Implementation of a Virtual Time Synchronizer for Distributed Databases

Azzedine Boukerche[1], Sajal K. Das[1], Ajoy Datta[2], and Timothy E. LeMaster[2]
[1] Department of Computer Science, University of North Texas, Denton, TX 76203-1366
[2] Department of Computer Science, University of Nevada, Las Vegas, NV 89154-4019

Abstract

The availability of high speed networks and improved microprocessor performance have made it possible to build inexpensive cluster of workstations as an appealing platform for parallel and distributed computing. In this paper, we study the performance of a distributed database system synchronized by virtual time *(VT) mechanism by experimenting on a LAN-connected collection of 12 Sun-SPARCS workstations. Our experimental results demonstrate that the VT synchronization protocol is a viable concurrency control method, and it yields about 25-30% reduction in the response time, when compared with a widely used, multiversion (MV) concurrency control scheme based on timestamps order.*

The reason for comparing with the MV approach is that both virtual time and multiversion protocols are based upon the concept of timestamps, and attain high performance by using multiple version of data objects.

1 Introduction

Despite the fact that distributed databases have been an area of extensive research, to the best of our knowledge, there are no reported results on the performance of virtual time synchronization in parallel and distributed databases, on a cluster of workstations.

Virtual Time is a fundamental concept in optimistic synchronization schemes with many applications in parallel simulations [3], distributed checkpointing and recovery. The Time Warp (TW) based mechanism [7, 8] is a technique to implement virtual time by allowing application programs to cancel previously scheduled events. The main advantages of TW over a conservative protocol is that the former offers the potential for greater exploitation of concurrency and parallelism, and more importantly greater transparency of the synchronization mechanism to the programmer. The transparency is due to the fact that Time Warp is less reliant on application specific information such as dependency among computations or transactions.

Although active research on virtual time has been pursued in the past, the actual performance study of a distributed database system (DDBS) synchronized by virtual time has not gained much attention. We believe that *virtual time* (VT) using a time warp protocol for synchronization is an elegant concept in DDBS. The VT synchronization algorithm has the following advantages over other approaches applied to database systems: (1) Time Warp has been shown to be deadlock-free; (2) the entire set of transactions are not aborted and/or restarted since individual actions within a transaction are rolled back when conflict occurs; (3) it is free from such requirements as predeclaration, sequentialily, or two-phase structure; and (4) it adheres to an object oriented approach to database design in which there is no formal distinction among *transactions*, *data*, and *system* objects. Therefore, a pragmatic question is whether virtual time is really a viable and alternative concurrency control method for DDBS and whether it can provide good performance when implmented on a simple distributed computing platform, like a workstation cluster. This motivates our work.

We implement a distributed database system synchronized by the *virtual time* (VT) mechanism[1], using a LAN-connected collection of 12 Sun SPARCS workstations, study the perfomance of the VT protocl and compare it with a multiversion (MV) concurrency control scheme based on timestamps order. We choose to compare the VT synchronization scheme with the multiversion protocol because they are conceptually similar in the sense that "pseudotime" defined in the MV is similar to virtual time, and also MV is a widely used technique in DDBS. Furthermore, both of these methods exhibit high level of concurrency; do not need deadlock detection and recovery; and are highly efficient unless there are conflicts, which are resolved by undoing operations or aborting transactions.

2 A Distributed Database System (DDBS)

We view a distributed database system as a collection of objects (sequential processes) which execute concurrently and communicate through timestamped messages. Each object is capable of performing both computation and communication, and has an associated *virtual clock* acting as the simulation clock for that object.

There are four major components of a DDBS at each site: transactions, transaction manager (TM), database manager (DM), and data. The transactions are generated from users; the TM supervises interaction between users and the DDBS; while the DM controls access to the data and provides an interface between the user (via the query) and the file system. The DM is also responsible for controlling the simultaneous use of the database and maintaining its integrity and security.

There is no formal distinction between the transaction and data objects. While data objects respond only to *Read*/*Write* messages, the transaction objects receive messages directly from external users. Objects communicate with each other through *Request* and *Response* primitives which send messages. A *Request* either reads or

[1] In the sequel, *virtual time* refers to the time warp based mechanism.

writes a data value, while a *Response* returns a data value. After an object issues a *Request*, it blocks to await a response. The possibility of deadlock exists since two objects within a transaction could request data values from each other at the same time, and each object will be waiting for the other to respond to the *Request*. As will be seen in the next section, the Virtual Time (VT) synchronization protocol prevents deadlock between transactions since all transactions are executed in timestamps order and all such timestamps are unique.

3 Synchronization by MultiVersion Timestamp Ordering

MultiVersion Timestamp Ordering (MV) is a well known concurrency control mechanism to achieve distributed database synchronization. Space limitations preclude us for describing the MV protocol here. Interested readers may wish to consult [2] for a complete description of the MV algorithm we use in our experiments.

4 Virtual Time (VT) Synchronizer

Virtual time can be viewed as a global, one-dimensional temporal coordinate system imposed on a distributed system to measure the computational progress and define synchronization [8]. Two important properties of virtual time are: (1) it may increase or decrease with respect to real time without any restriction on the amount or frequency by which it changes; and (2) it forms partial ordering on the relation "less than."

A manager (or process) may send a message to any other manager at any time, without any restrictions placed upon potential communication paths between processes. No static declarations are required before execution begins. The communication medium is assumed to be reliable but messages are not required to arrive in the order they are sent. Indeed, different transactions occasionally cause an unordered message to be received by an object, when a rollback is used for synchronization. However, instead of rolling back the entire transaction, only certain actions (such as those that are causally connected to the errant message) within the transsction are rolled back. This is where virtual time synchronization, an optimistic approach, fundamentally diverges from a conservative protocol, in which the entire transaction is either aborted and retried or is blocked.

We use the Time Warp based implementation of virtual time which uses several different data structures and control algorithms to produce optimistic processing.

4.1. Data Structures

The data structures include queues for storing messages, a unique process *id*, a local clock, and a permanent value (i.e., the virtual time). The algorithms include transaction managers for generating and submitting requests and database managers for processing requests.

Messages: A transaction request or response is represented by a message which contains the objects' id of the sender and the receiver, the virtual time (also called timestamp), the type, the text of the message, and the sign. In our implementation, the current value of the system clock is used to generate timestamps, ensuring that no two timestamp values are generated during the same tick of the clock. In order to guarantee that transactions have unique timestamps, each virtual time has two components – the timestamp value and the unique process identifier, *Pid*. The type and text components have the query information. The possible message types are *Read, Write, Read-Response*, and *Commit* messages. The sign field holds a bit indicating whether this is an antimessage or not, as required by the rollback mechanism.

Database Manager (DM) Internals: To support the distributed processing and rollback, the DM consists of the following: *Pid*. Local Clock, Permanent Value, Input Queue and Output Queue. The *Pid* is the unique process identifier for a node; the local clock is the current (local) virtual time of an item; the permanent value is the last committed Write value; the input queue stores messages from the virtual past along with the unprocessed messages in manager's virtual future for an item; and the output queue keeps the antimessage copies of all *Read-Response* messages generated and sent by the manager.

The DM processes continuously those messages that have already arrived in its input queue, and stores copies of any messages it sends in an output queue. The local clock is set equal to the timestamp of the message being processed. The output queue and the virtual past portion of the input queue are required to support the rollback mechanism. Whenever a *straggler* message (having timestamp smaller than the local virtual time of the receiving process) does arrive, a rollback occurs which means the Time Warp protocol restores the state of the object, cancels the side effects, and starts simulating the affected objects forward.

Transaction Manager (TM) Internals: The TM consists of three primary components *Pid*, Local Clock, and the Output Queue. The transaction manager adjusts its local virtual time by assigning the local clock equal to the timestamp of the current transaction. The output queue is ordered by timestamps and contains antimessage copies of all $Write$ messages generated and sent by the manager. Since the TM generates the $Read$ and $Write$ requests, no input queue is needed. However, if the manager received requests from an actual user, an Input Queue could be added.

Virtual Clocks: In the virtual time synchronized DDBS, each TM has its own local virtual clock, and each item managed by the DM has also its own virtual clock, which represents the virtual time of the message (or transaction) currently being processed. It acts as a simulation clock.

4.2. Local Control Mechanism

Both the database and transaction managers of the Time Warp perform common such tasks as managing messages, clocks, queues, events, and rollbacks. These are discussed below.

Message Management: Since message passing is used so frequently in a Time Warp mechanism, it is crucial that messages are managed efficiently. Our implmentation uses the following paradigm: When a message for an item arrives at the database manager, it is placed in an input queue with two distinct parts containing respectively unprocessed messages for the virtual future time and the processed messages from the virtual past time. Before a newly arrived message is placed in the input queue, the sign component is examined to determine whether the message is an event message (+ sign) or an antimessage (− sign).

Upon arrival of an event message, the database manager places it in the input queue in the timestamp order. If the message's timestamp is in the manager's virtual past, the manager must rollback to a virtual time earlier than the timestamp. Execution begins at this earlier time, ensuring that all messages are processed in the timestamp

order. The rollback mechanism is invoked to undo all the work done after the timestamp. If the timestamp is in the manager's virtual future, no further action is taken since the message has been queued. Thereafter, the DM resumes processing messages in the input queue.

Upon arrival of an antimessage, the DM also places it in the input queue in the timestamp order. This implies that some transaction managers have rolled back and are undoing work that should not have been done. If the antimessage's timestamp is in the manager's virtual future, the corresponding message found in the virtual future part of the input queue should be removed. If the antimessage's timestamp is in the manager's virtual past, the manager must be rolled back to a virtual time earlier than the antimessage's timestamp. All the work done after the antimessage's timestamp must be undone. In either case, the antimessage will eventually annihilate its corresponding message. Since the only purpose of the antimessage is to seek out its corresponding event message, it is cancelled along with any side effects it may have caused. The timestamps of the antimessage and its corresponding message are the same.

The output queue is used strictly to support the rollback mechanism. If a manager rolls back, antimessages may be sent out to cancel the original messages. The output queue contains timestamps ordered antimessages corresponding to the messages the manager has sent. All the work completed by a manager is conditional and could be rolled back.

As long as the unprocessed messages are in the input queue, the database manager processes them. When a *Read* message is processed, the input queue is searched for the latest, previous $Write$ messages. If no $Write$ is found, the value recorded in the database is used. A response message with this value is generated and sent to the sender object (or process). An antimessage copy is stored in the output queue in case a rollback is required later. In our implementation, a $Write$ message is simply marked as 'processed'. The physical $Write$ operation must be delayed until it can be committed. This action will be discussed in Section 3.4. Recall that the *Read* message are not rolled back since there is no causal relationship among them.

Clock Management: The synchronization of clocks in a virtual time environment is not typical of other DDBS protocols. The local virtual clock for a database manager is allowed to move ahead or behind in virtual time. The clock moves ahead by processing messages in its input queue while it moves behind due to rollback by an incoming message timestamped in the virtual past. The local clock for the TM always moves forward as transactions are completed. The local clocks of all managers are at best, loosely synchronized. Overall, it is expected that the local virtual clock advances ahead in time while occasionally falling behind.

4.3 Rollback Mechanisms

A key aspect of the time warp based scheme is that objects proceed with no delay and roll back whenever out-of-order messages arrive. The appeal of such an optimistic method depends on the efficiency of the rollback mechanism employed.

Database Manager Rollback: The rollback scheme for the DM handles late arrival *Read* messages, $Write$ messages, and $Write$ antimessages, each of which is handled differently. Our approach follows a method based on optimized semantics [1] as opposed to the traditional rollback mechanisms. The basic idea is not to roll back a manager just because a late message arrives. Instead, the message is processed at the current virtual time if the operation involved does not violate the computational correctness. A typical example is a *Read* message arriving late. The DM inserts it into the input queue and treats it as a normal *Read* message. $Write$ message is inserted into the input queue and marked as 'processed'. On the other hand, if a $Write$ antimessage is received, then its corresponding $Write$ from the input queue is removed. In both cases, a *Read-Response* antimessages is sent for any previously affected *Read-Response* messages.

For the late arrival *Read* messages, the receiver process (object) will find the latest, previous $Write$ and return the value recorded. When a $Write$ message is received, any processed *Read* messages with timestamps larger than the late $Write$ must be reprocessed. This is done by cancelling the *Read-Response* messages and sending out a new response message. If a $Write$ antimessage is received, the corresponding $Write$ messages must be annihilated. Any *Read-Response* message which was satisfied with this value must be cancelled too. The net effect after rollback must be to leave the database in a state as if the late message had arrived in order. Antimessages may be sent to accomplish this, but are not always necessary.

Transaction Manager Rollback: The rollback mechanism for the TM handles only Read-Response antimessages because these are received exclusively. There are two cases to consider: (a) If there is no $Write$ message sent, then only the new *Read-Response* values are recorded. Otherwise the *Read-Response* values are recorded, the $Write$ antimessages are transmitted for all $Write$ sent, and the $Write$ messages are sent; and (b) If $Write$ messages have been sent, they need to be cancelled. A new value has just arrived that is assumed to have an affect on the contents of the Write value. After cancellation, new $Write$ messages must be sent. It is assumed the *Read*s have no interdependencies.

4.4 Global Control Mechanism

The success of the virtual time protocol is embedded in the global control issues such as measuring global progress, managing memory, controlling message flow, committing irreversible actions, and taking snapshots. The focal point of all these issues is the *global virtual time* (GVT) computation [8].

Global Clock: Every manager in a virtual time system has its own local clock which may progress forward or backward in time. Since single local clock can be used as an indicator for the global progress of the system [8] a global clock containing the GVT is needed. The GVT serves as an absolute lower bound beyond which no process can roll back, and is typically used to perform fossil collection of obsolete messages and states. It is necessary to compute a GVT estimate periodically in order to do garbage collection. We make use of a simple algorithm where a predefined processor collects the local virtual times of individual clocks and periodically computes the GVT.

Memory Management: Experience with Time Warp over the past years has revealed two fundamental problems with this paradigm, namely memory management resulting from the necessity to save states, and unstable behavior. This problem can be tackled with the help of cancelback protocols and artificial rollback [6, 8].

In our distributed database model, a manager does not

need to store the history from the beginning of the computation if it cannot possibly roll back that far into the past. For efficient memory management, we choose a simple fossil collection algorithm. Any message in the input queue with a timestamp less than the GVT is discarded. If the message is a *Write*, then its timestamp is recorded as a permanent value. Similarly, any message in the output queue with a virtual send time less than the GVT is discarded. The discarded memory is recovered and reallocated to future requests.

Commitment: The nature of rollback in Time Warp restricts a class of operations from being performed immediately. These operations include irreversible actions such as the *Write* operation, in our case. No irreversible action will be allowed to be committed until it can be proven correct. To enforce it, a *Write* operation must be buffered (not committed) until the virtual time associated with it is less than the GVT. The delay due to buffering is not considered as a major drawback of virtual time, but specific applications could suffer from it. One objective of this paper is to determine how long is this delay and whether it could adversely affect the user response time.

5 Performance Study

We conducted experiments on a bus-based, message passing environment consisting of 12 Sun SPARC workstations. The machines were connected into a local area network via ethernet. The ISIS V2.1 toolkit [4] for distributed and fault-tolerant programming was used to implement the MV and VT synchronization protocols. Since the implementation of a distributed database system is more involved than a general distributed simulation environment, where needed, we digressed from the traditional implementations of virtual time mechanisms as discussed below.

5.1 Virtual Time Database Testbed

We used a rollback mechanism for both the transaction and database managers. The TM issues a set of *Read* requests followed by a subset of *Write* requests. Since it issues only one transaction at a time, it could not be rolled back beyond that transaction. We assumed that a *Read-Response* message that is rolled back does the writes, and it does not affect the other members of the read set. Furthermore, once the rollback starts at either managers, it was allowed to finish without any interruption. This diverged from normal rollback, but was easier to implement and oriented towards ISIS message delivery. The history information stored for rollback included all entries in the output queue and a portion of the input queue. One structure not needed in this implementation of virtual time was the state queue which contains the checkpoint information.

In our experiments, some assumptions were made about the nature of the transactions and their submissions since they have an impact on the performance of the DDBS. These models were similar to the ones suggested in [2, 5, 9]. No blind writes were allowed. That is, a transaction could not issue a *Write* operation to a database item without first reading it. The transaction manager generated the user requests randomly and issued them to the database managers. Only *Read/Write* synchronization were considered. The control variables for this model included the transaction/database sizes, the inter-transaction delay, and the update probability.

A simple database model was used. Each database manager was responsible for a specified number of items. Whenever a transaction requested a *Read* or *Write* access to an item, it was delayed 50 milliseconds to simulate the physical I/O operations.

5.2 Performance Metrics

Memory requirements, throughput, and execution time contribute to the overall cost of a virtual time system. Each of these parameters can be manipulated by the virtual time designer to tune the system for an optimal performance. The *response* time is defined as the average amount of clock time per processor required for a transaction manager to successfully submit a fixed number of transactions. The *throughput* is defined as $(P * N_t)/(\text{total response time of all processors})$, where P is the number of processors used, and N_t is the number of transactions. In our experiments, $N_t = 500$.

The most critical parameter is the frequency of the GVT computation. A high frequency leads to better response time and memory management, but requires more processing time and thus increases message traffic. On the other hand, a low frequency has the opposite effect. In our experiments , the GVT for every test run was individually tuned to achieve the maximum performance.

5.3 Experimental Results

We made use of several database/transaction sizes, inter-transaction delays and varied the number of processors from 4 to 12. The following experimental data were obtained by averaging several trial runs. Every test run required between 8 and 50 minutes of actual time to complete all experiments. The average length of the experiments was 24 minutes. Note that the results presented here are tempered with the overhead required to execute ISIS.

We first analyzed the effect of the transaction size on the performance of the DDBS synchronized by virtual time. To this end, several simulation tests were run on $P = 8$ processors. The transaction size was varied from 5% to 10% of the database size, and the inter-transaction delay was set to zero. All *Read* requests were updated with probability 25%. Figure 3 portrays the response time of the simulation of DDBS for both VT and MV protocols. We observe that the virtual time scheme exhibits a good response time when the transaction size is larger than 8.5%. The multiversion scheme performs better than VT only for a small transaction size, and this is due to the intrinsic properties of virtual time. Thus, undoing work as in the VT scheme has more appeal than aborting it as in the MV scheme since aborting is a more drastic and costly operation.

Figure 4 depicts the throughput for both the VT and MV synchronization protocols. The results demonstrate that good (resp. poor) response time is strongly affected by high (or low) overhead. In both schemes, the throughput decreases with the increase in the transaction size. However, the VT scheme exhibits a larger throughput since the recoverability condition in this scheme is easier to enforce than the serializability condition in the MV scheme.

Next, we analyzed the effect of the database size on the performance (response time and throughput) of the DDBS, when 8 processors were used. We varied the database size from 1 to 20, and the transaction size was set to 7% of the database size. All *Read* requests were updated with a probability 25%. The Inter-transaction delay was set to zero. Figure 5 presents the response time as a function of database sizes for both synchronizers VT and MV. They exhibit about the same response time for database size less than 12 beyond which the VT scheme responded faster

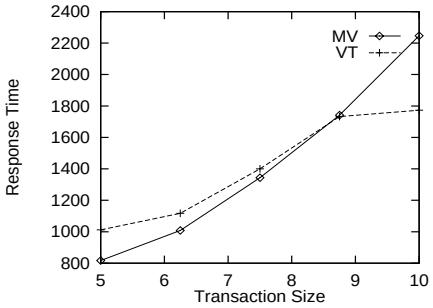

Fig. 3 Response Time Vs. Transaction Size

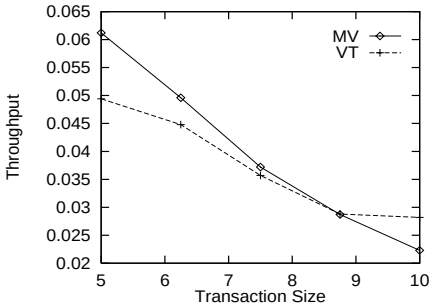

Fig. 4 Throughput Vs. Transaction Size

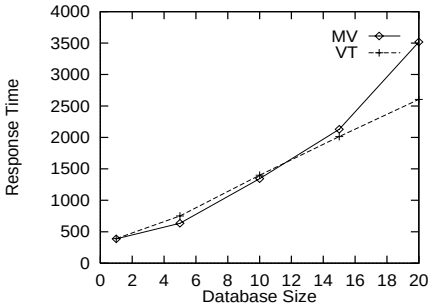

Fig. 5 Response Time Vs. Database Size

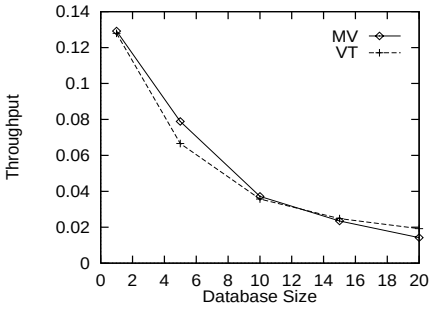

Fig. 6 Throughput Vs. Database Size

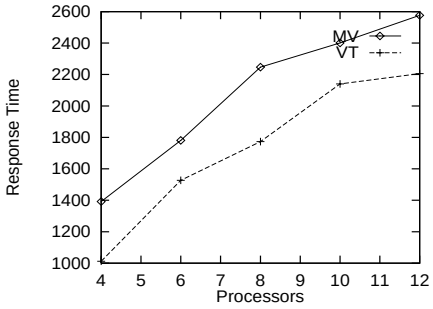

Fig. 7 Response Time Vs. Nbr. of Processors

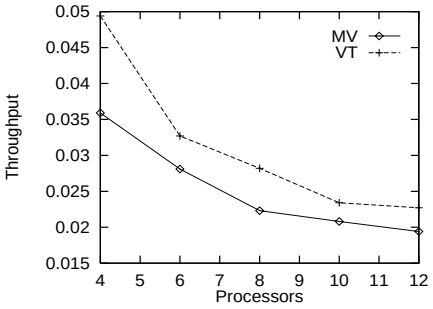

Fig. 8 Throughput Vs. Nbr. of Processors

than MV. We observe about 25% reduction in the response time for a database size of 20.

Figure 6 presents the througput which mirror the curves obtained for the response time. The throughput significantly decreases as the database size is increased. Finally, we ran experiments varying the number of processors from 4 to 12. The obatined results for the response time and throughput are depicted in Figures 7 and 8. As expected, with the increase in the number of processors, the response time increases for both the synchronization protocols. Furthermore, the VT scheme performs better than the MV scheme. For example, 25-30% reduction in the response time is observed with the VT scheme as compared with the MV scheme.

6 Conclusions

Workstation clusters in conjunction with high-speed interconnection networks offer a cost-effective and scalable alternative to monolithic supercomputers. This paper focuses on the performance of distributed database concurrency control schemes on a network of workstations. The experimental results indicate that a careful implementation of the virtual time (VT) protocol using time warp synchronization mechanism is a viable technique, and it does improve the performance (response time and throughput) of distributed databases. We study the effect of transaction and database sizes, and inter-transaction delay on the overall performance of the DDBS. The results demonstrate that the VT protocol outperforms the MV (multiversion timestamp ordering) protocol as the transaction size increases. We also observe about 25-30% reduction in the response time using the VT method over the MV one. This is due to the fact that the recoverability operation is a much easier condition to enforce than the serializability, and an undoing operation is less costly than an abortion. Another observation is that adding inter-transaction delay to a VT scheme only degrades its performance in comparing with the MV scheme. As regards to scalability, the VT outperforms the MV as the number of processors is increase,

References

[1] Badrainath, and K. Ramamritham, "Semantic-Based Concurrency Control: Beyond Computativity", *Trans. on Database Systems*, 17:1, 1992.

[2] P.A. Bernstein and N.Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[3] A. Boukerche, "Time Management in Parallel Simulation" In Preparation.

[4] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood, *The ISIS System Manual, Version 2.1*, The ISIS Project, 1990.

[5] R. Elmassri, S. Navathe "Fundamentals of Database Systems", Addison Wesley, 1994.

[6] A. Gafni, "Rollback mechanisms for optimistic distributed simulation systems," *Proceedings of the SCS Multiconference on Distributed Simulation*, 19:3, 1988, pp. 61-67.

[7] D. Jefferson and A. Motro, "The Time Warp Mechanism for Database Concurrency Control," *International Conference on Data Engineering*, February 1986, pp. 474-481.

[8] D.R. Jefferson, "Virtual Time," *ACM Trans. on Programming Lang. and Syst.*, 7:3, 1985, pp. 404-425.

[9] C, Hass and Weikum, "A Performance Evaluation of Multi-Level Transaction Management", in Proc. of Very Large Database (VLDB) Conf., 1991.