Index Concurrency Control in Firm Real–Time DBMS

Brajesh Goyal * Jayant R. Haritsa † S. Seshadri * V. Srinivasan ††

* Dept. of Computer Science and Engineering
Indian Institute of Technology, Bombay 400076, India
{brajesh, seshadri}@cse.iitb.ernet.in

† Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, India
haritsa@csa.iisc.ernet.in

†† IBM Santa Teresa Laboratory
555 Bailey Avenue, San Jose, CA 95050, U.S.A
srini@vnet.ibm.com

Abstract

Although real-time transaction concurrency control has been extensively studied, the design and evaluation of real-time *index* concurrency control algorithms has not yet been considered. In this paper, we develop real-time variants of several classical B-tree concurrency control algorithms and compare their performance using a detailed simulation model of a firm-deadline real-time database system. The experimental results show that the performance characteristics of the real-time version of an index concurrency control algorithm could be significantly different from the performance of the same algorithm in a conventional (non-real-time) database system. In particular, B-link algorithms, which are reputed to provide the best overall performance in conventional database systems, perform poorly under heavy real-time loads. We present and evaluate a simple load-adaptive variant of the B-link algorithm called LAB-link, which provides the best performance over the entire loading range for a variety of real-time transaction workloads.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

1 Introduction

A real-time database system (RTDBS) is a transaction processing system that is designed to handle workloads where transactions have completion deadlines. The objective of the system is to meet these deadlines, that is, to process transactions before their deadlines expire. Research on real-time database systems has focused mainly on identifying the appropriate choice, with respect to meeting the real-time goals, for the various database system policies such as priority assignment, concurrency control, memory management, etc. (e.g. [AG92, HCL92, AEJ92, PCL94, HJC93]). However, the design and evaluation of concurrency control algorithms for **indexes**, which are an integral part of database systems, has not yet been considered for the real-time environment (to the best of our knowledge). While index concurrency control is an important issue in a conventional (non-real-time) DBMS, it is especially important in the RTDBS environment since the system will rely heavily on indexes to help transactions meet their time constraints. In this situation, a poor index concurrency control algorithm may lead to many transactions missing their deadlines. We address this issue in this paper.

Database systems implement specialized access methods, such as indexing and hashing, in order to efficiently locate data items on disk. While a large variety of access methods have been proposed in the literature, commercial database systems typically use **B-tree** indexing [Com79] 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. In the remainder of this paper, our usage of the term B-tree refers to this variant.

For database systems supporting high transaction rates, the contention among transactions concurrently using the B-tree may itself form a performance bottleneck. To address this issue, a number of high-concurrency algorithms have been proposed for B-tree access (e.g. [BS77, Bil86, Sag85, KW82, LY81, LS86, SG88, ML92, Moh90, LS92]). The performance of a representative set of these algorithms has recently been profiled in [JS90, SC91] and their results indicate that **B-link** algorithms [LY81] provide the best performance over a wide range of workloads and system operating conditions.

The above-mentioned performance studies were done in the context of a conventional DBMS where transaction throughput or response time is the primary performance metric. In a real-time database system, however, performance is usually measured in terms of the *number* of transactions that complete before their deadlines. That is, a transaction that completes just before its deadline is no different, from a performance perspective, to one that finishes much earlier. Due to the difference in objectives, the performance of index concurrency control algorithms has to be reevaluated for the real-time domain.

Another important difference between our study and those of [JS90, SC91] is that transactions consist of *multiple* index actions in our model. In contrast, the earlier studies modeled "tree" transactions wherein each transaction performs only a single B-tree operation (search or update). The tree model is not appropriate for the real-time environment since the metric of missed deadlines is meaningful only when applied to complete transactions. In our model of an RTDBS, therefore, transactions are capable of performing multiple index operations and mechanisms for ensuring transaction serializability are implemented.

There are two major issues that need to be explored with regard to real-time index concurrency control: First, how do we adapt the concurrency control protocols to the real-time domain? Second, how do these real-time variants compare in their performance? In this paper, we address these questions for the "firm-deadline" [HCL92] application framework, wherein transactions that miss their deadlines are considered to be worthless and are immediately discarded from the system without being executed to completion. We develop real-time variants of several classical B-tree concurrency control algorithms and compare their performance using a detailed simulation model of a real-time database system. The performance metric is the steady-state percentage of transaction deadlines that are missed.

The results of our study show that, in moving from the conventional DBMS domain to the RTDBS domain, there are new performance-related forces that come into effect and that these factors can cause index performance behaviors that were valid in a conventional DBMS setting to be significantly altered in the corresponding RTDBS setting[1]. For example, our experimental results indicate that B-link algorithms provide the best real-time performance for light and moderate loads but perform poorly under heavy loads. Analysis of the results shows that the heavy-load deterioration is due to the B-link algorithms unique ability to fully utilize the physical resources of the DBMS. This feature led to their good performance in conventional DBMS [SC91], but in the real-time domain, resource saturation results in more missed deadlines. To address this problem, we have developed a load-adaptive variant of the B-link algorithm called **LAB-link**, which ensures that the bottleneck resource of the RTDBS does not become saturated. Simulation of the LAB-link algorithm shows that it provides the best performance over the entire loading range for a variety of real-time transaction workloads.

The remainder of this paper is organized as follows: The B-tree concurrency control algorithms evaluated in this study are outlined in Section 2, and the real-time aspects of index concurrency control are discussed in Section 3. The performance model is described in Section 4, and the results of the simulation experiments are highlighted in Section 5. In Section 6, we present the conclusions of our study and identify future research avenues.

2 B-tree CC Algorithms

In this section, we describe the set of B-tree concurrency control algorithms considered in our study. We assume, in the following discussion, that the reader is familiar with the basic features and operations of B-tree index structures [Com79, SC91].

The transaction operations associated with B-trees are *search*, *insert*, *delete* and *append* of key values. Search corresponds to transaction reads while insert, delete and append correspond to transaction updates. The basic maintenance operations on a B-tree are split and merge of index nodes. In practical systems, splits

[1] A similar, though unrelated, phenomenon has been observed for *transaction* concurrency control [HCL92].

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. A split or merge of a leaf node propagates up the tree to the lowest 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 concurrency control (CC) algorithms maintain index consistency in the face of concurrent transaction accesses. This is achieved through the use of locks[2] on index nodes. The index lock modes discussed in this paper and their compatibility relationships are given in Table 1. In this table, IS, IX, SIX and X are the standard "intention share","intention exclusive", "share and intention exclusive" and "exclusive" locks, respectively [Gra79][3].

Table 1: Index Node Lock Compatibility Table

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

Some B-tree CC algorithms 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, releasing the parent lock if the new node is found to be safe.

There are three well-known classes of B-tree CC algorithms: Bayer-Schkolnick, Top-Down and B-link. These classes primarily differ in the granularity of their scope update operations, as explained below. Each class has several flavors and we discuss only a representative set here, similar to [SC91].

2.1 Bayer-Schkolnick Algorithms

We consider three algorithms in the Bayer-Schkolnick class [BS77] called B-X, B-SIX and B-OPT, respectively. In all these algorithms, readers descend from the root to the leaf using lock-coupling with IS locks. They differ, however, in their update protocols: 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 their scope are converted to X locks. In **B-OPT**, updaters make an *optimistic* lock-coupling descent to the leaf using IX locks. The descent is called called optimistic since regardless of safety, the lock at each level of the tree is released as soon as the appropriate child has been locked. 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 Algorithms

In the Top-Down class of algorithms (e.g. [MR85, LS86]), readers use the same locking strategy as that of the Bayer-Schkolnick algorithms. 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 algorithms who essentially update the entire scope at one time, the scope update in Top-Down algorithms is split into several smaller, atomic operations.

We consider three algorithms 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 lock-couple using IX locks[4] 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 Algorithms

A B-link tree [LY81, Sag85, LS86] 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 contains a high key (the highest key of the subtree rooted at this node) and a link to the right sibling. These links are used to split nodes 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 [LS86], via a half-merge followed by the appropriate entry deletion at

[2] In commercial DBMSs, index node locks are usually implemented using *latches*, which are "fast locks". This optimization is taken into account in our performance study, as discussed later in Section 3.

[3] The lock modes and lock compatibility matrix used here are identical to those of [SC91], except for terminology – they used S to denote IS lock mode.

[4] In [SC91], updaters in the TD-OPT algorithm use IS locks in the first pass, not IX locks. This strategy could cause deadlocks between first pass updaters and second pass updaters. We have therefore used IX locks to avoid such deadlocks.

the next higher level.

In the B-link concurrency control algorithms, 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.

The Top-Down algorithms break down scope updating into sub-operations that involve nodes at two adjacent levels of the tree. The B-link algorithms, on the other hand,6-mtlimit each sub-operation to nodes at a single level. They also differ from the Top-Down algorithms in that they do their updates in a bottom-up manner. We consider only one B-link algorithm here, which exactly implements the above description. This algorithm is referred to as the **LY** algorithm in [SC91], and was found to have the best performance of all the above-mentioned algorithms with respect to transaction throughput.

2.4 ARIES/IM Algorithm

An algorithm for high-concurrency index management called ARIES/IM was described in [ML92]. ARIES/IM has both left and right pointers linking nodes at the leaf level, but unlike the B-link algorithm, the nodes at higher levels do not have right links. Updaters in ARIES/IM make an initial descent to the leaf using IS locks, and at the leaf level they may perform link-chases just as in the B-link algorithms. However, while the B-link algorithms perform link-chases at all levels, updaters in ARIES/IM instead use a complex protocol based on recursive restarts. These restarts do not have the disadvantage of creating any bottlenecks, however, as the operations use extra information stored in the B^+-tree nodes to ensure consistency rather than using exclusive locks. The performance study in [SC91] explains why the ARIES/IM algorithm will perform close to the LY algorithm for most workloads. We expect the same to happen here and do not explicitly simulate the ARIES/IM algorithm in our experiments.

3 Real–Time Index CC

Satisfaction of transaction timing constraints is the primary goal in real-time database systems (rather than other considerations, such as fairness). Therefore, the scheduling policies at the various resources (both physical and logical) in the system can be reasonably expected to be priority-driven with the priority assignment scheme being tuned to minimize the number of missed deadlines. The index concurrency control algorithms described above 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 [SRL87]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable. We therefore need to design preemption schemes for index concurrency control algorithms in order to adapt them to the real-time environment.

3.1 Index Node Locks

We have incorporated priority into the index CC algorithms 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 *preempted* and the requesting transaction is awarded the lock. The lower priority transactions then restart, from the beginning, their *current* index operations (not the entire transaction). The only exception to this procedure is when a low priority transaction is in the midst of *physically* making updates to a node which is currently locked by it. In this situation, the low priority transaction is not preempted until it has completed these updates and released the lock on the updated node. Since these operations are typically very fast, we expect that the effect of having these extremely short priority inversion periods is negligible.

Locks on index nodes are typically held for very short durations, and commercial database systems tend to implement such short duration locks as optimized "fast-locks" or *latches* [ML92]. In our simulation model also, index node locks are implemented as latches. Since latches are held only for very short durations, it may be questioned as to whether adding preemption to latches is really necessary. Our experiments (Section 5) show that adding preemption does have an appreciable performance effect.

3.2 Locking for Transaction Serializability

As mentioned in the introduction, the performance metric of missed deadlines applies only to complete transactions. We therefore need to consider transactions which consist of *multiple* index actions and ensure that transaction serializability is maintained. In our study, we use a real-time variant of a simplified form of the *key-value locking* (KVL) mechanism described in [Moh90] to provide transaction concurrency control. In formulating this simplified algorithm, we assume, as in [SC91, JS90], that all indexes are unique and that all index operations are point (single key) operations.

The KVL-based concurrency control algorithm that we use is as follows: A transaction that wishes to make an index access has to first obtain, from the database concurrency control manager, the appropriate lock on the associated key. An S (shared) lock is required for searches, while an X (exclusive) lock is required for updates. After this lock is obtained, the index operation is executed under the supervision of the index concurrency control algorithm. In KVL, the well-known (strict) two-phase locking algorithm (2PL) is used to maintain serializability by regulating access to the key-value locks. For our study, we use a real-time version of 2PL called 2PL-HP [AG92] which incorporates a priority mechanism similar to that described above for the index concurrency control algorithms[5]. Note, however, that transactions restarted due to key-value-lock preemptions have to commence the *complete* transaction once again.

4 Simulation Model and Methodology

In the previous section, we discussed various index concurrency control algorithms and their real time versions. To evaluate the performance of these algorithms, we developed a detailed simulation model of a firm-deadline real-time database system. The organization of our model is based on a loose combination of the database model of [HCL92] and the B-tree system model of [SC91]. A summary of the parameters used in the model are given in Table 2. The following subsections describe the workload generation process, the B-tree model and the hardware resource configuration.

4.1 Transaction Workload Model

Transactions arrive in a Poisson stream and each transaction has an associated deadline. A transaction consists of a sequence of index access operations such as search, insert, delete or append of a key value. After each index operation, the corresponding data access is made. The data access itself is not explicitly modeled but is assumed to take a period of time equal to one disk I/O. A transaction that is restarted due to a data conflict makes the same index accesses as its original incarnation. If a transaction has not completed by its deadline, it is immediately aborted and discarded.

Table 2: Model Parameters

Parameter	Meaning	Value
ArrRate	Transaction arrival rate	$0 - \infty$
TransSize	Average transaction size	8
SlackFactor	Deadline Slack Factor	4
SearchProb	Proportion of searches	0.0 – 1.0
InsertProb	Proportion of inserts	0.0 – 1.0
DeleteProb	Proportion of deletes	0.0 – 1.0
AppendProb	Proportion of appends	0.0 – 1.0
InitKeys	No. of keys in initial tree	100,000
MaxFanout	Key entries per node	300
NumCPUs	Number of processors	$1 - \infty$
SpeedCPU	Processor MIPS	20
LockCPU	Cost for lock/unlock	1000 inst.
LatchCPU	Cost for latch/unlatch	100 inst.
BufCPU	Cost for buffer call	1000 inst.
SearchCPU	Cost for page search	500 inst.
ModifyCPU	Cost for key insert/delete	500 inst.
CopyCPU	Cost for page copy	1000 inst.
NumDisks	Number of disks	$1 - \infty$
PageDisk	Disk page access time	20 ms
NumBufs	Size of buffer pool	$1 - \infty$

The *ArrRate* parameter specifies the mean rate of transaction arrivals. The number of index accesses made by each transaction varies uniformly between half and one-and-a-half times the value of *TransSize*. The overall proportion of searches, inserts, deletes and appends in the workload is given by the *SearchProb*, *InsertProb*, *DeleteProb* and *AppendProb* parameters, respectively. As mentioned earlier, all index search and update operations are point (single key) operations, as in [JS90, SC91].

Transactions are assigned deadlines with the formula $D_T = A_T + SF * R_T$, where D_T, A_T and R_T are the deadline, arrival time and resource time, respectively, of transaction T, while SF is a slack factor. The *resource time* is the total service time at the resources that the transaction requires for its data processing.

The *slack factor* is a constant that provides control over the tightness/slackness of deadlines[6].

[5] Earlier studies have shown optimistic algorithms to perform better than locking protocols in firm RTDBS (e.g. [HCL92]). However, since our experiments here consider only low data contention (but high index contention) situations, the choice of transaction CC mechanism has negligible effect on performance and therefore, for simplicity, a locking protocol has been used.

[6] Although the workload generator uses transaction resource requirements in assigning deadlines, we assume that the RTDBS itself lacks any knowledge of these requirements. This implies that a transaction is detected as being late only when it actually

4.2 B-Tree Model

For simplicity, only a single B-tree is modeled and all transaction index accesses are made to this tree. The initial number of keys in the index is determined by the *InitKeys* parameter. Each index node corresponds to a single disk block and the *MaxFanout* parameter gives the node key capacity, that is, the maximum number of $< key, pointer >$ entries in a node. We assume that all keys are of the same size, and that the indexed attribute is a key of the source relation. If an update transaction is aborted, any index modifications that it may have made have to be undone to maintain index consistency. For simplicity, however, we do not include this in our model – the impact of this choice is discussed at the end of Section 5.

4.3 Resource Model

The physical resources in our model consis6-mt processors, memory and disks. There is a single queue for the CPUs and the service discipline is preemptive-resume, with preemptions being based on transaction priorities. Each of the disks has its own queue and is scheduled with a priority Head-of-Line policy.

Buffer management is implemented using a two-level priority LRU mechanism: Higher priority transactions steal buffers from the lowest priority transaction that currently owns one or more buffers in the memory pool. The least-recently used clean buffer of this transaction is the one chosen for reallocation. If all its buffers are dirty, the least-recently used dirty buffer is flushed to disk and then transferred to the high priority transactions. The lowest priority transaction itself uses a similar LRU mechanism within the set of buffers currently allocated to it[7].

The *NumCPUs*, *NumDisks* and *NumBufs* parameters quantitatively determine the resource configuration. The processing cost parameters for each type of index operation are also given in Table 2.

5 Experiments

In this section, we present the performance results from our simulation experiments comparing the various index concurrency control algorithms in a firm-deadline real-time database system environment (the simulator is written in C++ [Str86]). The transaction priority assignment scheme used in all the experiments reported here is the widely-used *Earliest Deadline*: transactions with earlier deadlines have higher priority than transactions with later deadlines[8].

misses its deadline.

[7] Pinned buffers, are, of course, not eligible to be replaced.

[8] Earliest Deadline has been found to perform poorly in overload situations and modifications have been proposed in the literature to address this problem (e.g. [HCL91]. We intend to experiment with these priority schemes in our future research.

The index key generation process is implemented in the following manner: The keys for the search, insert and delete operations are chosen from a key space that consists of integer values between 1 and 300,000. The inserts can use all key values in the key space that are not exact multiples of 3, whereas the deletes can use the remaining keys (i.e. exact multiples of 3). In contrast to updaters, searches can use all key values in the key space. Finally, the keys for appends are chosen sequentially from 300,001 onwards.

The above key generation scheme is designed to ensure that inserts and deletes do not interfere at the level of key values. Further, to ensure that deletes are always successful, an initial tree is built using a random permutation of all of the keys which are multiples of 3 in the key space, that is, the initial number of keys in the B-tree is 100,000.

5.1 Performance Metric

The performance metric of our experiments is *Miss Percent*, which is the percentage of input transactions that the system is *unable* to complete before their deadlines. A long-term operating region where the miss percentage is high is obviously unrealistic for a viable RTDBS. Exercising the system to high miss levels (as in our experiments), however, provides valuable information on the response of the algorithms to brief periods of stress loading. All the MissPercent graphs of this paper show mean values that have relative half-widths about the mean of less than 10 percent at the 90 percent confidence level, with each experiment having been run until at least 10000 transactions were processed by the system. Only statistically significant differences are discussed here.

The simulator was instrumented to generate a host of other statistical information, including resource utilizations, number of link-chases for B-link algorithms, number of restarts for optimistic algorithms, etc. These secondary measures help to explain the MissPercent behavior of the index concurrency control algorithms under various workloads and system conditions.

5.2 Parameter Settings

As mentioned earlier, the initial tree for each experiment consists of 100,000 keys. The fanout (key capacity) of each node of the B-tree is set to 300. With this fanout, the resultant initial tree is 3 levels deep, consisting of 3 internal nodes and 506 leaf nodes. The tree nodes are assumed to be uniformly distributed across all the disks. The other constant simulation parameters are set to the values shown in Table 2.

We performed several experiments by varying the variable simulation parameters (Table 2). We mainly focus on three representative experiments here which correspond to system conditions of low contention, moderate contention and high contention for the index, respectively. In addition, we also present results on the contribution of index contention to the miss percentage, and the performance in the absence of resource contention. These experiments cover transaction workloads that are similar to those considered in [SC91]. For all the experiments described here, unless explicitly mentioned otherwise, the resource parameter settings are: $NumCPUs = 1$, $NumDisks = 8$, and $NumBufs = 250$.

In all our experiments, no appreciable difference was observed between the performance of the corresponding algorithms from the Bayer-Schkolnick and Top-Down classes (i.e. between B-OPT and TD-OPT, B-X and TD-X, B-SIX and TD-SIX). This is to be expected since the B-tree used in our experiments, as described above, has only 3 levels due to the large fanout. Consequently, the number of exclusive locks held at one time on the scope of an update is hardly different in the two cases. We will therefore simply use OPT, SIX and X to denote these algorithms in the following discussions.

5.3 Expt. 1: Low Index Contention

In our first experiment set, a workload that consisted of 80% searches, 10% inserts and 10% deletes was used. The high percentage of reads as compared to updates results in a low contention environment. Moreover, the balance of insert and delete index operations limits the number of index node splits and merges. For this experiment, Figure 1 shows the miss percent behavior as a function of transaction arrival rate. These results clearly show that the X algorithms perform poorly with respect to the other algorithms. At low arrival rates, the performance of the B-link algorithms is the best, but at high arrival rates the SIX algorithms perform noticeably better. Predictably, the performance of the OPT algorithms is almost identical to that of B-link algorithms, since the number of splits and merges is very low.

The poor behavior of the X algorithms is explained by considering the average amount of time a transaction has to wait for an index lock in each of the algorithms – this statistic is shown as a function of arrival rate in Table 3. The table clearly shows that the average lock wait time for X algorithms is the highest[9]. The reason for this is that the root of the

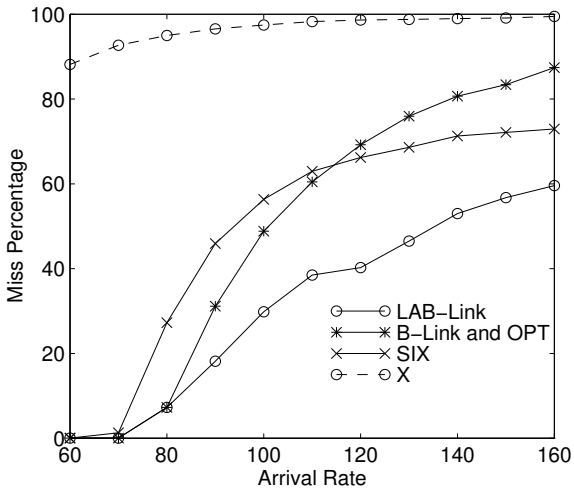

Figure 1: Low Index Contention

B-tree becomes a severe bottleneck, as also observed in [JS90, SC91].

In the corresponding (non-real-time) experiment in [SC91], the B-link algorithms performed much better than the SIX algorithms. However, in our case, though B-link performs the best for low arrival rates, the situation is reversed at high arrival rates where the SIX algorithms are found to outperform the B-link algorithms. The explanation for this counter-intuitive performance is as follows: The SIX algorithms give preferential treatment to read index operations over update index operations (by allowing readers to overtake updaters during tree traversal) [SC91]. This leads to SIX providing commensurately better performance for transactions that have either no updates or only a few updates. In order to confirm this, we measured the miss percent considering only the read-only transactions. On an average, for the transaction workload considered here, close to 21% of the transactions that were generated were read only transactions. At an arrival rate of 140 transactions per second, for example, the SIX algorithms missed only 0.04% of the read only transactions whereas the B-link algorithms missed about 73% of the read only transactions. Notice that the overall miss percent of B-link algorithm at the same arrival rate is close to 79% (Figure 1) which demonstrates that the B-link algorithm does not discriminate between read-only and update transactions. In [SC91], for the SIX algorithms, the slower updaters used to clog up 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.

[9]The reason that the average lock wait times for B-link show a non-monotonic behavior is that the number of samples used in computing each average are very few (since lock waits are extremely rare for B-link).

Table 3: Average Lock Wait Time (in micro seconds)

ArrRate	B-link	OPT	SIX	X
60	0	22	1454	58314
70	0	43	5756	69365
80	1	549	27228	79252
90	1	1207	35525	89101
100	7	1602	42420	97366
110	3	2092	49356	109902
120	5	2435	52587	120131
130	7	3138	56105	128209
140	18	3392	60511	135311
150	7	3574	61702	148064
160	8	3874	63990	158935

If we view the above result from a different angle, we observe that the SIX algorithms, by giving preferential treatment to read only transactions, are applying a form of *load control*. On the other hand, the B-link algorithms tend to *saturate* the disk due to treating all transactions uniformly and therefore miss significantly more deadlines than SIX. This naturally suggests that the performance of the B-link algorithms could be improved by adding a load-control component *without* sacrificing their desirable fairness feature.

To evaluate the potential of the above idea, we developed a variant of the B-link algorithm called **LAB-link** (Load Adaptive B-link) which ensures that the utilization of the bottleneck resource is not allowed to exceed acceptable levels. This is achieved through a simple feedback mechanism that monitors the utilization at all the system resources and prevents new transactions from entering the system whenever the utilization of the bottleneck resource exceeds a prescribed amount, $MaxUtil$. Transactions which are denied entry are eventually discarded when their deadlines expire and, for the miss percent computation, are considered to be transactions that have missed their deadlines.

The choice of $MaxUtil$ in the LAB-link algorithm is dictated by two factors: If $MaxUtil$ is set too high, then the effect of the load control comes into play too late, resulting in more missed deadlines. On the other hand, if $MaxUtil$ is set too low, then again the miss percentage is increased, since transactions that could probably have made their deadlines are unnecessarily shut out from the system.

Our solution to the above problem is based on the following observation, similar to that of [HCL91]: Load control should kick in only when the system has actually started missing the deadlines of at least a small fraction of the transactions in the system. Therefore, a miss percent limit, $MinMiss$, is included in the LAB-link algorithm – at miss percents below this limit, the load control mechanism is inoperative. We conducted several experiments for a variety of transaction workloads and system configurations and empirically found that $MinMiss$ settings in the range of 8 to 10 percent and $MaxUtil$ settings in the range of 97 to 99 percent resulted in satisfactory load control behavior. For the LAB-link performance results described in this paper, $MinMiss$ is 9 percent and $MaxUtil$ is 98 percent.

The performance of the LAB-link algorithm is also shown in Figure 1. It is clear from this graph that the overload performance of LAB-link improves dramatically over B-link. As explained above, this is due to its admission control policy which ensures that the bottleneck resource (in this case, the disk) does not become saturated, thereby comfortably completing the admitted transactions.

5.4 Expt. 2: Moderate Index Contention

In our next set of experiments, the workload consisted of 100% inserts, resulting in a moderate contention environment. For this experiment, Figure 2 shows the graph of miss percent as a function of transaction arrival rate. We observe here that the SIX and X algorithms behave identically – this is only to be expected since, in the absence of readers, SIX locks do not permit any concurrent access, just like X locks. Note also that the performance of these algorithms is considerably below that of OPT and B-link.

In the corresponding (non-real-time) experiment in [SC91], the B-link algorithms performed noticeably better than the OPT algorithms. This was because the OPT algorithms suffered from a large number of restarts caused by high contention, thereby resulting in the root becoming a bottleneck due to the large number of second-pass updaters. In our experiment, however, we observe that the performance of the OPT and the B-link algorithms is very close. The reason for this surprising result is that the number of restarts for OPT in the real-time environment is, in contrast, quite small. This reduction in restarts arises from the priority feature incorporated in the CC algorithms. To verify this, we ran the same experiment without preemption for index node locks and found that the number of restarts for OPT was significantly higher than in the prioritized case. This shows that adding preemption to index latches can result in tangible performance benefits (the other index CC algorithms also exhibited similar improvement due to incorporating preemption).

The explanation for priority resulting in fewer restarts is as follows: 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

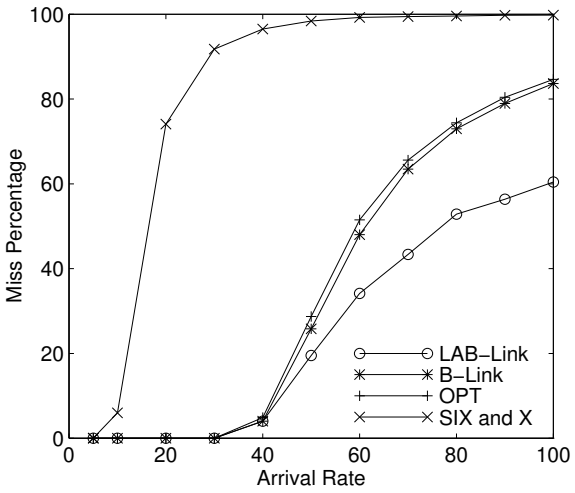

Figure 2: Moderate Index Contention

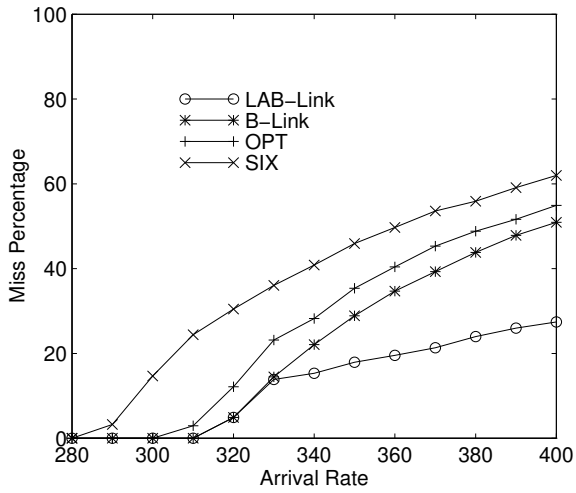

Figure 3: High Index Contention

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 second pass updater typically completes its second pass very quickly, thereby allowing only relatively few of the other transactions to see the 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.

Finally, we observe that the performance of LAB-link is again better than that of all the other algorithms. In fact, at an arrival rate of 100 transactions per second, the performance gain of LAB-link over B-link is almost 25%.

5.5 Expt. 3: High Index Contention

In the final set of experiments, the workload consisted of 25% searches and 75% appends. The appends create extremely high contention for the few right-most nodes in the tree, and as a side-effect, ensure that these nodes are permanently in the buffer pool. The keys for the searches are randomly generated and they therefore interfere only minimally with the appends. For this experiment, Figure 3 shows the graph of miss percent as a function of transaction arrival rate.

We do not explicitly plot the performance of the X algorithms, since, at an arrival rate of 300 transactions/sec, their miss percent was already close to 50%. Moving on to the SIX algorithms, we observe that they perform much worse than the OPT and B-link algorithms. This is again due to the root becoming a bottleneck, resulting in much higher average lock waiting times as compared to the OPT and B-link algorithms.

The gap between the miss percentage curves of OPT and B-link in Figure 3 is slightly wider than in the moderate data contention experiment discussed previously. The reason for the wider gap is that the probability of multiple transactions restarting here is higher than in the moderate contention case since 75% of the index actions are directed towards the righ-mtnd of the tree. Once again, just as in the moderate contention case, we experimented with having no preemption for index node locks and found that OPT without preemption had many more restarts than the prioritized OPT algorithm. The reduction in the number of restarts for the prioritized OPT algorithm makes the performance differences between the OPT and B-link algorithms to be less significant here than they would otherwise be.

Finally, notice that, in this experiment too, LAB-link performed the best (in this case, the bottleneck resource is the CPU, not the disk).

5.6 Expt. 4: Conflict Analysis

In our experiments, transactions which miss their deadlines do so due to the cumulative effect of three factors: The conflict for the data, the conflict for the index, and the conflict for the physical resources (processors, disks, memory). The relative extent to which each of these factors contributes to the miss percent is a function of the transaction workload and system configuration. We analyze below the effects of these factors with respect to the miss percent performance of B-link and LAB-link in Experiment 1 (Low Index Contention).

In all our experiments, the data conflict due to key value locking was negligible since the maximum number of transaction restarts never exceeded twelve out of the 10,000 transactions that were processed. In order to study the relative extent to which index conflict and (physical) resource conflict contributed to the miss percentage, we conducted an experiment where

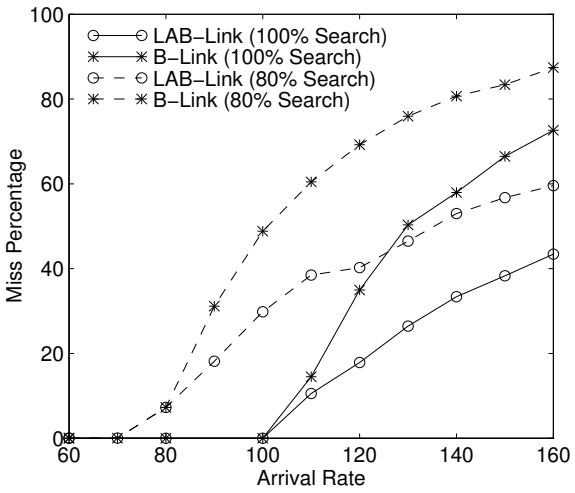

Figure 4: No Index Contention

there were no index conflicts. For this experiment, the workload and system configuration were identical to Experiment 1 (Low Index Contention) except that *all* the index operations were searches. Figure 4 shows the performance of B-link and LAB-link for this full-search workload and the corresponding figures for the mixed workload of Experiment 1. Note that the performance of B-link for the full-search workload is significantly better than its performance with the mixed workload. The difference corresponds to the performance drop due to data conflicts at index nodes. This difference is quite substantial (about 20%) even at very high arrival rates and demonstrates that data conflict at the index nodes is *significant* even for B-link algorithms. A similar performance difference is observed for the LAB-link algorithm as well.

5.7 Expt. 5: Performance Limits

In the previous experiment, we observed that both index contention and resource contention had a significant role to play in determining the miss percentage. Index contention is primarily determined by the transaction workload characteristics and since the workload is typically decided by the users, the only unilateral performance improvement option available for the RTDBS designer is to reduce resource contention. One method of reducing resource contention is to purchase more and/or faster resources. In the experiment described below, the miss percent performance is evaluated for an "infinite resource" system, that is, a system where there is no queueing for resources. While abundant resources are usually not to be expected in conventional database systems, they may be more common in RTDBS environments since many real-time systems are sized to handle transient heavy loading. This directly relates to the application domain of RTDBSs, where functionality, rather than cost, is often the driving consideration.

Table 4: Miss Percent with Infinite Resources

ArrRate	B-link	OPT	SIX
1000	0.01	0.01	0.01
5000	0.04	0.04	65.90
10000	0.05	0.07	77.33
15000	0.08	0.09	77.72
20000	0.18	0.14	77.74

We conducted an experiment to evaluate the miss percent performance that could be achieved for the transaction workload of Experiment 1 (Low Index Contention) in the absence of resource contention. For this experiment, the entire index tree was resident in memory and the number of CPUs was infinite, and the miss percent was therefore determined solely by index contention. Note that this means that the performance numbers observed in this experiment capture the *best* performance that each index concurrency control algorithm can deliver for the chosen transaction workload.

The performance in this experiment is shown in Table 4 for the B-link, OPT and SIX algorithms (LAB-link is identical to B-link here since the resource utilization is very low). Note that the OPT and B-link algorithms manage to complete almost all the submitted transactions, even at an arrival rate of 20,000. In contrast, SIX algorithms miss close to 77% of their transactions at higher arrival rates. However, the miss percent of SIX algorithms does not degrade beyond 77%. This is yet another indication of the fact that SIX algorithms preferentially complete all the read only transactions (recall that the percentage of read only transactions is close to 21%).

5.8 Modeling Undos

While discussing the B-tree model in Section 4, we mentioned that undos of index updates made by discarded (aborted) transactions was not included in our model. It seems reasonable to assume that the number of undos to be done is directly related to the transaction miss percent, that is, a higher miss percent leads to more undos. Given this, we expect that if we implemented undos in our model, the difference in performance of the algorithms seen in the above experiments would increase *even further*. This is because the poorly performing algorithms would have *more* cleanup work to do than the better algorithms and therefore miss even more transaction deadlines. In summary, there is a positive feedback between the undo overhead and the miss percentage.

6 Conclusions

In this paper, we investigated the problem of index concurrency control for real-time database systems supporting transactions with firm deadlines. Using a detailed simulation model of an RTDBS, we studied the deadline miss percent performance of real-time variants of three different classes of index CC algorithms: Bayer-Schkolnick, Top-Down and B-link, under a range of workloads and operating conditions.

The experimental results showed that two factors characteristic of the (firm) real-time domain: addition of priority and discarding of late transactions, significantly affected the performance of the index concurrency control algorithms. In particular, B-link algorithms missed many more deadlines at high loads as compared to lock-coupling algorithms. This was in contrast to conventional DBMS where they (B-link) always exhibited the best throughput performance. In fact, the very reason for their good performance in conventional DBMS (full resource utilization) turned out to be a liability here. Secondly, the optimistic algorithms performed almost as well as the B-link algorithms even under high index contention conditions (in contrast to conventional DBMS). This was because prioritization of transactions caused a marked decrease in the number of index operation restarts. In short, these experiments show that the performance behaviors exhibited by index CC algorithms in conventional DBMS cannot be blindly assumed to be valid in the corresponding real-time situation also.

We introduced the LAB-link algorithm, which augmented the basic B-link algorithm with a simple load-control system to ensure that the bottleneck resource was not saturated. The LAB-link algorithm significantly reduced the miss 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 load control in index management in real-time database systems. Interestingly, such need for load control has also been identified in other modules of real-time database systems (e.g. [HCL91, PCL94]).

Our current study was limited to point (single key) index operations. In our future work, we plan to extend our study to include range (multiple key) operations. In its current implementation, LAB-link uses a utilization-based load control. We also plan to evaluate the performance that would be obtained by implementing load-control through *priority assignments* (as done, for example, in Adaptive Earliest Deadline [HCL91]).

Acknowledgements

We thank Navneet Yadav, Gopal Reddy and Raghu Jorapur, for their help in setting up the environment for conducting our experiments. We thank Prof. N. Balakrishnan, Ramesh Gupta and Tulika Pradhan, for their help with the tools and the computing facility for preparing the final version of this paper. The work of J. R. Haritsa was supported in part by a research grant from the Dept. of Science and Technology, Govt. of India.

References

[AEJ92] D. Agrawal, A. El Abbadi and R. Jeffers. Using Delayed Commitment in Locking Protocols for Real-Time Databases. In *Proc. of ACM SIGMOD Conference*, June 1992.

[AG92] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. In *ACM Trans. on Database Systems*, September 1992.

[Bil86] A. Biliris. A comparative study of concurrency control methods in B-trees. In *Proc. of the Agean Workshop on Computing*, Loutraki, July 1986.

[BS77] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. In *Acta Informatica*, 9, 1977.

[Com79] D. Comer. The Ubiquitous B-Tree. In *ACM Computing Surveys*, 11(4), 1979.

[Gra79] J. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, eds. R. Bayer, R. Graham and G. Seegmuller, Springer-Verlag, 1979.

[HCL91] J. Haritsa, M. Carey, and M. Livny. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proc. of 1991 IEEE Real-Time Systems Symp.*, December 1991.

[HCL92] J. Haritsa, M. Carey and M. Livny. Data Access Scheduling in Firm Real-Time Database Systems. In *Journal of Real-Time Systems*, September 1992.

[HJC93] D. Hong, T. Johnson and S. Chakravarthy. Real-Time Transaction Scheduling: A Cost Conscious Approach. In *Proc. of ACM SIGMOD Conf.*, May 1993.

[JS90] T. Johnson and D. Shasha. A framework for the performance analysis of concurrent B-tree algorithms. In *Proc. of ACM Symp. on Principles of Database Systems*, April 1990.

[KW82] Y. Kwong and D. Wood. A new method for concurrency in B-trees. *IEEE Trans. on Software Engineering*, SE-8(3), May 1982.

[LS86] V. Lanin and D. Shasha. A Symmetric Concurrent B-tree Algorithm. In *Proc. of Fall Joint Computer Conf.*, 1986.

[LS92] D. B. Lomet and B. Salzberg. Access method concurrency with recovery. In *Proc. of ACM SIGMOD Conf.*, June 1992.

[LY81] P. Lehman and S. Yao. Efficient Locking for Concurrent Operations on B-trees. In *ACM Trans. on Database Systems*, 6(4), 1981.

[Moh90] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes. In *Proc. of VLDB Conf.*, September 16-mt

[MR85] Y. Mond and Y. Raz. Concurrency Control in B^+-trees Databases Using Preparatory Operations. In *Proc. of VLDB Conf.*, September 1985.

[ML92] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-Ahead Logging. In *Proc. of ACM SIGMOD Conf.*, June 1992.

[PCL94] H. Pang, M. Carey and M. Livny. Managing Memory for Real-Time Queries. In *Proc. of ACM SIGMOD Conf.*, May 1994.

[Sag85] Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences*, 33(2), 1985.

[SC91] V. Srinivasan and M. Carey. Performance of B-Tree Concurrency Control Algorithms. In *Proc. of ACM SIGMOD Conf.*, May 1991.

[SG88] D. Sasha and N. Goodman. Concurrent search structure algorithms. *ACM Trans. on Database Systems*, 13(1), March 1988.

[SRL87] L. Sha, R. Rajkumar and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Technical Report CMU-CS-87-181*, Depts. of CS, ECE and Statistics, Carnegie Mellon University, 1987.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.