Issues and Approaches to Design of Real-Time Database Systems

Mukesh Singhal
Dept. of Computer & Information Science
The Ohio State University
Columbus, OH 43210

Abstract

Real-time database systems support applications which have severe performance constraints such as fast response time and continued operation in the face of catastrophic failures. Real-time database systems are still in the state of infancy, and issues and alternatives in their design are not very well explored. In this paper, we discuss issues in the design of real-time database systems and discuss different alternatives for resolving these issues. We discuss the aspects in which requirements and design issues of real-time database systems differ from those of conventional database systems. We discuss two approaches to design real-time database systems, viz., main memory resident databases and design by trading a feature (like serializability).

We also discuss requirements in the design of real-time distributed database systems, and specifically discuss issues in the design of concurrency control and crash recovery. It is felt that long communication delays may be a factor in limiting the performance of real-time distributed database systems. We present a concurrency control algorithm for real-time distributed database systems whose performance is not limited by communication delays.

Index terms :- concurrency control, conflicts, main memory databases, performance, real-time database systems, replicated database systems, timestamp, transactions, update.

1. Introduction: Real-Time Databases

A typical real-time system consists of two closely couple subsystems: a controlled process and a controller. The *controlled process* is the underlying application, and the *controller* is a computer which monitors the status of the controlled process as well as supplies it with appropriate driving signals. In real-time systems, supported applications have stringent timing constraints; that is, when the application reaches certain state, some actions must be taken instantaneously to avert a disaster. Examples of such applications are command and control, weapon systems, process control, automated manufacturing, mission control, surveillance, and stock exchange. In such systems, the controller must respond to changes in the system state within a short period of time to produce useful results or to avert severe consequences.

With ever increasing volume of information being handled by real-time systems over the last several years, these systems are now being equipped with database systems for

efficient storage and manipulation of information. This has resulted in the marriage of real-time systems and database systems, and the resulting systems are called *real-time database systems*, as the underlying database system should be able to provide real-time response.

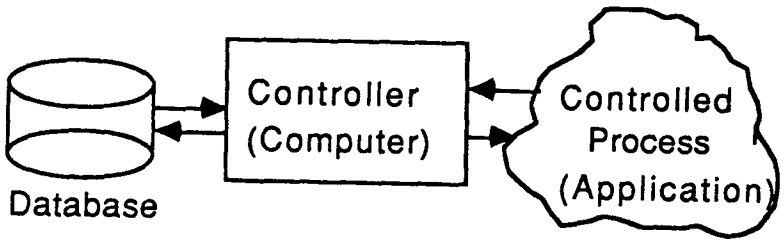

Figure 1. A Real-Time Database System

For example, if a radar is tracking an enemy aircraft and is continuously updating the information about its location, speed, and direction in the database, and if an anti-aircraft gun is using this information (by retrieving it from the database) to aim at the aircraft, then update and retrieval operations on the database must be almost instantaneous. Otherwise, the gun will always trail the aircraft.

Deficiencies of Conventional Databases: Although conventional database systems provide efficient ways to store data and user friendly interface, they rely on secondary storage to store the database. In conventional database systems, transaction processing requires accessing database stored on the secondary storage; thus, transaction response time is limited by latency delays (in disks) and rotational delays (in drums), which can be of the order of 30 ms. Still these database are fast enough for traditional applications because a response time of few seconds is often tolerable to a human user. However, conventional database systems may not be able to provide a response which is fast enough for several real-time applications (these applications require response within hundreds of micro seconds). Consequently, requirements and design issues of real-time database systems widely differ from those of conventional database systems which do not have stringent constraints on response time.

In the design of conventional database systems, main emphasis is on maintaining correctness (integrity) of the data, keeping the cost of the system low, and providing a human friendly interface. In real-time database systems, the emphasis is somewhat reversed. In conventional database systems, algorithms are designed to minimize accesses to disk (I/O delays) while in main memory database systems algorithms should be designed to minimize memory space and amount of computation (because database access time is no longer a bottleneck).

In the design of real-time database systems, the primary objective is to attain extremely fast response for updates and queries. A natural question is how should we redesign conventional database systems so that its performance is acceptable for real-time applications. One way is to invent a radically different architecture for a database system which can provide fast response (about 10 times faster over the conventional database

systems). However, this will tantamount to a breakthrough in database technology and is not expected to happen in near future. Second alternative is to tinker with conventional database systems so that their performance is significantly improved. There are two approaches to achieve high performance in this way. First, by putting more dollars; e.g., we can replace the bottleneck device (e.g., a disk) by a high speed version. Second, at the cost of some (desired) features of conventional database systems. For example, we can trade serializability of transaction execution to increase the performance. In this paper, we discuss issues and approaches to design of real-time database systems.

2. Performance Enhancement by High Cost

2.1. Main Memory Database Systems

Since database operations are mostly I/O bound, the main hurdle in improving the performance of conventional database system is intolerable disk access delays (about 30 ms for each access). Therefore, one primary issue is to somehow eliminate disk access delays from database access. We can eliminate (or substantially reduce) delays in accessing database by having a large main memory and storing a large portion of the database, or the entire database, in the main memory. Depending upon whether we store entire database or a large portion of it in the main memory, such database systems are called main memory or large main memory database systems, respectively, [2].

In main memory database database systems, accesses to secondary storage are altogether eliminated. Large main memory database systems substantially enhance the performance of conventional database systems because by judiciously prefetching a portion of the database in the main memory, we can have most of the data accesses satisfied by a main memory read. Thus, I/O delays are significantly reduced giving extremely fast response (or extremely high processing rate) to user transactions. Due to advancements in integrated circuits technology, the chip densities of semiconductor memories have increased dramatically and the cost of main memory has declined substantially over the last decade (and continue to do so). Consequently, main memory databases have become economically feasible and are expected to become common in near future.

2.2. Design Issues in Main Memory Database Systems

Although main memory database systems are capable of providing response which is fast enough for several real-time applications, they introduce some problems and design issues of their own [2]. The most critical of these is the problem of crash recovery. In addition, design of main memory database systems entails developing new schemes for physical data organization, query optimization, and concurrency control which can exploit main memory residency of the database to enhance the performance.

Crash Recovery

The problem of crash recovery becomes challenging in main memory database systems because a system crash (e.g., power failure) can destroy the contents of main memory (semiconductor main memories are volatile). Consequently, a main memory database system still requires a disk to act as stable storage to provide a back up copy for the database. Since in a main memory database system, an update is not propagated

immediately to the disk version of the database, the state of database in main memory may differ from the state of database on the disk. A recovery technique of conventional database systems which recovers the system from a crash by first loading the database from disk and then applying transaction log to it to bring it to up-to-date state may be intolerably slow for real-time applications. Therefore, one of the biggest challenges of the design of main memory database systems is to develop crash recovery techniques which can quickly restart the system after a failure and to insure that transactions are atomic and durable despite that the main memory is volatile.

Some techniques for crash recovery in main memory database systems have been proposed in [10, 13, 16, 21]. Almost all the recovery techniques assume that a small portion of main memory can be made stable (can be made to survive system power failures) by using battery-backup RAM modules and use this portion of the main memory to store transaction log. Rather than using simple method for data recovery (i.e., load a previous database state and execute the log), recovery techniques for main memory database systems should use clever mechanisms to speed up the recovery process. For examples, Son uses "non-interfering checkpointing" and "log compression" techniques [21], DeWitt et al. use "overlapped checkpointing" and "reduced logs" [6] to reduce recovery time, and Lehman and Carey [13] reduce recovery time by loading the working sets of the current transactions in the main memory and letting the rest of the database be updated and loaded by a background process (a kind of incremental recovery [1] which is based upon working sets).

Data Organization, Access Methods, and Query Processing

In conventional database systems, data organization, access methods, and query processing algorithms are designed to reduce I/O traffic at the expense of CPU time. (This is because in conventional database systems main memory can hold a small fraction of the entire database and processing a query may require/generate large amount of intermediate data.) Access methods and query processing algorithms which work efficiently for conventional (disk-based) database systems may not be efficient on main memory database systems [2]. In main memory database systems, efficient usage of space is very important [2]. Therefore, access methods and query processing algorithms must optimize memory space as well as processing time. In main memory database systems, query processing must exploit direct addressability (random access) of data in the main memory. Several studies have investigated suitability of access methods and join algorithms for large main memory database systems using analytic and simulation techniques [6, 17]. It has been found that access methods and join algorithms which work efficiently for conventional database systems may not be suitable for large main memory database systems.

In [13], Lehman and Carey have studied the performance of several algorithms for selection, join, and projection operations in main memory database systems from the point of view of query processing. Their study reveals that T trees (a new data structure which combines the properties of AVL and B trees) provides good performance for selection and join operations in main memory database systems. Also, hashing techniques were found to fare very well for projection operations in main memory database systems. In conclusion to their study, query optimization in main memory database systems should be simpler than conventional database systems. This is because in main memory database

systems, the notions of clustering and projection size reduction are factored out.

Concurrency Control

The issue of concurrency control is relatively unexplored for real-time database systems. Concurrency control degrades the performance by blocking a transaction or by restarting it. In main memory database systems, due to absence of disk access delays, blocking delays and transaction execution time are likely to be small. However, this does not help in identifying which of the known concurrency control algorithms will be suitable in the environments of main memory database systems.

Since in main memory database systems conflicts are resolved quickly (because blocking delays are small) and the processor is likely to be a bottleneck, a coarse level of granularity for data objects (locking) is acceptable. Coarse granularity tends to increase conflicts but small blocking delays subdue the effect of them, and coarse granularity can substantially reduce overhead on the processor. Moreover, since memory space is at premium, multiversion concurrency control algorithms and the concurrency control algorithms which create temporary data objects (e.g., optimistic algorithms) may not be attractive in main memory database systems, unless we use storage saving techniques such as differential files.

3. Performance Enhancement by Trading a Feature

We can enhance the performance of conventional database systems by trading some feature for increased performance. There are two obvious candidate features which can be traded for the performance: user friendly interface and serializability of transaction execution. Since in real-time database systems the database is primarily interfaced to an application (such as an avionic system or an automated manufacturing process) and since they are used for special purposes, we can trade friendliness of user interface to enhance the performance. That is, we can choose a data model and a data manipulation language which result in efficient implementation and fast processing rather than increased friendliness to naive human users.

Sacrificing Serializability

Database systems exercise concurrency control to guarantee correctness of the database (i.e., to maintain database consistency) in presence of concurrent database access by several users. Serializability is widely used as a correctness criterion for concurrency control in database systems [15]. Execution of a set of concurrent transactions is serializable if their effect is the same as though they were executed serially.

Enforcement of serializability degrades the performance (i.e., lengthens transaction response time) because serializability is achieved by transaction blocking or transaction restarts. Therefore, one way to achieve better performance is to sacrifice serializability, that is, to sacrifice database consistency/correctness. (Sacrificing serializability may also increase availability under failure conditions, e.g., in the face of network partitioning [5,7]). It is generally felt that serializability as a correctness criterion is too stiff, and there are several applications where non-serializable schedules are correct or are acceptable. In some applications it may be necessary to get information very quickly even though it may be partially incorrect rather than getting correct information when it is too

late.

In several cases, a priori knowledge about the underlying application may be used to relax the condition of serializability to give better performance. As an example, suppose a real-time database system is controlling a chemical plant. There is an update transaction which periodically (or whenever values of data objects change substantially) updates temperature, pressure, and velocity. There is another transaction which reads temperature, pressure, and velocity and takes appropriate actions (like opening a valve, changing temperature setting, ringing an alarm, etc.) if any of these parameters are found out of range. In this example, serializability is too restrictive because the second transaction can read any of three parameters from the database (and take appropriate actions based upon the value) even though the first transaction has not finished with updating all the three parameters.

Although it is well-known that several non-serializable schedules maintain database consistency and are acceptable in various applications, it is very difficult to design a general-purpose concurrency control algorithm which will allow non-serializable but semantically consistent schedules. This is because if a concurrency control algorithm has to allow non-serializable but semantically consistent schedules, then the only choice is to execute transactions and verify consistency assertions on the fly. That is, to verify that values read by transactions and final database state satisfy consistency assertions. Such execution, if not impossible, has the following two problems: First, it may not always be possible to enumerate all the consistency assertions for a database system; Second, checking if consistency assertions are satisfied by transaction execution may be terribly slow (even slower than meeting the condition of serializability). Therefore, serializability seems to be the only criterion which can be efficiently enforced algorithmically to maintain database consistency irrespective of the semantics of the underlying database system.

However, semantic knowledge of applications may be used to design concurrency control algorithms which allow non-serializable but consistent schedules. Since most real-time database systems are used for highly specialized applications, semantic properties of transactions and data can be known a priori. Moreover, since types of transactions are limited in such applications, it is possible to carry out semantic analysis of transactions to determine what (non-serializable) interleavings will maintain database consistency. In [9], Garcia-Molina has outlined a method which uses semantic knowledge of applications to determine non-serializable schedules which maintain database consistency. The method exploits the semantics of transactions to determine for every transaction a set of transactions which can be interleaved with it without violating database consistency. To understand the basic idea, consider a simple database consisting of two accounts X and Y. A transaction T1 moves $100 from X to Y and transaction T2 moves $200 from Y to X. That is,

T1 = {X:=X-100; Y:=Y+100}
T2 = {Y:=Y-200; X:=X+200}

Actions of T1 and T2 can be interleaved in any fashion without violating database consistency (even though resulting schedule is not serializable). For example, if initially X=1000 and Y=2000, then an interleaving X:=X-100, Y:=Y-200, Y:=Y+100, X:=X+200 leaves X=1100 and Y=1900. Though the schedule is not serializable (becaues T1 -- T2 and T2 -- T1), it leaves database in a consistent state. If a transaction T3 credits interest

to both these accounts and transaction T4 prepares monthly statements for both these accounts, then T3 and T4 can not be arbitrarily interleaved with T1 and T2.

In applications where it is more important to get (partially) incorrect information quickly than to wait for correct information, a better approach may be not to exercise any concurrency control and periodically examine the database for inconsistencies and restore it to a consistent state. However, removal of inconsistencies from a database can be a difficult task. It may require keeping history of transaction execution and rolling back of some transactions which have caused potential inconsistencies. Rolling back of transactions is not very practical in real-time applications because most actions (such as firing a missile) are not reversible. Therefore, backward recovery is not very appealing and forward recovery seems to be a viable approach for recovery from inconsistencies.

An argument which supports sacrificing serializability (or sometimes even database consistency) to enhance performance in real-time database systems is the fact that data is often short-lived in several real-time applications. For example, in a weapon system where a gun is trying to shoot down an aircraft, data is useful only during the time the aircraft is visible. Because of the short life of data, inconsistency does not spread too much over the database and content of the database does not get corrupted very much. (Once an inconsistency is injected in the database, it spreads with time because transactions read inconsistent values and write back inconsistent values.) More work still need be done in the direction of formalization of the trade offs between database correctness and performance, and new algorithms need to be developed to materialize these trade offs.

4. Real-Time Distributed Database Systems

Falling cost of hardware and advances in communication technology over the last decade have triggered considerable interests in distributed database systems. In such systems, data objects are spread over a collection of computers/sites which are connected via a communication network. A distributed database system offers several advantages over a single-site database system, such as data and program sharing, higher system throughput, higher system availability, load sharing, and easy expandability.

Distributed database systems are of great importance to several real-time applications (e.g., military exercises and weapon systems) because such applications require severe performance constraints such as fast response time and continued operation in the face of catastrophic failures. (Sometimes these applications naturally evolve into distributed database systems because they are inherently distributed.) Replicated database systems are of special interests to real-time applications because they possess several desirable features: enhanced reliability, improved responsiveness, no directory management, and easier load balancing. Enhanced reliability results because a site crash or network partitioning does not prevent access to data objects and, in general, does not stop transaction processing under these failures. Because of the enhanced reliability, replicated database systems are highly desirable in hostile environments (e.g., weapon systems) where site crashes or network partitionings are inevitable and continued operation under these failures is very crucial. Replicated database systems provide extremely fast response to queries because all the data objects are available locally at every site (There is no need to access data remotely which eliminates lengthy communication delays).

However, replication of data has its cost: In addition to added storage cost, updates in replicated databases are expensive because synchronization of multiple copy updates is complex and requires extensive communication. Despite these overheads, replicated database systems are suitable for real-time distributed database systems because they have several desired features. Therefore, in this paper, we will consider only replicated database systems.

5. Design Issues in Real-Time Distributed Database Systems

Since a real-time distributed database system is collection of cooperating autonomous single-site real-time database systems, the main issue in the design of real-time distributed database systems is how to make several autonomous database systems work collectively so that a high level of performance is obtained [20]. That is, given that the database at each site can provide real-time response, the issue is how to design concurrency control, query optimizer, deadlock detection/handling scheme, etc. so that a desired level of performance is obtained for the real-time distributed database system. Since execution of a transaction requires message exchanges (for information exchange and control) over the communication network, speed of the communication network will limit the performance of a real-time distributed database system (as the speed of the disk limits the performance a single-site database system). Therefore, a major issue in the design of real-time distributed database systems is to reduce need for intersite communications.

5.1. Issues in Fault Tolerance and Reliability

Fault tolerance and reliability are highly desirable in many real-time applications because in these applications, continued operation under catastrophic failures and quick recovery from failures is very crucial. Often the requirements of "fault tolerance and high reliability" and "fast response" are conflicting because fault tolerance and high reliability involve doing redundant processing and/or exchanging extra messages. In this section, we discuss issues involved in the design of recovery techniques for site crash and network partitioning.

Site Recovery

In a replicated database system, when a site recovers from a crash, its database must be restored to an up-to-date state and must be consistently merged with the rest of the sites. Most recovery techniques maintain redundant information about the database or history of transaction execution and recover a site by restoring its database to a previous consistent state. Some examples of redundant information are *audit-trail or log, checkpoint, and differential files*. The recovery in SDD-1 is based on the notion of *spoolers [11]*, where in case of a site crash, all messages directed to the crashed site are buffered at its spooler sites. When a site recovers, it unspools all messages buffered for it from its spooler sites. In real-time applications, it is necessary that a crashed site be recovered quickly and/or a site be (partially) functional when the recovery is taking place. DDM [4] is an example which attains these features to some extent: it improves data availability by using *incremental recovery [1]* and reduces intersite data transfer, in case of failures, using log-based recovery mechanism.

Network Partitioning

Due to the requirement of high availability, real-time applications require that in the face of a network partitioning, different partitions continue with transaction processing. However, if we permit transaction processing in presence of a network partitioning, the state of the databases in different partitions is likely to diverge with time and the consistency of the database is likely to be violated. Therefore, when the database system recovers from a partitioning, database state in different partitions must be reconciled to a common value. This can be achieved by having every site keep a log or audit trail, and when a database recovers a partitioning, having databases in different partitions to reconcile inconsistencies by merging the journals of different partitions to get a common journal and applying it to all the sites of the partitions being merged. In real-time applications, we require techniques which can achieve this task quickly and allow partial functionality during the recovery. Several techniques for consistently integrating partitions of a replicated database system are described in [3, 5].

5.2. Issues in Concurrency Control

Since concurrency control is a fundamental problem in database systems, a real-time distributed database system calls for a high performance (i.e., short response time and/or high throughput) concurrency control algorithm. Concurrency control algorithms degrade performance because they introduce blocking whenever there are conflicts. In existing concurrency control algorithms blocking delays seriously limit the performance because blocking delays depend upon the speed of communication network. This is because global synchronization of transactions requires exchange of messages over the communication network. Therefore, development of high performance concurrency control algorithms requires reducing the blocking delays due to conflicts. Since conflicts among transactions are inherent to a system, we require radically different techniques for concurrency control (where blocking delays due to conflicts do not dependent upon the speed of communication network).

In previous concurrency control algorithms for replicated databases, the performance degrades because blocking delays due to conflicts are sensitive to communication delays and the probability of conflicts among updates. Blocking delays are sensitive to these parameters because update execution is asymmetric in these algorithms: "When a user submits an update to a site, the site performs synchronization, executes the update, writes the computed values into its copy of the database, and broadcasts the computed values to all other sites; On receipt of these values, other sites write them into their copies of the database". Asymmetry arises because only one site completely executes an update and other sites just execute update's write action. These algorithms are also referred to as *semi-distributed algorithms* due to obvious reasons [18].

Informally, sensitivity of blocking delays to communication delays arises in these algorithms because at a site an update may remain blocked until the site receives computed values from other sites. Since values from remote sites come via communication network, duration for which an update remains blocked depends upon communication network delays. Sensitivity of blocking delays to the probability of conflicts arises because higher the conflicts the more often an update waits resulting in larger blocking delays.

In next section, we present a new technique for concurrency control in replicated database systems where the blocking delays due to conflicts depend upon the speeds of disk (I/O device) and CPU rather than the speed of communication network (the technique was first proposed in [18]). Since disk and CPU are usually much faster than the communication network, the new approach has potential for substantial improvement in the performance. Also, since I/O delays are substantially reduced in real-time database system (due to main memory residency of the database), the proposed approach becomes even more attractive.

6. A High Performance Concurrency Control Algorithm

In this section, we show that by adapting a different approach (symmetric) to update execution, we can make blocking delays independent of communication delays and the probability of conflicts. In this approach, update execution is replicated at every site; that is, every site completely executes every update making it a symmetric or fully-distributed approach.

6.1. Preliminaries

A database consists of M data objects which are numbered from 1 to M. A user modifies the database by executing update transactions. An *update* U is defined by a 3-tuple U=(rs, ws, f), where (a) rs $\subseteq$ {1,2,3,...,M} indicates the data objects whose values are read by U and is referred to as the *readset* of U, (b) ws $\subseteq$ {1,2,3,...,M} indicates the data objects that are written by U and is referred to as the *writeset* of U, and (c) f is a function that models the computation done by U. We assume that the function f maintains internal consistency of a database. We denote readset and writeset of update U by RS(U) and WS(U), respectively.

For any two updates U^1 and U^2, U^1 is said to have r-w, w-r, or w-w conflict with U^2 if $RS(U^1) \cap WS(U^2) \neq \phi$, $WS(U^1) \cap RS(U^2) \neq \phi$, or $WS(U^1) \cap WS(U^2) \neq \phi$, respectively. Also, updates U^1 and U^2 are said to *conflict* if at least one of these conflicts exists between them. A site S_j executes an update U^i in the following manner: (a) S_j reads the data objects $RS(U^i)$, (b) S_j computes values for data objects in $WS(U^i)$, and (c) S_j writes the computed values onto the data objects in $WS(U^i)$ of its database copy. The readset and the writeset of an update are known at the time of its inception in the system. Timestamps are generated according to Lamport's scheme [12]. The timestamp of an update U is denoted by TS(U).

6.2. Basic Idea: Symmetric Approach to Update Execution

We propose the following approach to updates execution in a replicated database system: When a site receives an update from a user, it assigns the update a timestamp and transports the update and its timestamp to all other sites. After it, every site executes that update completely without any exchange of computed values of data objects. That is, every site performs synchronization and executes the read, compute, and write actions of every update. Thus, each site has equal involvement in executing every update which makes it a symmetric or fully-distributed approach to update execution.

In the fully-distributed approach to update execution, one can use any optimistic or pessimistic technique for synchronization. In the algorithm which is presented in this

paper, we embed the synchronization scheme of the algorithm in [19] into the fully-distributed approach to update execution. Since algorithm in [19] is based on a semi-distributed approach to update execution, we refer to it as the SDA (Semi-Distributed Algorithm) henceforth, and since the algorithm presented in this paper is based on the fully-distributed approach to update execution, we refer to it as FDA (Fully-Distributed Algorithm) henceforth in this paper.

Next, we give an informal description of the FDA (A formal description of the algorithm and a proof of its correctness are given in [18].) and present an analysis of the blocking delays which shows how in symmetric approach to update execution, blocking delays become insensitive to communication delays and the probability of conflicts.

6.3. Description of the Algorithm (FDA)

When a site S_i receives an update U=(rs, ws, f) from a user, it takes the following sequence of actions: it updates its clock and assigns a timestamp, say ts, to U; it sends an UPDATE(rs, ws, f, ts) message to all other sites; and it saves the update with its timestamp. A site S_j responds to the UPDATE(rs, ws, f, ts) message in the following manner: it updates its clock; it returns a REPLY(ts1) message to S_i (ts1 is the updated timestamp at S_j); and it saves (rs, ws, f, ts).

A site S_k, k=1,2,...N, executes the read and the compute actions of U after (i) it has received a message with a timestamp larger than 'ts' from all other sites and (ii) write action of all updates which have their timestamps smaller than 'ts' and which intend to modify some data objects in 'rs', have been executed at S_k. A site writes the computed values for U onto the data objects in 'ws' of its copy of the database after (iii) read action of all updates which have their timestamp smaller than 'ts' and intend to read some data objects in 'ws' and (iv) write action of all updates which have their timestamp smaller than 'ts' and intend to modify some data object in 'ws' have been executed at that site.

6.4. Analysis of Blocking Delays

Since in fully-distributed approach, every site computes the values to be written by an update, an important aspect of this approach is that a site does not have to wait for computed values from other sites. Because of the absence of wait for computed values, there is a decoupling effect among the sites. As we discuss next, decoupling makes blocking delays insensitive to communication delays and probability of conflicts. Also, it significantly impacts the update response time. We explain how decoupling makes blocking delays independent of communication delays by comparing the blocking delays due to r-w conflict in the SDA [19] and FDA. (The same argument holds for w-r and w-w conflicts.) Since both algorithms use the same synchronization rules, any difference in their performance is attributed to their different styles of update execution.

Consider an update U at site S_i for which the condition (i) holds, but the condition (ii) does not hold. This means U intends to read some data objects whose value is not up-to-date as of TS(U) at S_i, i.e., a set of updates $\{U^1, U^2,..., U^n\}$ with their timestamps smaller than TS(U) have yet to modify some data objects in RS(U) at site S_i. Update U remains blocked until write action of every update in the set $\{U^1, U^2,..., U^n\}$ is executed at site S_i. The time interval for which U remains blocked is the *blocking delay* of U.

For the sake of simplicity, we assume that update arrival rate is the same at each site. Therefore, with probability (N-1)/N an update in the set $\{U^1, U^2,..., U^n\}$ is remote to site S_i, i.e., it originated at a site other than S_i. (N is the total number of sites in the system.)

Blocking Delays for SDA

In SDA, with probability (N-1)/N values for an update in the set $\{U^1, U^2,..., U^n\}$ come from a remote site. That is, a site (other than S_i) computes the values and those values propagate through the communication network before they reach S_i. At S_i, U remains blocked at least until S_i has committed those received values into its database. Consequently, after condition (i) has been satisfied, additional time required for condition (ii) to hold for update U depends upon communication delays (TM). Therefore, for SDA,

Blocking Delays = f(Comm. delays, Prob. of conflicts, disk speed, CPU speed)

$$\text{ResponseTime}=2*TM+T_{cpu}+2*T_{disk}+K1*f(...) \quad ...(1)$$

T_{cpu} and T_{disk} are response times at CPU and disk, respectively, and K1 is a constant. It has been shown analytically in [18] that $f(...) \alpha \frac{TM^2}{TM+W_w}$, ($W_w$ is blocking delays due to w-w conflict), which indicates that blocking delay f(...) is sensitive to TM in quadratic manner. When TM is increased, it causes blocking delays due to conflicts to increase ensuing an increase in the probability of conflicts which in turn elongates blocking delays further. It is due to this self-feeding nature of the blocking delays that f(...) is non-linear with respect to TM which in turn causes update response time to rise non-linearly with communication delays. This phenomenon, in general, occurs in all previous algorithms because a site waits for computed values from other sites.

Blocking Delays in FDA

In FDA, however, after condition (i) holds for update U, it no longer waits for a message from remote sites for condition (ii) to hold. This is because site S_i has received all updates up-to timestamp TS(U) from other sites and can compute values of the data objects in WS(U) locally by first executing all the conflicting updates in the set $\{U^1, U^2,..., U^n\}$, then executing U. Such computation only requires access to CPU and secondary storage (disk) at S_i. Therefore, when condition (i) holds, amount of time elapsed before condition (ii) holds (i.e., blocking delay due to conflicts) depends upon the speeds of CPU and disk rather than on the speed of communication network. Therefore, for FDA,

Blocking Delays = g(disk speed, CPU speed)

$$\text{ResponseTime}=2*TM+T_{cpu}+2*T_{disk}+K2*g(..) \quad ...(2)$$

Note that in FDA, g(..) does not depend upon probability of conflicts because when once condition (i) holds for U at S_i (i.e., S_i has received all updates up-to timestamp TS(U) from other sites), it can compute values of the data objects in WS(U) locally, and for which it has to wait utmost until S_i has executed all updates with timestamp smaller than TS(U). Since execution of all such updates requires access to only local resources at S_i, on the average U waits for fixed amount of time regardless of its conflicts with other

updates. In a replicated database system of N sites, if update arrival rate at each site is λ and CPU and disk speeds are μ_{CPU} and μ_{disk}, then it can be shown that,

$$T_{cpu} + 2*T_{disk} + K2 * g(..) = \frac{1}{\mu_{CPU}-\lambda*N} + \frac{2}{\mu_{disk}-\lambda*N}.$$

Thus, in symmetric approach to update execution, we factor out communication delays from the blocking delays of updates, and response time is proportional to TM.

6.5. A Remark on the Performance

We conducted a performance study in [18], where we compared the performance of FDA with the performance of Garcia-Molina's centralized locking algorithm [8], Milenkovic's pessimistic algorithm [14], and Singhal-Agrawala algorithm [19]. The results of the performance study show that FDA has much better update response time and throughput characteristics as compared to these algorithms (unless disk is very slow and/or message propagation delay is very small). FDA performs better (i.e., has smaller response time and higher throughput) than these algorithms because in it blocking delays depend upon the speeds of CPU and disk rather than on the speed of communication network.

An interesting finding of the performance study is that the maximum throughput of the centralized locking algorithm [8], Milenkovic's pessimistic algorithm [14], and Singhal-Agrawala algorithm [19] (for that matter of all existing algorithms) is limited because updates inflict large blocking delays due to conflicts. On the contrary, the maximum throughput of FDA is limited by the disk throughput. Since throughput of the I/O device can be easily adjusted (by putting more disks or using a disk of higher speed), the performance of FDA can be readily controlled. Whereas, in SDA (and other asymmetric algorithms) the performance is limited by blocking delays due to conflicts which are difficult to control; therefore, tuning the performance of these algorithms is difficult.

FDA has some processing overheads because each site completely executes every update. However, a sacrifice in the power of CPU and disk may be worthwhile when substantial improvement in the performance accrues from it. FDA becomes even more attractive for real-time databases because the presence of large main-memory acts as a catalyst to further improve the performance. Since the performance of FDA is limited by the speed of I/O device, availability of main memory databases will not only reduce its I/O overhead, but will also further enhance its performance.

7. Concluding Remarks

A real-time database system supports applications which have severe performance constraints such as fast response time and continued operation in the face of catastrophic failures. Real-time database systems are still in the state of infancy, and issues and alternatives in their design are not very well explored. In this paper, we have addressed issues in the design of real-time database systems.

We have discussed two approaches to design real-time database systems. In the first approach, we reduce I/O delays by having main memory resident database. Advancements in main memory technology have triggered considerable interests in main memory databases where entire or a major portion of a database resides in the main memory. As a result, access to data objects is extremely fast in these databases as most read and write

actions of updates can be executed without making an access to the disk. The second approach calls for trading a feature of conventional databases to enhance the performance. In this paper, we have discussed the possibilities of trading user friendly interface and the condition of serializability to obtain high performance.

We have also discussed the requirements in the design of real-time distributed database systems and have addressed issues in the design of concurrency control and crash recovery. Long communication delays are found to be a factor in limiting the performance of real-time distributed database systems. We have presented a concurrency control algorithm for real-time distributed database systems whose performance is not limited by the speed of communication network. Instead, the performance depends upon the speeds of CPU and I/O device. Since real-time database systems have extremely small I/O delays (as most of them have main memory resident database), the proposed algorithm becomes even more attractive for such environments.

Acknowledgements

The author is grateful to Professor Son of University of Virginia for making useful comments of the previous version of this paper.

References

1. ATTAR, R., BERNSTEIN, P. A., AND GOODMAN, N., "Site Initialization, Recovery, and Backup in a Distributed Database System," *IEEE Trans. on Software Engineering*, pp. 645-650 (November 1984).
2. BITTON, DINA, "The Effect of Large Main Memory on Database Systems," *Proc. of the ACM SIGMOD*, pp. 337-339 (1986).
3. BLAUSTEIN, B.T., GARCIA-MOLINA, H., RIES, D.R., CHILENSKAS, R.M., AND KAUFMAN, C.W., "Maintaining Replicated Databases Even in Presense of Network Partitions," *EASCON*, pp. 353-360 (1983).
4. CHAN, A., DAYAL, U., FOX, S., GOODMAN, N., SKEEN, D., AND RIES, D., "DDM: An Ada Compatible Distributed Database Manager," *IEEE COMPCON Digests of Papers*, (1983).
5. DAVIDSON, S. B., GARCIA-MOLINA, HECTOR, AND SKEEN, DALE, "Consistency in Partitioned Networks," *ACM Computing Surveys*, pp. 341-370 (September 1985).
6. DEWITT, D., KATZ, R., OLKEN, F., SHAPIRO, L., STONEBRAKER, M., AND WOOD, D., "Implementation Techniques for Main Memory Database Systems," *ACM SIGMOD Conference Proceedings*, (1984).
7. FISCHER, M. J. AND MICHAEL, A., "Sacrificing Serializability to Attain High Availability in an Unreliable Network," *Proc. of the First ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (1982).
8. GARCIA-MOLINA, H., "Performance Comparison of Two Update Algorithms For Distributed Databases," *Proc. of 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 108-119 (Aug. 1978).
9. GARCIA-MOLINA, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Trans. on Database Systems*, pp. 186-213 (June 1983).

10. HAGMANN, R., "A Crash Recovery Scheme for a Memory Resident Database System," *IEEE Transactions of Computers*, pp. 839-843 (September 1986).

11. HAMMER, M. AND SHIPMAN, D., "Reliability Mechanism for SDD-1: A System for Distributed Databases," *ACM Trans. on database Systems*, pp. 431-466 (December 1980).

12. LAMPORT, L., "Time, Clocks and Ordering of Events in Distributed Systems," *Communications of ACM*, pp. 558-565 (July 1978).

13. LEHMAN, TOBIN AND CAREY, MICHAEL, "Query Processing in Main Memory Database Management Systems," *in the Proc. of ACM SIGMOD*, pp. 239-250 (1986).

14. MILENKOVIC, M., "Synchronization of Concurrent Updates in Redundant Distributed Databases," *Distributed Data Bases*, pp. 49-65 North-Holland Publishing Co., (1980).

15. PAPADIMITRIOU, C. H., "Serializability of Concurrent Updates," *Journal of ACM*, pp. 631-653 (Oct. 1979).

16. SALEM, K. AND GARCIA-MOLINA, H., "Crash Recovery Mechanisms for Main Storage Database Systems," *Tech. Rep. CS-TR-034-86*, Dept. of Computer Science, Princeton University, Princeton, (April 1986).

17. SHAPIRO, L., "Join Processing in Database Systems with Large Memories," *ACM Trans. on Database Systems*, pp. 239-264 (September 1986).

18. SINGHAL, MUKESH, "Concurrency Control Algorithms and Their Performance in Replicated Database Systems," *Ph.D. dissertation*, Dept. of Computer Science, University of Maryland, College Park, (February, 1986).

19. SINGHAL, MUKESH AND AGRAWALA, A. K., "A Concurrency Control Algorithm and its Performance for Replicated Database Systems," *Proc. of the 6th International Conference on Distributed Computing Systems, Cambridge, Massachusetts*, (May 19-23, 1986).

20. SON, S. H., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *in the Proc. of the Symposium on Real-Time Systems*, pp. 79-86 (December 1987).

21. SON, S. H., "A Recovery Scheme for Database Systems with Large Main Memory," *in the Proc. of the 11th Annual International Computer Software and Applications Conference, Tokyo, Japan*, pp. 422-427 (October 7-9, 1987).