Chapter 6

REAL-TIME INDEX CONCURRENCY CONTROL

Jayant R. Haritsa
Database Systems Lab, Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, INDIA.
haritsa@dsl.serc.iisc.ernet.in

S. Seshadri
Lucent Bell Labs
600 Mountain Avenue, Murray Hill, NJ 07974, USA.
seshadri@research.bell-labs.com

1. INTRODUCTION

Database systems implement **indexes** to efficiently locate data items on disk and thereby significantly improve query response times. While indexes are an integral component of traditional DBMS, they assume *even greater significance* in the real-time environment due to the stringent time constraints placed on transaction completion. Therefore, it can be reasonably expected that well-designed RTDBS will make extensive use of indexes to enhance their ability to meet transaction deadlines.

A practical example of the above scenario is the following: The rapid spread of cellular communication technology has resulted in database systems having to support environments with a large of number of highly mobile users [12]. Due to the users mobility, the spatio-temporal characteristics of the answers, and the need to conserve battery power, these mobile interactions with the database system have to be processed in real-time and indexes will be essential for quickly processing frequent operations such as location updates ("I am here") and location queries ("Where is X").

For RTDBS supporting high transaction processing rates, the contention among transactions concurrently using the index may *itself* form a performance bottleneck. In particular, note that while contention for the physical resources

can always be reduced by purchasing more and/or faster hardware, there exists no equally simple mechanism to reduce index contention since the base data, and hence indexes, cannot be "manufactured". Similarly, even in environments where the contention on base data is largely absent due to transactions accessing mostly disjoint sets of data, index contention may *continue* to pose performance problems since a common access structure is used to access this base data. Therefore, in this sense, *index contention is a more "fundamental" determinant of RTDBS performance as compared to other shared system resources*.

From the above discussion, it is clear that a pre-requisite for RTDBS to realize the benefits of indexes is incorporation of **index concurrency control(ICC)** protocols that are *tuned* to the objectives of the real-time environment. However, while a rich body of ICC protocols is available for traditional DBMS, and RTDBS design has been investigated for over a decade, research into real-time ICC protocols is not only very sparse but also only of recent vintage. This lacuna is rather surprising since, as mentioned earlier, it appears highly reasonable to expect that RTDBS will make extensive use of indexes to quickly access the base data and thereby help more transactions meet their deadlines. In the remainder of this chapter, we describe our efforts, which represent the first work in the area, to address this lacuna.

Our study is carried out in the context of real-time applications with "firm-deadlines" [8] – for such applications, completing a transaction after its has expired is of no utility and may even be harmful. Therefore, transactions that miss their deadlines are considered to be worthless and are immediately "killed" – that is, aborted and permanently discarded from the RTDBS without being executed to completion. Accordingly, the performance metric is *KillPercent*, the steady-state percentage of killed transactions.[1]

2. B-TREE INDEXES AND PROTOCOLS

While a large variety of index structures have been proposed in the literature, commercial database systems typically use **B-tree** indexing [3] as the preferred access method. In particular, they implement the B^+ variant in which all data key values are stored at the leaf nodes of the index; hereafter, our usage of the term B-tree refers to this variant.

The transactional operations associated with B-trees are *search*, *insert*, *delete* and *append* of key values. Search, insert and delete are the traditional index operations. The append operation is a special case of the insert operation wherein the inserted key is larger than the maximum key value currently in the index. In this case, the new key will always be inserted in the rightmost leaf node of the B-tree. The distinction between appends and generic inserts

[1] Or, equivalently, the percentage of missed deadlines.

is important since when appends are frequent[2] the rightmost leaf node of the index effectively becomes a "hot-spot". Finally, note that search operations arise out of transaction reads while the insert, delete and append operations arise out of transaction updates.

The basic maintenance operations on a B-tree are *split* and *merge* of index nodes. In practical systems, splits are initiated when a node overflows while merges are initiated when a node becomes empty. An index node is considered to be **safe** for an insert if it is not full and safe for a delete if it has more than one entry (index nodes are *always* safe for searches since they do not modify the index structure). A split or a merge of a leaf node propagates up the tree to the closest (with respect to the leaf) safe node in the path from the root to this leaf. If all nodes from the root to the leaf are unsafe, the tree increases or decreases in height. The set of nodes that are modified in an insert or delete operation is called the **scope** of the update.

B-tree ICC protocols maintain index consistency in the face of concurrent transaction accesses and updates. This is achieved through the use of locks on index nodes. Index node locks are typically implemented in commercial DBMS using *latches*, which are "fast locks" [17]. An important aspect of latches is that deadlocks involving latches are not detected and ICC protocols have to therefore ensure that latch deadlocks can never occur.

The index lock modes discussed in this chapter and their compatibility relationships are given in Table 6.1. In this table, IS, IX, SIX and X are the standard "intention share","intention exclusive", "share and intention exclusive" and "exclusive" locks, respectively [5].

mode	IS	IX	SIX	X
IS	Y	Y	Y	N
IX	Y	Y	N	N
SIX	Y	N	N	N
X	N	N	N	N

Table 6.1 Index Node Lock Compatibility Table

Some B-tree ICC protocols use a technique called **lock-coupling** in their descent from the root to the leaf. An operation is said to lock-couple when it requests a lock on an index node while already holding a lock on the node's parent. If the new node is found to be safe, all locks held on any ancestor are released. A variant of this technique is *optimistic lock-coupling*, wherein,

[2]For example, when tuples are inserted into a relation in index key order.

regardless of safety, the lock at each level of the tree is released as soon as the appropriate child has been locked.

There are three well-known classes of B-tree ICC protocols: Bayer-Schkolnick, Top-Down and B-link. Each class has several flavors and we discuss only a representative set here.

2.1 BAYER-SCHKOLNICK PROTOCOLS

We consider three protocols in the Bayer-Schkolnick class [2] called B-X, B-SIX and B-OPT, respectively. In all these protocols, readers descend from the root to the leaf using optimistic lock-coupling with IS locks. Their update protocols, however, differ: In **B-X**, updaters lock-couple from the root to the leaf using X locks. In **B-SIX**, updaters lock-couple using SIX locks in their descent to the leaf. On reaching the leaf, the SIX locks in the updaters scope are converted to X locks. In **B-OPT**, updaters make an optimistic lock-coupling descent to the leaf using IX locks. After the descent, updaters obtain a X lock at the leaf level and complete the update if the leaf is safe. Otherwise, the update operation is restarted, this time using SIX locks.

2.2 TOP-DOWN PROTOCOLS

In the Top-Down class of protocols (e.g. [20, 15]), readers use the same locking strategy as that of the Bayer-Schkolnick protocols. Updaters, however, perform *preparatory* splits and merges during their index descent: If an inserter encounters a full node it performs a preparatory node split while a deleter merges nodes that have only a single entry. This means that unlike updaters in the Bayer-Schkolnick protocols who essentially update the entire scope at one time, the scope update in Top-Down protocols is split into several smaller, atomic operations. In particular, a lock on a node can be released once its child is either found to be safe or made safe by the preparatory split or merge mentioned above.

We consider three protocols in the Top-Down class called TD-X, TD-SIX and TD-OPT, respectively: In **TD-X**, updaters lock-couple from the root to the leaf using X locks. In **TD-SIX**, updaters lock-couple using SIX locks. These locks are converted to X-locks if a split or merge is made. In **TD-OPT**, updaters optimistically lock-couple using IX locks in their descent to the leaf and then get an X lock on the leaf. If the leaf is unsafe, the update operation is restarted from the index root, using SIX locks for the descent.

2.3 B-LINK PROTOCOLS

A B-link tree [16, 22, 15] is a modification of the B-tree that uses links to chain together all nodes at each level of the B-tree. Specifically, each node in a B-link tree additionally contains a high key (the highest key of the subtree

rooted at this node) and a link to the right sibling, which are used during a node split or a node merge. A node is split in two phases: a half-split, followed by the insertion of an index entry into the appropriate parent. Operations arriving at a newly split node with a search key greater than the high key use the right link to get to the appropriate node. Such a sideways traversal is called a *link-chase*. Merges are also done in two steps [15], via a half-merge followed by the appropriate entry deletion at the next higher level.

In B-link ICC protocols, readers and updaters *do not* lock-couple during their tree descent. Instead, readers descend the tree using IS locks, releasing each lock *before* getting a lock on the next node. Updaters also behave like readers until they reach the appropriate leaf node. On reaching the leaf, updaters release their IS lock and then try to get an X lock on the same leaf. After the X lock is granted, they may either find that the leaf is the correct one to update or they have to perform link-chases to get to the correct leaf. Updaters use X locks while performing all further link chases, releasing the X lock on a node before asking for the next. If a node split or merge is necessary, updaters perform a half-split or half-merge. They then release the X lock on the leaf and propagate the updates, using X locks, to the higher levels of the tree, holding at most one X lock at a time.

As discussed above, B-link protocols limit each sub-operation to nodes at a single level. This is in contrast to Top-Down protocols which break down scope updating into sub-operations that involve nodes at *two* adjacent levels of the tree. B-link protocols also differ from Top-Down protocols in that they do their updates in a bottom-up manner. We consider only one B-link protocol here, which exactly implements the above description. This protocol is referred to as the **LY** protocol in [23], and was found to have the best performance of all the above-mentioned protocols with respect to the conventional DBMS metric of transaction throughput.

3. REAL-TIME INDEX CONCURRENCY CONTROL

In the previous section, we described the functioning of the various classical ICC protocols. We now move on to considering the major issues that arise when incorporating these protocols in an RTDBS environment. In particular, we discuss how real-time priorities are incorporated, how transaction serializability is ensured and, finally, how the undos of the index actions of aborted transactions are implemented.

3.1 PRIORITY INCORPORATION

Since satisfaction of transaction timing constraints is the primary goal in RTDBS, transactions are usually assigned priorities to reflect to the system resources each transaction's urgency relative to other concurrently executing

transactions – the Earliest Deadline policy [14], for example, assigns priorities in the order of transaction deadlines.

The ICC protocols described in the previous section do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature [24]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable. We therefore need to design priority-cognizant schemes for ICC protocols in order to adapt them to the real-time environment.

Latches (i.e., locks on index nodes) are typically held only for a short duration, in contrast to data locks which are usually retained until EOT. Therefore, it may be questioned as to whether adding priority-cognizance to latches is really necessary. The reason this can make a difference is the following: The time for which computation is performed on a node is usually much shorter than the time it takes to retrieve a node from disk. Given this observation, if two transactions conflict on a parent node but require access to different children, then, in the absence of priority-cognizance, a higher priority transaction may have to wait for a lower priority transaction while this lower priority transaction retrieves its desired child from the disk. This delay can prove to be significant, especially in resource-constrained environments, as confirmed in our experiments.

3.1.1 Priority Preemption and Priority Inheritance. Priority can be incorporated into the ICC protocols in the following manner: When a transaction requests a lock on an index node that is held by higher priority transactions in a conflicting lock mode, the requesting transaction waits for the node to be released (the wait queue for an index node is maintained in priority order). On the other hand, if the index node is currently held by only lower priority transactions in a conflicting lock mode, the lower priority transactions are *pre-empted* and the requesting transaction is awarded the lock. The lower priority transactions then re-perform their *current* index operation (not the entire transaction).

As an alternative to the above "priority pre-emption"-based approach, we could also consider techniques based on "priority inheritance" [24]. Here, low priority transactions that block a high priority transaction *inherit* the priority of the high priority transaction. The expectation is that the blocking time of the high priority transaction will be reduced since the low priority transactions will now execute faster and therefore release their index nodes earlier.

The positive feature of the inheritance-based approach is that, unlike pre-emption, it does not run the risk of operation restart and therefore does not waste resources. However, a serious limitation is that due to the nature of index traversal, it is possible that a high-priority transaction may be blocked at *multiple* nodes during its index processing, the *cumulative* effect of which

may result in a missed deadline. We, therefore, utilize the priority preemption approach in our study.

3.1.2 Physical Updates. Independent of whether priority-pre-emption or priority inheritance is used to address the priority inversion discussed above, we recommend a different procedure for the special case where a low priority transaction is in the midst of *physically* making updates to a node that is currently locked by it. In this situation, it appears best to allow the transaction to just complete these updates and release the lock on the updated node in the normal manner, without taking any special action. The reasons are the following:

- A transaction starts making a physical update only after all the required nodes are in memory. Therefore, no disk accesses are required during the updation process, resulting in swift completion. This means that the priority inversion period is very small, having only a negligible performance effect.
- If priority preemption is employed, the physical updates that have already been made have to be *undone*, which may require *commensurate or more effort* as compared to just allowing the update to complete.
- On the other hand, if priority inheritance is used, the protocol code will become significantly more complex and therefore slower to execute, negating the real-time objectives.

3.2 TRANSACTION SERIALIZABILITY

The KillPercent performance metric applies to *entire* transactions, not to individual index actions. We therefore need to consider transactions which consist of *multiple* index actions and ensure that transaction serializability is maintained.

In conventional DBMS, a popular mechanism for providing transaction data concurrency control (DCC) is the well-known ARIES *Next-Key-Locking* protocol [18]. We now describe as to how this protocol could be adapted to the real-time domain. In the following discussion, we use the term "key" to refer to key values in the index and the term "next key" (with respect to a key value k) to denote the smallest key value in the index that is $\geq k$.

Our variant of the Next-Key-Locking protocol works as follows: A transaction that needs to perform an index operation with respect to a specific key (or key range) first descends the tree to the corresponding leaf. It then obtains from the database concurrency control manager, the appropriate lock(s) on the associated key(s). For point searches an S lock is requested on the search key value (or the associated next key, if the search key is not present). A range

search operation has to acquire S locks on *each* key that it returns. In addition, it also acquires an S lock on the next key with respect to the last (largest) key in the accessed range. Inserts acquire an X (exclusive) lock on the next key with respect to the inserted key, acquire an X lock on the key to be inserted, insert the key, and then release the lock on the next key. Deletes, on the other hand, acquire an X lock on the next key with respect to the deleted key, acquire an X lock on the key being deleted, delete the key, and then release the lock on the deleted key.

In traditional Next-Key-Locking, all the above-mentioned locks (unless otherwise indicated) are acquired as needed and released only at the end of the transaction (i.e., strict 2PL). In the RTDBS domain, however, a real-time version of 2PL should be used – in our study, we use 2PL High Priority (2PL-HP) [1] which incorporates a priority pre-emption mechanism similar to that described above for the ICC protocols. An important difference, however, is that transactions restarted due to key-value-lock preemptions have to commence the *entire* transaction once again, not just the current index operation.

Earlier studies of DCC protocols in firm RTDBS (e.g. [8]) have shown optimistic algorithms to usually perform better than locking protocols, especially in environments with significant data contention. In light of this, our choice of 2PL-HP may seem surprising – the rationale is that open problems remain with respect to integrating optimistic DCC schemes with index management [7, 19], making them currently infeasible for incorporation in complete database systems.

3.3 UNDO TRANSACTIONS

Real-time transactions may be aborted due to priority resolution of data conflicts (the 2PL-HP protocol mentioned above), or due to missing their deadlines and therefore being killed. This results in transaction aborts being *much more common* in RTDBS than in conventional DBMS. (In a conventional DBMS, aborts are usually resorted to only to resolve data deadlocks, and deadlocks occur very infrequently in practice [26].)

For aborted transactions, it is necessary to *undo* any effects they may have had on the index structure; since aborts are common, as mentioned above, the processing of these undos may have a significant performance impact. Therefore, the undos of the index actions of aborted transactions must be considered in the design of real-time ICC protocols.

We hereafter use the term *undo transaction* to refer to transactions that require undoing of their index actions. The undo transaction removes all the changes that the original transaction had made on the index structure and also releases all its data locks. Note that is imperative to complete the undo transactions quickly since regular index operations cannot proceed with an

inconsistent index. To ensure this, undo transactions are treated as "golden" transactions, that is, they are assigned higher priority than all other transactions executing in the system. Among the undo transactions, the relative priority ordering is the same as that which was existing between them during their earlier (pre-abort) processing.

4. PERFORMANCE RESULTS

Using a detailed simulation model of a firm-deadline RTDBS, we evaluated the performance of the various real-time ICC protocols over a wide range of index contention environments. The full description of our experiments and results are available in [11] – here, we restrict our attention to three representative workloads: **Low Index Contention (LIC)** – 80% searches, 10% inserts and 10% deletes; **Moderate Index Contention (MIC)** – 100% inserts; and **High Index Contention (HIC)** – 25% searches and 75% appends.

Apart from the primary performance metric of *KillPercent*, two additional performance metrics related to fairness were also evaluated: *SizeFairness* and *TypeFairness*. The *SizeFairness* factor captures the extent to which bias is exhibited towards transactions based on their sizes (number of data items accessed). The *TypeFairness* factor captures the extent to which bias is exhibited towards transactions based on their type. Two types of transactions were considered: *read-only* and *update* – read-only transactions are composed exclusively of search index actions while update transactions include at least one update (insert, delete or append) index action.

Our experimental results showed that index conflict has a considerable impact on the real-time performance, *highlighting the need for designing sophisticated real-time ICC protocols*. Further, two factors characteristic of the (firm) real-time domain: Addition of priority and discarding of late transactions, significantly affected the performance of the ICC protocols, as compared to their behavior in conventional DBMS [23]. This outcome justifies the need to re-evaluate ICC protocol behavior in the real-time environment.

More specifically, the following observations were made:

- Adding preemption to index latches resulted in tangible performance benefits, in accordance with our expectations (Section 3.).
- No appreciable difference was observed between the performance of the corresponding protocols from the Bayer-Schkolnick and Top-Down classes (i.e., between B-X and TD-X, B-SIX and TD-SIX, B-OPT and TD-OPT). This is because the number of exclusive locks held at one time on the scope of an update is hardly different for the two classes for trees with few levels and large fan-outs, which is typically how B-trees are organized. We will therefore simply use X, SIX and OPT to denote these protocols in the sequel.

- All the protocols were noticeably unfair with regard to one or both fairness metrics. For example, B-link and OPT progressively favor smaller-sized transactions with increasing loading levels, while SIX shows extreme bias favoring read-only transactions.

- The X protocol performed very poorly with respect to the other protocols, due to the root of the B-tree becoming a severe bottleneck.

- Surprisingly, B-link missed many more deadlines under heavy loads as compared to lock-coupling protocols. This is in contrast to conventional DBMS where B-link always exhibited the best throughput performance [23]. The reason for the altered performance in the real-time environment is the following: Algorithms such as SIX apply a form of *load control* by giving preferential treatment to read-only transactions. B-link, on the other hand, tends to *saturate* the disk due to largely treating transactions uniformly independent of type and therefore misses significantly more deadlines. That is, the very reason for B-link's good performance in conventional DBMS – full resource utilization – turns out to be a liability here.

  An additional factor is that in conventional DBMS, the slower updaters of SIX clogged the system, resulting in much higher contention levels and poor performance – in the firm real-time environment, however, that does not happen because transactions are *discarded* as soon as their deadlines expire. Therefore, this type of clogging is inherently prevented.

- The OPT protocol performed almost as well as B-link even under HIC conditions. This is also in contrast to conventional DBMS, where OPT performed poorly in this scenario – a large number of restarts were caused by high contention, resulting in the root becoming a bottleneck due to the large number of second-pass updaters. In the real-time environment, however, prioritization of transactions causes a marked decrease in the number of index operation restarts, as explained below.

  In the no priority case, several first pass transactions see the same unsafe leaf node before the *first* transaction which saw it as unsafe has completed its second pass and made the leaf safe (by splitting or merging). This is the source of the large number of restarts. In the prioritized environment, however, the highest priority transaction overtakes other transactions at the nodes it traverses during its descent and therefore completes its second pass very quickly. This results in only relatively few other transactions seeing the leaf node while it is unsafe. In essence, the *time period* for which a node is unsafe is much smaller in the prioritized environment as compared to the non-real-time environment.

4.1 EFFECT OF UNDOS

We now move on to considering the performance impact of modeling the *undos* of the index actions of aborted transactions. It is tempting to surmise that there is really no need to explicitly evaluate this feature based on the following argument, which was advanced in [4]: The number of undos can be expected to be directly related to the transaction kill percentage, that is, a higher kill percentage leads to more undos. Given this, implementing undos in the model would only result in *further increasing* the difference in performance of the protocols seen in the above experiments. This is because the poorly performing algorithms would have *more* clean-up work to do than the better protocols and therefore miss even more transaction deadlines. In summary, there is a direct positive feedback between the undo overhead and the kill percentage. Due to this relationship, incorporating undo actions will correspondingly increase, but not qualitatively alter, the observed performance differences between the various protocols.

While the above argument may appear plausible at first sight, our study showed that the impact of undos is not so simply characterized and that unexpected effects can show up in certain environments.

Firstly, under LIC and MIC environments, undos had only a *marginal* adverse impact on performance. Further, this marginal impact was *not limited to light loads* – it also occured for high kill percentage values where we would expect the undo overheads to be considerable due to the large number of aborted transactions.

Even more interestingly, the protocols often *performed better with undos than in the absence of undos*, especially in the lower range of the loading spectrum! While the improvement is typically small, for SIX a *significant* improvement was observed. These results, which suggest that the presence of overheads helps to *improve* performance may appear at first glance to be illogical. A careful investigation showed, however, that there are subtle effects that come into play in the undo model, which do indeed cause this apparently strange behavior. We explain these reasons below.

1. The primary reason for little adverse performance impact in the LIC and MIC environments is that the *disk* is the bottleneck in these experiments. Undo operations, however, typically only need the CPU since the index pages they require will usually be *already resident in the buffer pool* as they have been accessed recently. In essence, undo operations primarily impose a *CPU overhead*, not a disk overhead. This means that only when the CPU itself is a bottleneck, as in the HIC environment, are there the large differences that we might intuitively expect to see.

2. The number of *index node splits and merges reduces significantly in the presence of undos*. This is because they *compensate* for the actions of

aborted transactions. This is especially significant in light of the fact that index node splits and merges are expensive operations since they necessitate additional disk I/O.

3. The reason for SIX performing so much better with undos at low arrival rates in the LIC environment is that the average latch wait time for the normal operations is greatly reduced by the presence of undo operations. This is explained as follows: In SIX, readers usually descend the tree very quickly since they use IS locks which are compatible with the SIX locks of updaters. Updaters, on the other hand, have to wait for each other since SIX locks are mutually incompatible. Due to the lock-coupling nature of SIX, this may result in a *convoy* phenomenon, wherein there is a chain of updaters from the leaf to the root, each holding a latch that the next one wants. Since index trees are usually of small height, such convoys can occur very easily. Moreover, the convoy may persist for quite a while since the updater on the leaf node typically has to wait for the leaf node to be brought from the disk before it can release its latch on the parent node (and that too, only if the leaf is safe).

   The presence of undo transactions *eliminates* the convoy bottlenecks described above. This is because, by virtue of their "golden" priority, they preempt during undo processing all transactions holding an incompatible latch in their tree descent. In particular, they preempt updaters holding leaf level latches and waiting for their disk requests to be completed. This eliminates the convoy bottleneck and allows other updaters to gain possession of the higher level latches and quickly descend to the leaf nodes, especially since latching times are very small as compared to disk access times.

   The benefit of convoy elimination is felt only under light and moderate loads but not under heavy loads. This is because, under heavy loads, the disk is almost fully utilized and becomes the primary bottleneck. In this situation, removing the latching bottleneck proves ineffective and is actually harmful since it increases the amount of wasted work due to the large number of preemptions.

4.2 RANGE QUERIES

For range queries, we observed that the results are qualitatively similar to those seen with point queries. Quantitatively, however, the performance differences between the protocols generally *increased*. The reason for this increase is that range searches significantly increase the level of *data contention* in the system since they hold locks on several key values simultaneously.

5. THE GUARD-LINK PROTOCOL

One of the outcomes of the results presented above is that it appeared that the performance of B-link could be improved by adding a *load-control* component. Accordingly, we designed a new protocol called **GUARD-link**, which augments the real-time variant of the B-link protocol with a feedback-based *admission control* mechanism called GUARD (Gatekeeping Using Adaptive eaRliest Deadline). The details of the algorithm are in [11].

On evaluation, we found that GUARD-link, by virtue of its admission control policy which successfully limits transaction admissions to a sustainable number, significantly reduced the kill percentage of B-link in the overload region and thereby provided the best performance over the entire loading range. This clearly demonstrates the need for admission control in real-time index management. Interestingly, such need for load control has also been identified in other RTDBS modules (e.g. [9, 21]).

Apart from its good MissPercent performance, GUARD-link also provided close to *ideal fairness*, not discriminating either based on transaction size or on transaction type. Therefore, we suggest that RTDBS designers may find the GUARD-link protocol to be a good choice for real-time index concurrency control.

6. RELATED WORK

The material discussed in this chapter is based on our earlier papers [4, 10, 11]. Apart from this, the only other work that we are aware of on real-time ICC appeared very recently in [13]. While our focus was on improving the real-time performance of classical ICC protocols by suitable use of priority and admission-control mechanisms for both point and range queries, the emphasis in [13] is on "relaxing" traditional index structures in a manner that improves timeliness for *batch* queries, where several keys are provided at a time. In particular, a B-tree that is re-balanced only periodically, but not continuously, is considered. Specialized techniques to increase the buffer hit ratio across transactions and to directly access the index from internal nodes are evaluated and shown to provide improved real-time performance.

7. SUMMARY AND FUTURE WORK

In this chapter, we have attempted to characterize the state-of-the-art with regard to real-time index processing. This topic is of importance since it appears highly reasonable to expect that RTDBS will make extensive use of indexes to quickly access the base data and thereby help more transactions meet their deadlines. We studied the real-time performance of the Bayer-Schkolnick, Top-Down and B-link classes of ICC protocols under a range

of workloads and operating conditions. We also proposed and evaluated a new ICC protocol called GUARD-link, which augments the classical B-link protocol with an admission control mechanism. The GUARD-link protocol provided the best real-time performance in terms of both the number of missed deadlines and the fairness with respect to the distribution of these missed deadlines across transaction classes. Finally, we identified some unexpected performance outcomes of executing the undos of index actions of aborted transactions.

We now outline a variety of issues that appear appropriate for further research: First, we have not considered in our study a high-performance ICC protocol called ARIES/IM that was proposed in [17]. In [23], several reasons were advanced as to why ARIES/IM might be expected to exhibit performance behavior similar to that of the B-link protocol in the conventional database domain. While we expect a similar outcome to also arise in the real-time environment, confirmation of this expectation requires actual implementation and evaluation of the ARIES/IM protocol.

Second, as mentioned in Section 3., an alternative to the priority-preemption approach that we considered here is priority inheritance. It would be interesting to evaluate the performance behavior of this scheme. Similarly, instead of treating undo transactions as "golden" transactions, a priority inheritance approach could be used there too to ensure that these transactions interfere only minimally with ongoing processing.

Third, we assumed here that all transactions have the same "criticality" or "value" [25], and therefore the performance goal was to minimize the *number* of killed transactions. An important open area is the characterization of ICC protocol behavior for workloads where transactions have varying values, and the goal of the RTDBS is to maximize the *total value* of the completed transactions. These results could also be extended to "soft- deadline" applications, where there is a diminishing residual value for completing transactions even after their deadlines have expired.

Finally, another interesting issue is to extend our analysis to *multi-dimensional* index structures such as, for example, *R-trees* [6]. We expect that these structures would be utilized in real-time applications such as mobile databases that include spatial data in their repositories.

Acknowledgements S. Seshadri worked on this project when he was with the Dept. of Computer Science and Engineering, Indian Institute of Technology, Mumbai, India. Brajesh Goyal, S. R. Narayanan and V. Srinivasan made significant contributions at various stages to this project. Jayant Haritsa's work was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *Proc. of 14th Intl. Conf. on Very Large Data Bases*, August 1988.

[2] R. Bayer and M. Schkolnick, "Concurrency of Operations on B-trees", *Acta Informatica*, 9, 1977.

[3] D. Comer, "The Ubiquitous B-tree" *ACM Computing Surveys*, 11(4), 1979.

[4] B. Goyal, J. Haritsa, S. Seshadri, and V. Srinivasan, "Index Concurrency Control in Firm Real-Time DBMS", *Proc. of 21st Intl. Conf. on Very Large Data Bases*, September 1995.

[5] J. Gray, "Notes on Database Operating Systems", *Operating Systems: An Advanced Course*, R. Graham, R. Bayer and G. Seegmuller, eds., Springer-Verlag, 1979.

[6] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1984.

[7] T. Haerder, "Observations on Optimistic Concurrency Control Schemes", *Information Systems*, 9(2), 1984.

[8] J. Haritsa, M. Carey and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems", *Journal of Real-Time Systems*, September 1992.

[9] J. Haritsa, M. Livny and M. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems", *Proc. of 1991 IEEE Real-Time Systems Symp.*, December 1991.

[10] J. Haritsa and S. Seshadri, "Real-Time Index Concurrency Control", *SIGMOD Record*, 25(1), March 1996.

[11] J. Haritsa and S. Seshadri, "Real-time Index Concurrency Control", *IEEE Trans. on Knowledge and Data Engineering*, 12(3), May/June 2000.

[12] T. Imielinski and B. Badrinath, "Querying in Highly Mobile Distributed Environments", *Proc. of 18th Intl. Conf. on Very Large Data Bases*, September 1992.

[13] T. Kuo, C. Wei and K. Lam, "Real-time Access Control and Reservation on B-tree Indexed Data", *Proc. of 15th Intl. Conf. on Data Engineering*, March 1999.

[14] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, January 1973.

[15] V. Lanin and D. Shasha, "A Symmetric Concurrent B-tree Algorithm", *Proc. of Fall Joint Computer Conf.*, 1986.

[16] P. Lehman and S. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Trans. on Database Systems*, 6(4), 1981.

[17] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-Ahead Logging", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1992.

[18] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes", *Proc. of 16th Intl. Conf. on Very Large Data Bases*, September 1990.

[19] C. Mohan, "Less Optimism about Optimistic Concurrency Control", *Proc. of 2nd Intl. Workshop on RIDE: Transaction and Query Processing*, February 1992.

[20] Y. Mond and Y. Raz, "Concurrency Control in B^+-trees Databases using Preparatory Operations", *Proc. of 11th Intl. Conf. on Very Large Data Bases*, September 1985.

[21] H. Pang, M. Carey and M. Livny, "Managing Memory for Real-Time Queries", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1994.

[22] Y. Sagiv, "Concurrent Operations on B*-trees with Overtaking", *Journal of Computer and System Sciences*, 33(2), 1986.

[23] V. Srinivasan and M. Carey, "Performance of B-tree Concurrency Control Algorithms", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1991.

[24] L. Sha, R. Rajkumar and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *Tech. Report CMU-CS-87-181*, Depts. of CS, ECE and Statistics, Carnegie Mellon University, 1987.

[25] J. Stankovic and W. Zhao, "On Real-Time Transactions", *ACM SIGMOD Record*, March 1988.

[26] A. Thomasian and I. Ryu, "Performance Analysis of Two-Phase Locking", *IEEE Trans. on Software Engg.*, March 1991.