter the node and arrive at the selector. The selector divides the requests into specific requests regarding data at the node which are sent to the process and general requests which are sent to the originator. Each new request opens a new originator which shares a common input stream with all other originators. The messages from the originators pass through a linear merger and are then merged with the messages from the process and passed to the connector.

C) Communication between nodes: The communication protocol is *msg(IDreceiver,Message)*. The connector at each node is in charge of the communication. Each connector has 7 input-output channels, 6 to it's neighbors and one to itself, and one input channel from the system host. The connectors perform two tasks:

a. Passing messages to the correct node via the shortest route possible.

b. Accepting incoming messages and removing their communication protocol.

The concurrency algorithm itself is performed by the *process* process.

D) The structure of the data at each node is a list of pairs (Key,Value). The range of the key is Node_ID*10 - Node_ID*10+9 so that it will be possible to know at which node the data resides. Transactions and queries' contents are in terms of (Key,Value) pairs.

The structure of a transaction is for example:

msg(Orig_ID,originator([(Rec_ID,sub_t([],[(1014,abc)],[(777,vvv)]))|

More_sub_transactions]))

The structure of a query is for example:

msg(Orig_ID,q_originator([(Receiver_ID,sub_q([(1014,A^)]))|

More_sub_queries]))

Further details on the implementation can be found in [9].

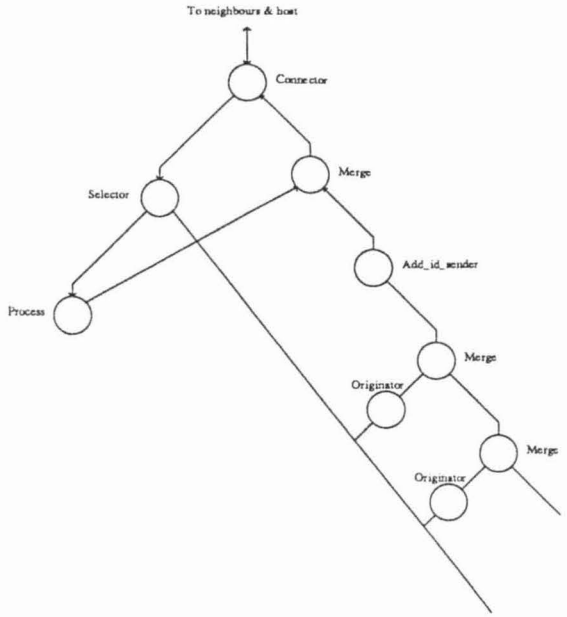


Figure no. 1 - The description of a single node

SIMULATION RESULTS

The first example shows a clash between two transactions:

```
@spawn_hyper#hyper(2,In)
spawned: 0 - 0 - 0
spawned: 0 - 0 - 1
spawned: 0 - 1 - 0
<1> started
spawned: 0 - 1 - 1
spawned: 1 - 0 - 0
spawned: 1 - 0 - 1
spawned: 1 - 1 - 0
spawned: 1 - 1 - 1
@In!msg(111,originator([(101,sub_t([],[(1014,abc)],[(777,vvv)])),
(110,sub_t(U,[],[(888,vvv)]))),(1,sub_t([],[],[(777,vvv)]))]))
@In!msg(11,originator([(101,sub_t([(1014,abc)],[(1014,abc)],[(777,vvv)])),
(110,sub_t([],[],[(888,vvv)])),(1,sub_t([],[],[(555,vv v)]))]))
kill transaction:, 17197
@U=[]
successful transaction:, 14470
```

The second example shows an example of killing notes. A commit message for a transaction whose sub-transaction is in the hold queue arrives with killing-notes. A query arrives and is killed by the notes left by the transaction.

```
@process#process(101,In,Out)
<1> started
@In!(sub_t([(1014, abc)], [(1014, abc)], [(1013, vvv)]),k(101,777))
@In!(commit(101,777+[(101,888)]))
@In!(sub_q([(1014,H^)]),k(101,888))
@Out^
Out = [(101, agree(777 + [])), (101, kill(888)) | _]
```

The third example, without script, involved running the system with a continuous stream of transactions for a fixed period of time on an eight node system with each node containing 10 data items.

The system run with:

a. Transactions containing 3 sub-transactions.

b. Queries containing 3 sub-queries.

c. An evenly distributed load on each node.

d. Transaction:query ratio of 2:8.

produced the following results:

a. All the queries succeeded.

b. Approximately 10% of the transactions were killed.

Changing each transaction to contain 2 sub-transactions and changing the transaction:query ratio to 1:9 caused the percentage of transactions killed to be reduced to approximately 4%.

These case study results clearly show, as the algorithm predicts, that the preference of transactions over queries reduces the contention in this environment.

REFERENCES

[1] Kung H. T., and Robinson J. T., "On Optimistic Methods for Concurrency Control", ACM Trans. on Database Systems, V. 6, No. 2, 1981.

[2] Carey M. J.,"Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions", IEEE Trans. on Software Engineering, V. 13, No. 6, 1987.

[3] Rishe N., Tal D., and Gudes E., "An Optimistic Concurrency control algorithms for distributed-storage semantic database machines", submitted for publication, 1989.

[4] Shapiro E., Concurrent Prolog - Collected Papers, The MIT Press, 1987.

[5] Reches E., Gudes E., Shapiro E., "Parallel access to a distributed database and its implementation in Flat Concurrent Prolog", Weizmann Inst. technical report, CS88-11, July, 1988.

[6] Gudes E., Shapiro E., "A parallel B-tree process structure", Weizmann Inst. technical report, CS89-06, April, 1989.

[7] Bernstein P. A., Hadzilacos P. A., Goodman N., "Concurrency control and recovery in database systems", Addison-Wesley, 1987.

[8] Ceri and Pelagatti, Distributed database systems, McGraw-Hill, 1986.

[9] Addess J., Bloch E., "The implementation of a concurrency control algorithm for a distributed-storage semantic database machine in FCP", Ben-Gurion University, technical report, August, 1989.