# ROBUST REAL-TIME INDEX CONCURRENCY CONTROL

S.R. Narayanan  Brajesh Goyal [1]  Jayant Haritsa  S. Seshadri [1]

**Technical Report**

**TR-1997-03**

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

http://dsl.serc.iisc.ernet.in

[1] Dept. of Computer Science and Engineering, Indian Institute of Technology, Mumbai 400076, India

# Abstract

Real-time database systems are expected to rely heavily on **indexes** to speed up data access and thereby help more transactions meet their deadlines. Accordingly, high-performance *index concurrency control* (ICC) protocols are required to prevent contention for the index from becoming a bottleneck. We took a first step towards achieving this objective in a recent simulation-based study that evaluated the performance of a representative set of B-tree ICC protocols for real-time applications with firm deadlines. A load-adaptive variant of the B-link protocol, called LAB-link, was found to provide the best real-time performance in that study. In this paper, we extend and improve on this prior research in several ways: First, we present a new real-time ICC protocol called **AED-link**, which employs an adaptive feedback-based admission control mechanism. Our results show that it provides significantly better and more robust performance than LAB-link. Second, we make a detailed study of the fairness characteristics of the ICC protocols. Our results indicate that AED-link provides almost ideal fairness for a wide range of workloads. Third, we explicitly model the "undos" of index actions of aborted transactions. An unexpected and interesting result here is that performing undos usually has only marginal adverse impact on real-time performance and can, in a few scenarios, actually result in *better* performance. Finally, we model the performance effects of range index operations whereas the previous study was limited to point index operations.

# 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 complete processing of transactions before their deadlines expire. To achieve this objective, it appears reasonable to expect that an RTDBS will make extensive use of **indexes** to quickly access data. A potential problem, however, is that the contention among transactions concurrently using the index may *itself* form a performance bottleneck. In this situation, employing an "off-the-shelf" index concurrency control protocol that has not been specifically designed for the real-time environment would only aggravate the problem and result in many more transactions missing their deadlines. Therefore, there is a clear need for developing index concurrency control (ICC) protocols that are *tuned* to the objectives of the real-time domain.

## Previous Study

We took a first step towards addressing the above issue in a recent study whose results are reported in [5]. Our study considered the design and performance behavior of real-time ICC protocols for **B-tree** indexes [3] (we use the term B-tree to refer to the $B^+$ variant in which all data key values are stored at the leaf nodes). In particular, we investigated the ICC problem in the context of real-time applications with "firm deadlines" [7] – for such applications, completing a transaction after its deadline has expired is of no utility and may even be harmful. The performance metric for these applications is the steady-state percentage of transaction deadlines that are missed. Using a detailed simulation model of a firm-deadline RTDBS, we studied the deadline miss percent performance of real-time variants of three different classes of B-tree ICC protocols: *Bayer-Schkolnick* [2], *Top-Down* [10, 14] and *B-link* [10, 11, 15], for a range of real-time workloads and system operating conditions.

The experimental results indicated that the performance characteristics of the real-time version of an ICC protocol could be significantly different from the performance of the original protocol in a conventional (non-real-time) database system. For example, B-link algorithms, which are reputed to provide the best overall performance in conventional database systems [9, 16], performed poorly under heavy real-time loads. In fact, the very reason for their good performance in conventional DBMS (full resource utilization) turned out to be a liability here. To address this problem, we proposed the **LAB-link** algorithm, which augmented the basic B-link algorithm with a simple load-control system that ensured that the bottleneck resource was not saturated. The LAB-link algorithm provided the best real-time performance among all the evaluated protocols for a variety of workloads and system configurations.

## Limitations of Previous Study

The above study was a useful first step towards addressing the basic issues of real-time index concurrency control. It had several limitations, however: First, the LAB-link algorithm, as we will show in this paper, is effective for only a restricted class of operating environments. Moreover, its performance is sensitive to the choices made for its algorithmic parameters. Second, the *fairness* aspects of the protocols, in terms of bias for or against a particular class of transaction, were

not considered quantitatively. Fairness is a particularly important aspect in RTDBS that serve "public applications" such as stock markets and telecommunications. Third, the *undos* of the index actions of aborted transactions were not modeled. It was surmised that there is a direct positive feedback between the undo overhead and the miss percentage and therefore incorporating undo actions into the model would only significantly increase, but not qualitatively alter, the observed performance differences between the various protocols. However, as we will show in this paper, the impact of undos is not so simply characterized and that unexpected effects can show up in certain situations. Finally, the study was limited to point (single key) index operations. *Range* (multiple key) index operations, which are a basic form of index access, were not modeled at all.

### Our Contributions

We address the above-mentioned limitations of the previous study in this paper. In particular, we extend and improve on this prior research in several ways: First, we present a new real-time ICC protocol called **AED-link**, which employs an adaptive feedback-based admission control mechanism. Our results show that it provides both significantly better and more robust performance than LAB-link. Second, we evaluate the fairness characteristics of all the ICC protocols. Our results indicate that AED-link provides the maximum fairness for a wide range of index contention levels. Third, we explicitly model the undos of index actions of aborted transactions. An unexpected and interesting result here is that performing undos usually has only marginal adverse impact on real-time performance and can, in a few scenarios, actually result in *better* performance. Finally, we model the performance effects of range index operations.

### Organization

The remainder of this paper is organized as follows: The B-tree concurrency control algorithms evaluated in this study are outlined in Section 2. The adaptive load-control protocols, LAB-link and AED-link, are described in Section 3. Related issues such as incorporating real-time priorities in the ICC protocols and guaranteeing transaction serializability are discussed in Section 4. The performance model is described in Section 5, and the results of the simulation experiments are highlighted in Section 6. Finally, in Section 7, we summarize the conclusions of the study.

## 2 B-tree ICC Protocols

In this section, we describe the set of B-tree index 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 [3, 16].

The 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 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 ICC protocols maintain index consistency in the face of concurrent transaction accesses. This is achieved through the use of locks[1] 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 [4].

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 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, releasing the parent lock if the new node is found to be safe.

There are three well-known classes of B-tree ICC protocols: 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.

## 2.1 Bayer-Schkolnick Algorithms

We consider three algorithms in the Bayer-Schkolnick class [2] 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 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.

[1] 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 4.

## 2.2 Top-Down Algorithms

In the Top-Down class of algorithms (e.g. [10, 14]), 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 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 [10, 11, 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 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 reach the appropriate node. Such a sideways traversal is called a *link-chase*. Merges are also done in two steps [10], via a half-merge followed by the appropriate entry deletion at 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, limit 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 [16], and was found to have the best performance of all the above-mentioned algorithms with respect to transaction throughput.

# 3 Load-Control-Based ICC Protocols

As mentioned in the Introduction, the relative performance of the real-time variants of the ICC protocols described in the previous section was evaluated in [5]. It was found there that B-link algorithms missed many more deadlines under heavy loads as compared to lock-coupling algorithms. This was in contrast to conventional DBMS where they always exhibited the best throughput performance [9, 16]. The reason for B-link's degraded performance in the real-time domain was that it tended to *saturate* the bottleneck resource (usually the disk) leading to many more missed deadlines. Based on this observation, a variant of the B-link protocol called **LAB-link** (Load Adaptive B-link) was proposed, wherein a simple load-control component was incorporated to ensure that the utilization of the bottleneck resource is not allowed to exceed acceptable levels. In the remainder of this section, we briefly describe the LAB-link algorithm and also present **AED-link**, our new load-control-based ICC protocol.

## 3.1 LAB-link

The LAB-link algorithm achieves its load control 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 that 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 setting of the $MaxUtil$ parameter has to be made carefully: 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. To address this issue, a miss percent limit, $MinMiss$, is included in the LAB-link algorithm – at miss percents below this limit, the load control mechanism is *inoperative*. The limit ensures that load control kicks in only when the system has actually started missing the deadlines of at least a small fraction of the transactions in the system. For the experiments described in [5], setting $MaxUtil$ and $MinMiss$ to 98 percent and 9 percent, respectively, was empirically found to provide the best load control with respect to minimizing the miss percentage.

## 3.2 AED-link

As an alternative to the LAB-link algorithm described above, we have developed a new load-control-based ICC protocol called **AED-link**. The AED-link protocol is based on incorporating a modified version of the *Adaptive Earliest Deadline* (AED) priority assignment mechanism proposed in [6]. The mechanism is based on the following observation: The Earliest Deadline policy *minimizes* the number of missed transaction deadlines when the system is lightly loaded [8]. At heavier loads, however, its performance steeply degrades; in fact, it has been observed to perform worse than even *random* scheduling in this region [6]. The goal of the AED policy is therefore to stabilize the overload performance of Earliest Deadline without sacrificing its light-load virtues. It does this by using a feedback process to estimate the number of transactions that

are *sustainable* under an Earliest Deadline schedule.

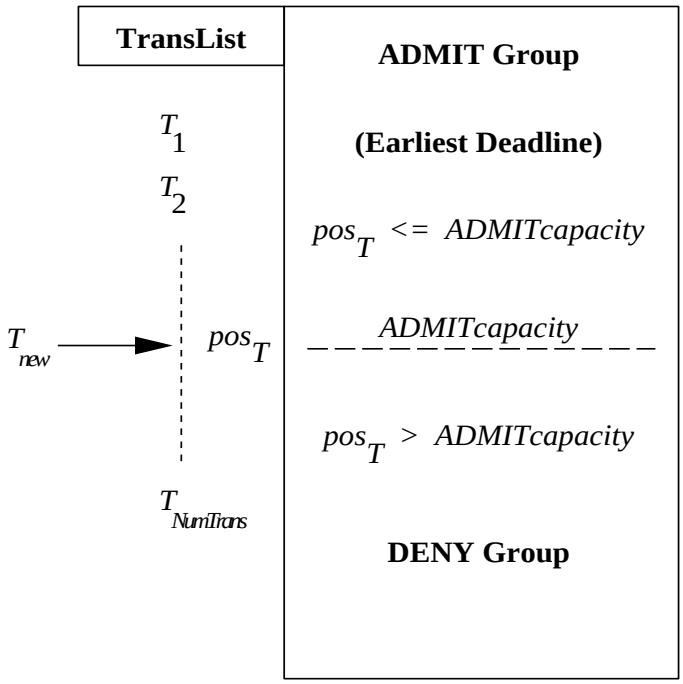



Figure 1: AED-link Admission Control

### 3.2.1 Group Assignment

In the AED-link protocol, transactions entering the system are dynamically split into two groups, ADMIT and DENY, as illustrated in Figure 1. The assignment of transactions to groups is done in the following manner: Each newly-arrived transaction is assigned a random integer key, $I_T$. This transaction is then inserted into a *key-ordered* list of transactions in the system and the position $pos_T$ of this transaction in the list is noted. If the value of $pos_T$ is less than or equal to *ADMITcapacity*, which is a dynamic control variable of the AED-link protocol, the transaction is assigned to the ADMIT group; otherwise, it is assigned to the DENY group.

The goal of the splitting process is to collect in the ADMIT group the *largest* set of transactions that can be completed before their deadlines. It tries to achieve this by dynamically controlling the size of the ADMIT group, using the *ADMITcapacity* variable. Then, by having an Earliest Deadline priority ordering *within* the ADMIT group, the algorithm incorporates the observation discussed earlier. Transactions that cannot be accommodated in the ADMIT group are estimated to miss their deadlines and are therefore assigned to the DENY group. These transactions are denied entry to the system and are eventually discarded when their deadlines expire. For the miss percent computation, they are considered to be transactions that have missed their deadlines (as in LAB-link).

### 3.2.2 Feedback Process

The key to the successful operation of the above mechanism is the proper setting of the *ADMITcapacity* control variable. In AED-link, the setting is chosen using a feedback process that has two parameters, *ADMITbatch* and *ALLbatch*, and uses two system output measurements, *HitRatio(ADMIT)* and *HitRatio(ALL)*. The process operates as follows: Assume that the *ADMITcapacity* value has just been set. The next set of *ADMITbatch* transactions that are assigned to the ADMIT group are marked with a special label. At the RTDBS output, the completion status (in-time or deadline-missed) of these specially-marked transactions is monitored. When the last of these *ADMITbatch* transactions exits the system, *HitRatio(ADMIT)* is measured as the fraction of these transactions that completed before their deadlines. Concurrently, *HitRatio(ALL)* is measured as the fraction of the last *ALLbatch* transactions that completed before their deadlines.

Using the above measurements, and denoting the number of transactions currently in the system by *NumTrans*, the *ADMITcapacity* control variable is reset with the following two-step computation:

*(1)* $ADMITcapacity := \lceil HitRatio(ADMIT) * ADMITcapacity * 1.05 \rceil$;
*(2)* *if* $HitRatio(ALL) < 0.95$ *then*
$ADMITcapacity := Min(ADMITcapacity, \lceil HitRatio(ALL) * NumTrans * 1.25 \rceil)$;

STEP 1 of the *ADMITcapacity* computation incorporates the feedback process in the setting of this control variable. By conditioning the new *ADMITcapacity* setting based on the observed hit ratio in the ADMIT group, the size of the ADMIT group is adaptively changed to achieve a 1.0 hit ratio. Our goal, however, is not just to have a *HitRatio(ADMIT)* of 1.0, but to achieve this goal with the *largest* possible transaction population in the ADMIT group. It is for this reason that STEP 1 includes a 5 percent expansion factor. This expansion factor ensures that the *ADMITcapacity* is steadily increased until the number of transactions in the ADMIT group is large enough to generate a *HitRatio(ADMIT)* of 0.95, that is, a five percent miss level. At this point, the transaction population size in the ADMIT group is close to the "right" value, and the *ADMITcapacity* remains stabilized at this setting (since $0.95 * 1.05 \simeq 1.0$).

STEP 2 of the above *ADMITcapacity* computation is used to deal with the special case where the system suddenly makes a *transition* from an extended period of being lightly-loaded to an overloaded condition. If the system suddenly becomes overloaded, it would take the system a lot of time to get the *ADMITcapacity* down to the actual value by STEP 1. In the meantime, since the *ADMITcapacity* is very large, all transactions would be accommodated in the ADMIT group, which reduces to the unstable high-miss region of Earliest Deadline. To take care of this undesirable situation, the *ADMITcapacity* is brought down to the appropriate value quicker than STEP 1 by STEP 2. The HitRatio(ALL) value becoming less than 0.95 signals this overloaded situation. In this case, the *ADMITcapacity* is not allowed to go more than an upper bound which is set to 25% greater than the *right* value of *ADMITcapacity* that is estimated by computing the number of transactions that are currently meeting their deadlines (the value 25% was arrived at based on the assumption that the estimate is close enough to the *right ADMITcapacity* value). After *ADMITcapacity* is brought down close to the *right* value, STEP 1 takes over as a fine tuning mechanism.

At system initialization time, *ADMITcapacity* is set equal to the database administrator's

estimate of the number of concurrent transactions that the RTDBS can handle without missing deadlines. Note that this estimate does not have to be accurate; even if it were grossly wrong, it would not impact system performance in the long run. The error in the estimate only affects how long it takes the *ADMITcapacity* control variable, at system startup time, to reach its steady state value.

### 3.3 Discussion

We have described above two types of load-control-based ICC protocols, LAB-link and AED-link. We now informally motivate as to why AED-link appears to be preferable to LAB-link:

- LAB-link monitors *system internal* characteristics in its feedback process whereas AED-link monitors *system output* values. Moreover, AED-link bases its feedback *directly* on the performance metric of interest, the miss percentage. This makes the feedback process more responsive to the desired goal of minimizing the miss percentage.

- The feedback process in LAB-link is based on monitoring *resource contention* levels. The problem with this approach is that it is ineffective in database systems where the main bottleneck is *data contention* or *metadata (index) contention* – this scenario is especially plausible in RTDBS since they are usually sized to handle transient heavy loading and can therefore be expected to have ample resources. AED-link, on the other hand, does not suffer this problem since it monitors *system performance* levels, thus making it impervious to the type of contention existing in the system.

- The applicability of the load control policy of LAB-link is limited to the set of ICC protocols that tend to *saturate* the resources. With protocols that involve more blocking at index nodes and thereby less utilization of system resources, it becomes ineffective. In contrast, the AED approach can be used with any ICC protocol since it only needs to monitor the output performance for its feedback process to operate.

- LAB-link applies a simplistic form of load-control that merely attempts to limit resource utilization. In contrast, AED-link is more "rational" in that it is based on well-established *real-time* principles – in particular, it makes use of the idiosyncratic behavior of Earliest Deadline.

- As we will show in our experiments, the performance of LAB-link is sensitive to the values chosen for its *MaxUtil* and *MinMiss* parameters whereas AED-link is relatively robust to the values chosen for *ADMITbatch* and *ALLbatch*.

## 4 Related Issues

In the previous sections, we described the functioning of the various ICC protocols evaluated in our study. To complete this description, there are a few related issues that need to be addressed and we cover these in this section. In particular, we discuss how real-time priorities are incorporated

into the ICC protocols, how transaction serializability is ensured and, finally, how the undos of the index actions of aborted transactions are implemented.

## 4.1 Index Node Locks

We have incorporated priority 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 *preempted* and the requesting transaction is awarded the lock. The lower priority transactions then reperform, from the beginning, their *current* index operation (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* [13]. 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. The experiments of our previous study, however, showed that adding preemption does have an appreciable performance effect [5].

## 4.2 Locking for Transaction Serializability

The performance metric of missed deadlines applies only to *entire* transactions, not to individual index actions. We therefore need to consider transactions that 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 well-known ARIES *Next-Key-Locking* protocol [12] to provide transaction data concurrency control (DCC). In formulating this simplified algorithm, we assume that all indexes are unique.

The next-key-locking-based DCC protocol that we use is 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 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 smallest key greater than the last key in the accessed range, that is, the *next key* to the last key in the range. Inserts acquire an X (exclusive) lock on the next key (next 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 key being deleted, acquire an X key lock on the next key (next with respect to the deleted key), 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). For our study, as in [5], we use a real-time version of 2PL called 2PL-HP [1] which incorporates a priority mechanism similar to that described above for the index concurrency control algorithms. An important difference, however, is that transactions restarted due to key-value-lock preemptions have to commence the *complete* transaction once again, not just the current index operation.

### 4.3 Modeling Undos

In an RTDBS, executing transactions may be aborted due to priority resolution of data conflicts (the 2PL-HP protocol mentioned above), or due to missing their deadlines. For aborted transactions, it is necessary to *undo* any effects they may have had on the index structure. As mentioned earlier, undos were not considered in our previous study. We explicitly model undos here and implement them in the manner described below.

We use the term *undo transaction* to refer to transactions that require undoing of their index actions. The first step taken by an undo transaction is to release all latches held for the current index action. Every index action previously completed by the transaction is then examined in *reverse* chronological order. For a search, the corresponding key-value lock is released, whereas for a range search, *all* the key-value locks obtained during the search process are released. The undo of an update action involves performing its *compensating* action (insert for a delete, delete for an insert or append) on the B-tree. On completion of this compensating action, the corresponding key value lock is released.

To ensure that undo transactions complete soon and quickly bring the index back to a consistent state for use by other transactions, 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 normal processing.

## 5 Simulation Model and Methodology

In the previous section, we discussed various index concurrency control protocols and their real time versions. To evaluate the performance of these protocols, we used a detailed simulation model of a firm-deadline real-time database system. The model is the same as that described in [5], which is based on a loose combination of the database model of [7] and the B-tree system model of [16]. 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.

### 5.1 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 (point or range)*, *insert*, *delete* or *append* of a key value. After each index operation, the corresponding data access

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 |
| RangeKeys | Size of range search | $1 - \infty$ |
| 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 | 50 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 | $1 - \infty$ |

is made (for a range search, the data access is made for each key-value returned). 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 (of course, if the transaction is currently physically making an update to a node, then the abort is delayed until the update is completed, as discussed in Section 4.1).

The *ArrRate* parameter specifies the mean rate of transaction arrivals. The number of index operations 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, searches can be either point or range operations, whereas all updates are point operations. For range searches, rather than specifying a range from the domain space, we specify it in terms of the number of keys that are to be retrieved. This number is set using the *RangeKeys* parameter.

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[2].

### 5.2 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 has made have to be undone to maintain index consistency.

### 5.3 Resource Model

The physical resources in our model consist of 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[3].

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.

## 6 Experiments

In this section, we present the results from our simulation experiments comparing the performance of the various ICC protocols in a firm-deadline RTDBS environment. The transaction priority assignment scheme used in all the experiments reported here is *Earliest Deadline*, wherein transactions with earlier deadlines have higher priority than transactions with later deadlines. Of course, in LAB-link and AED-link, this assignment is used in conjunction with their respective admission control policies.

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. Inserts can use all key values in the key space that are not exact

[2]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 misses its deadline.

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

multiples of 3, whereas deletes can use the remaining keys (i.e., exact multiples of 3). In contrast to updates, 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 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.

## 6.1 Performance Metric

The primary 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 20000 transactions were processed by the system. Only statistically significant differences are discussed here.

In this study, we introduce two additional performance metrics, *SizeFairness* and *TypeFairness*. The *SizeFairness* factor tries to capture the extent to which bias is exhibited towards transactions based on their sizes and is evaluated using the formula

$$SizeFairness = \frac{Average\ Size\ of\ Completed\ Transactions}{Average\ Size\ of\ Input\ Transactions}$$

In similar fashion, the *TypeFairness* factor tries to capture the extent to which bias is exhibited towards transactions based on their type (read-only or update). It is computed as

$$TypeFairness = \frac{Number\ of\ Completed\ Readers\ /\ Number\ of\ Completed\ Transactions}{Number\ of\ Input\ Transactions\ /\ Number\ of\ Input\ Transactions}$$

With these formulations, a protocol is ideally fair if *SizeFairness* and *TypeFairness* are both equal to *one*. (Note that the term "Completed" in the above formulae refers to transactions that are completed before their deadlines and does not include killed transactions.)

The simulator was instrumented to generate a host of other statistical information, including resource utilizations, number of index node splits and merges, 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 ICC protocols under various workloads and system conditions.

## 6.2 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. For all the experiments described here, the resource parameter settings are: $NumCPUs = 1$, $NumDisks = 8$, and $NumBufs = 250$. The parameter settings for the LAB-link algorithm are the same as those used in [5]: $MaxUtil = 98$ % and $MinMiss = 9$ %. Similarly, for the AED-link algorithm, the parameter settings are the same as those used in [6]: $ADMITbatch = 20$ and $ALLBatch = 20$.

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.

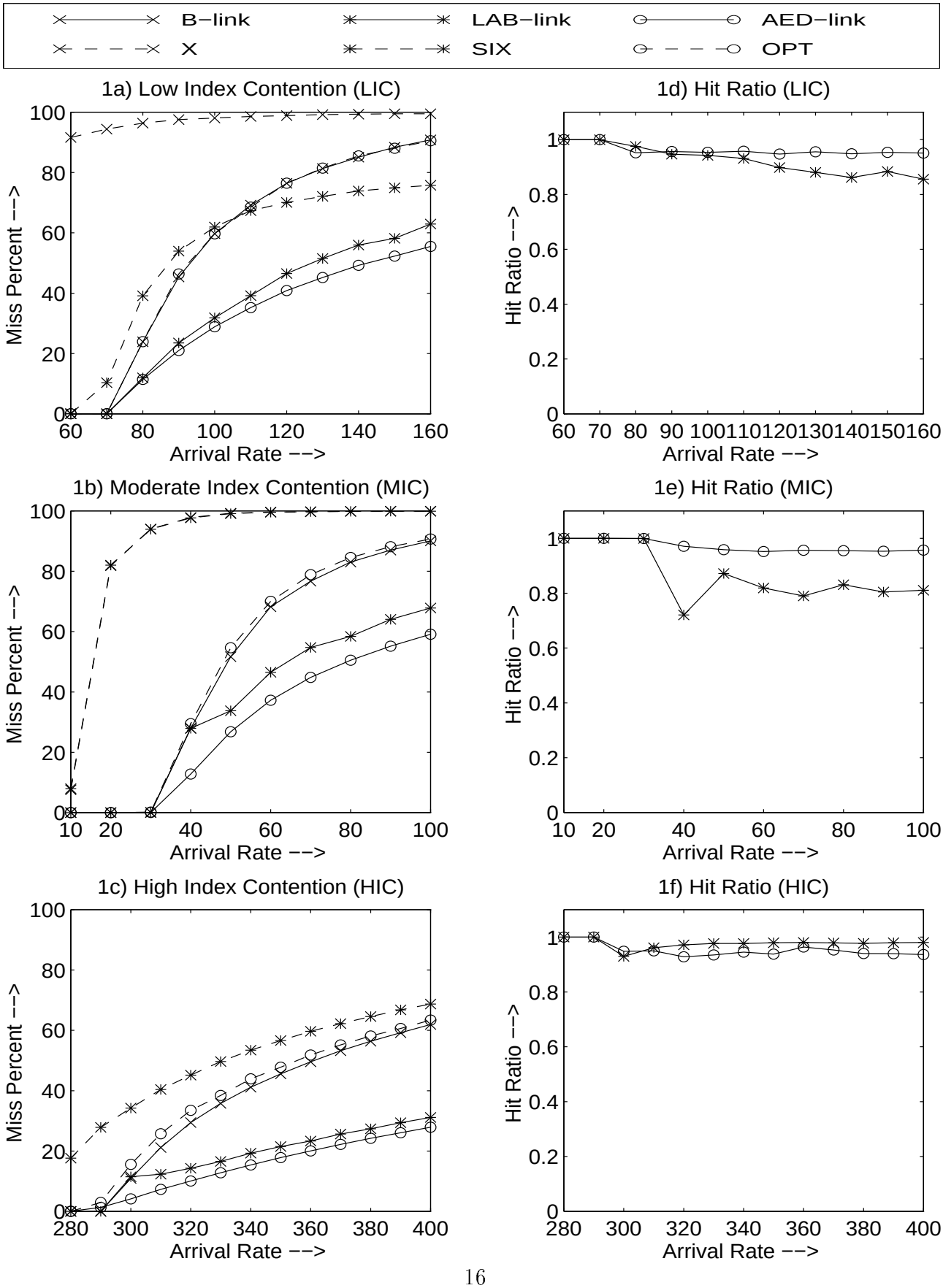
In our previous study, three representative experiments which correspond to system conditions of *low index contention* (80% point searches, 10% inserts and 10% deletes), *moderate index contention* (100% inserts), and *high index contention* (25% point searches, 75% appends), respectively, were presented. We have used the same workloads for most of the experiments described here, and will refer to them hereafter as LIC, MIC and HIC, respectively.

We present our experimental results in three stages: First, we evaluate the MissPercent and fairness performance of the newly proposed AED-link ICC protocol. Then, we consider the performance effects of incorporating undos of index actions of aborted transactions. Finally, we profile ICC protocol performance for workloads that include range searches.

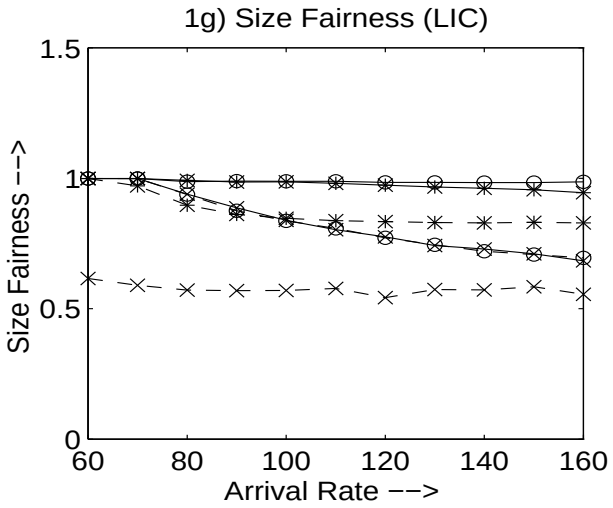
## 6.3 Experiment 1: Load Control

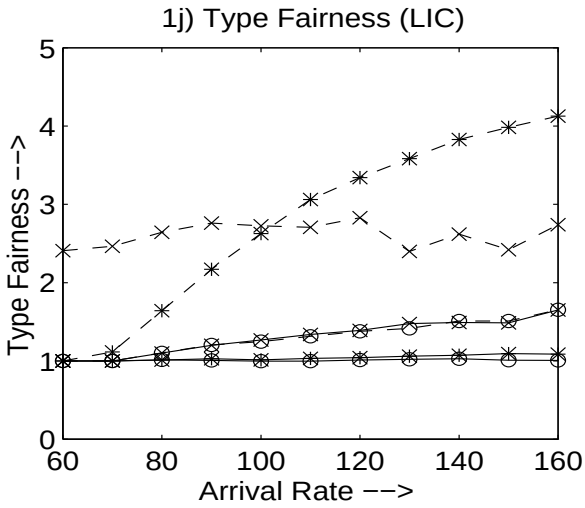
In Figures 1a, 1b and 1c, we profile the performance of the new protocol, AED-link, in comparison to the protocols of the previous study, for the LIC, MIC and HIC environments, respectively. (We will only comment on the results for AED-link here since the results for the other protocols have already been discussed in [5]). We observe first, in these figures, that the performance of AED-link is vastly superior to that of the classical protocols (X, SIX, OPT and B-link). More interestingly, its performance is noticeably better than that of LAB-link also, especially in the MIC and HIC environments (Figures 1b and 1c). The primary reason for the difference is the following: LAB-link takes only *resource* contention into account in its load control process. This means that when *index* contention is the primary bottleneck, as is the case in the MIC and HIC environments, LAB-link does not kick in even when there are already a significant number of transactions missing their deadlines due to index contention. In contrast, AED-link smoothly applies load control with increasing miss percentage since it is based on system output metrics rather than on internal parameters, making its performance impervious to the source of the bottleneck.

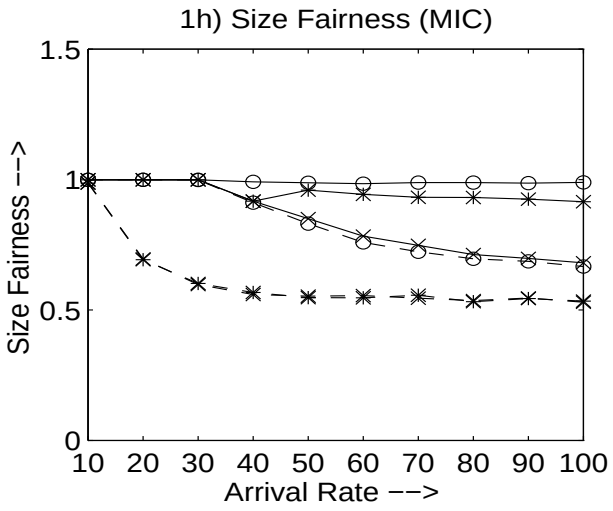
More importantly, note that LAB-link's performance is not as good as AED-link's even *after* resource contention has risen to an extent that LAB-link's load-control process has become operative. This is because it tends to *overestimate* the number that can be successfully let into

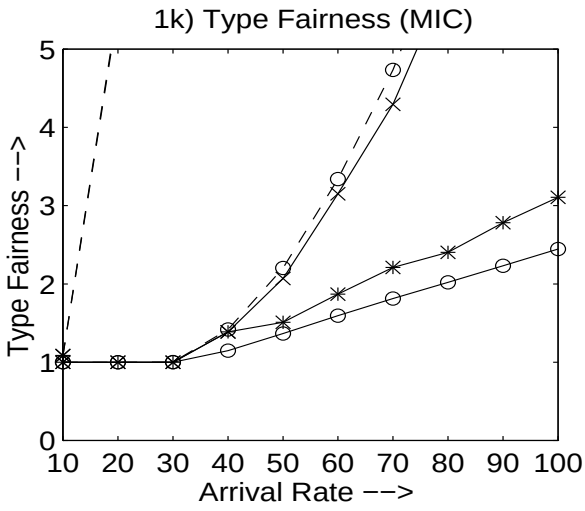

B−link
LAB−link
AED−link
X
SIX
OPT
1a) Low Index Contention (LIC)
Miss Percent −−>
Arrival Rate −−>
1d) Hit Ratio (LIC)
Hit Ratio −−>
Arrival Rate −−>
1b) Moderate Index Contention (MIC)
Miss Percent −−>
Arrival Rate −−>
1e) Hit Ratio (MIC)
Hit Ratio −−>
Arrival Rate −−>
1c) High Index Contention (HIC)
Miss Percent −−>
Arrival Rate −−>
1f) Hit Ratio (HIC)
Hit Ratio −−>
Arrival Rate −−>

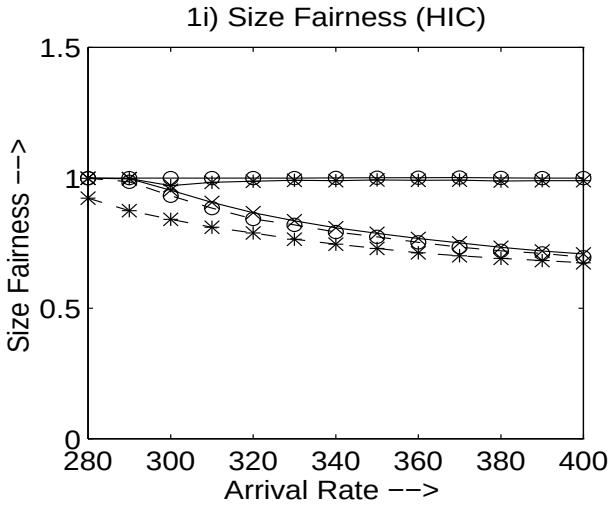

B−link
LAB−link
AED−link
X
SIX
OPT

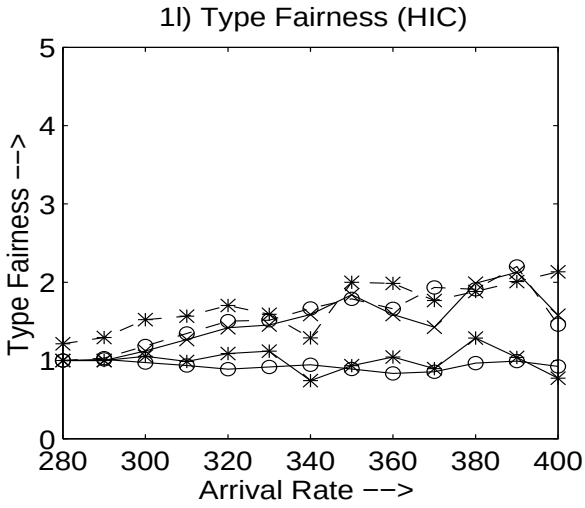

1g) Size Fairness (LIC)
Size Fairness −−>
Arrival Rate −−>
0
0.5
1
1.5
60
80
100
120
140
160


1j) Type Fairness (LIC)
Type Fairness −−>
Arrival Rate −−>
0
1
2
3
4
5
60
80
100
120
140
160


1h) Size Fairness (MIC)
Size Fairness −−>
Arrival Rate −−>
0
0.5
1
1.5
10
20
30
40
50
60
70
80
90
100


1k) Type Fairness (MIC)
Type Fairness −−>
Arrival Rate −−>
0
1
2
3
4
5
10
20
30
40
50
60
70
80
90
100


1i) Size Fairness (HIC)
Size Fairness −−>
Arrival Rate −−>
0
0.5
1
1.5
280
300
320
340
360
380
400


1l) Type Fairness (HIC)
Type Fairness −−>
Arrival Rate −−>
0
1
2
3
4
5
280
300
320
340
360
380
400

the system, again since it only considers resource contention. This is confirmed in Figures 1d-f, where we show the "hit ratio" in the ADMIT group for both AED-link and LAB-link for the LIC, MIC, HIC experiments. This figure clearly shows that while AED-link successfully maintains a hit ratio close to 0.95 as designed for, LAB-link overadmits transactions to the extent of reducing its hit ratio to only around 0.80, and consequently suffers significant performance degradation.

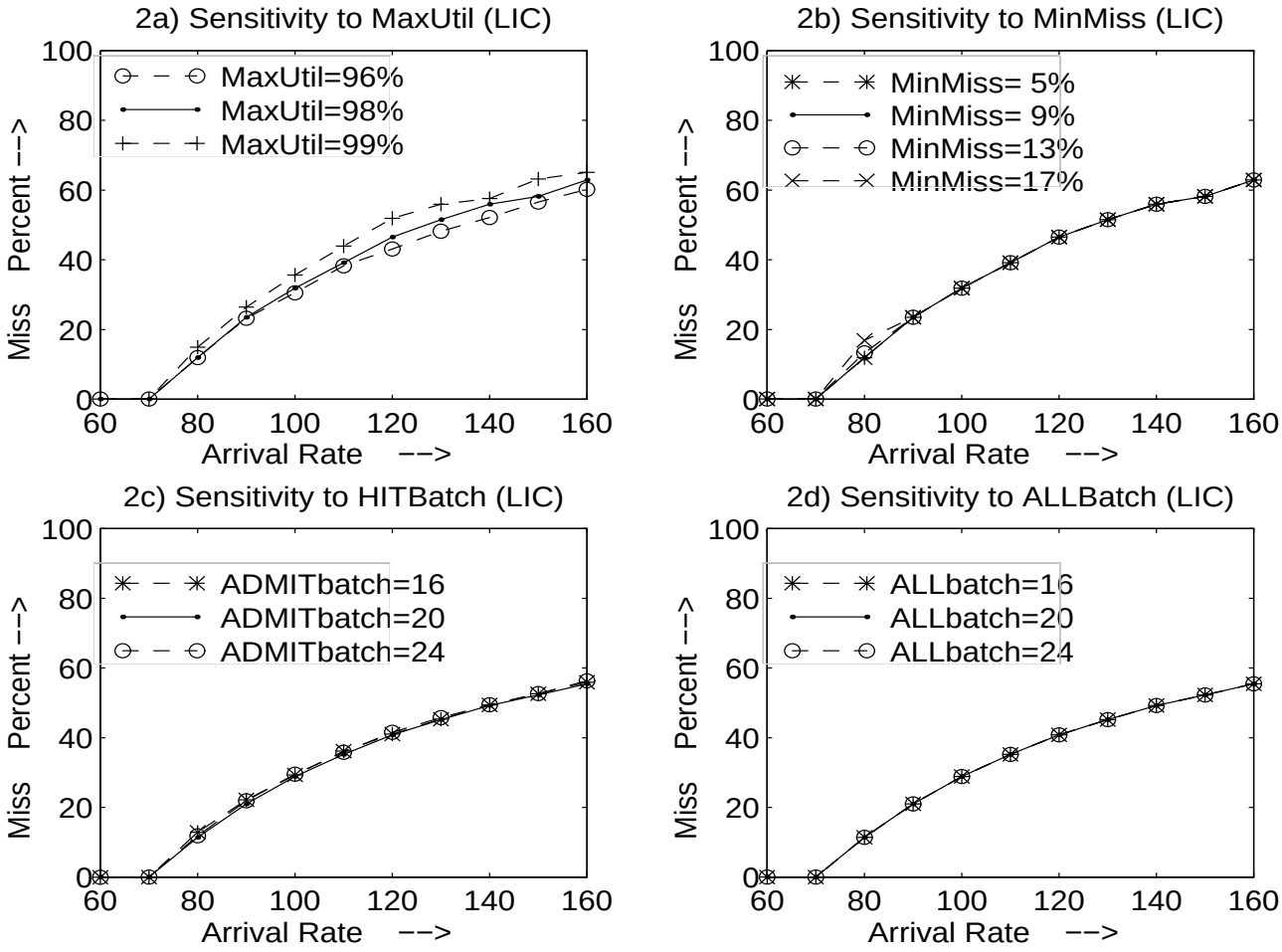
Moving on to the fairness aspects of performance, we present the *SizeFairness* and *TypeFairness* metrics in Figures 1g-i and 1j-l, respectively. In these figures we observe that AED-link does not purchase its good MissPercent performance by discriminating against one or the other class of transactions but, instead, is almost ideally fair with respect to both size and type. Note that LAB-link also provides fairness levels that are almost as good as that of AED-link. In contrast, the classical protocols are noticeably unfair with regard to both metrics. For example, B-link and OPT progressively favor smaller-sized transactions with increasing loading levels (Figure 1g). Similarly, SIX shows extreme bias favoring readers (Figure 1j) – the reason for this is that the SIX protocols allow readers to overtake updaters during tree traversals [16]. The *SizeFairness* and *TypeFairness* graphs for the MIC and the HIC experiment (Figures 1h, 1k and 1i, 1l) too show the same behaviour.

In summary, this experiment shows that AED-link is capable of simultaneously providing both the lowest miss percentage *and* close to ideal fairness.

## 6.4 Experiment 2: Load Control Sensitivity

The LAB-link protocol uses two algorithmic parameters *MaxUtil* and *MinMiss*, while the AED-link protocol uses *ADMITbatch* and *ALLbatch*. To evaluate the sensitivity of these protocols to their algorithmic parameters, we varied *MaxUtil* from 96% to 99%, *MinMiss* from 7% to 11% for LAB-link and varied *ADMITbatch* and *ALLbatch* from 16 to 24 for AED-link. (As mentioned earlier in Section 6.2, the recommended settings for these parameters are 98 percent and 20, respectively – these were used in Experiment 1.)

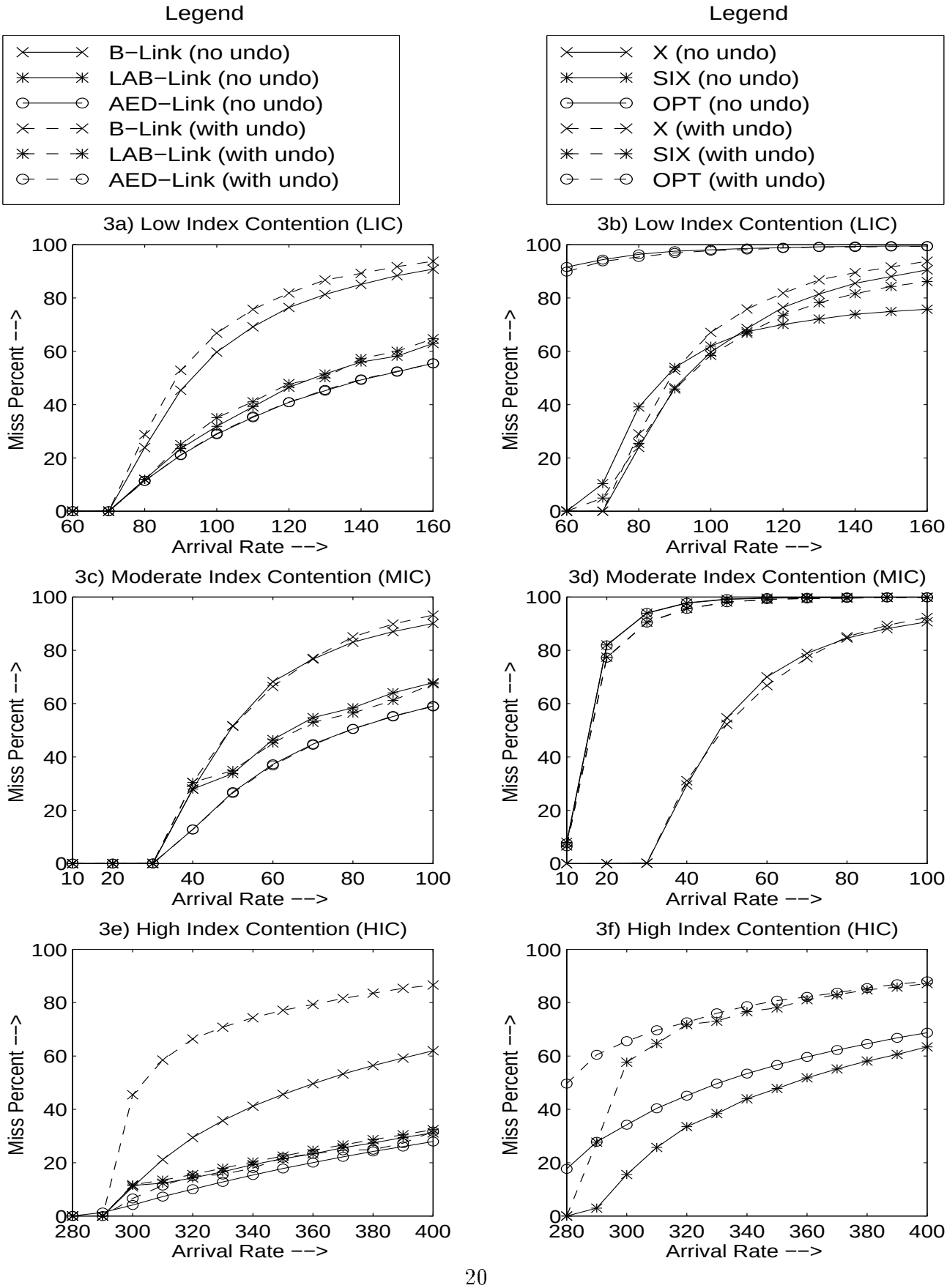
The results of this experiment are shown in Figures 2a,2c and 2b,2d for LAB-link and AED-link, respectively, in the LIC environment. These figures show that LAB-link is quite sensitive to the value of *MaxUtil* – for example, at an arrival rate of 120 transactions per second, there is a difference of nearly 5% in the absolute miss percentage between LAB-link performance with *MaxUtil* values of 96% and 99%. AED-link, on the other hand, behaves almost indistinguishably for all the *ADMITbatch* range of values that were considered. This kind of robustness of the AED policy to the choice made for the *ADMITbatch* value was also observed in [6]. Also, AED-link is stable to changes in *ALLbatch*. There is a noticeable difference in the results for different values of *MinMiss* for LAB-link at an arrival rate of 80 since this is the where the MissPercent is between two MinMiss parameter settings. There is no difference at other arrival rates since either the MissPercent is nearly 0% (for arrival rates 60 and 70) or the MissPercent is too high (for arrival rates ¿ 80) compared to the MinMiss parameter setting for it to come into effect. In summary, this experiment indicates that the performance of AED-link is relatively stable as compared to that of LAB-link.




## 6.5 Experiment 3: Effect of UNDOs

We now move on to considering the performance impact of modeling the *undos* of the index actions of aborted transactions. In [5], it was surmised that there is a direct positive feedback between the undo overhead and the miss percentage (based on the logic that the poorly performing protocols would have *more* cleanup work to do than the better protocols and therefore miss even more deadlines). Due to this relationship, it was felt that incorporating undo actions would correspondingly increase, but not qualitatively alter, the observed performance differences between the various protocols. However, as we will now show, the impact of undos is not so simply characterized and that unexpected effects can show up in certain situations.

With the inclusion of undos, Figures 3a-b, 3c-d and 3e-f present the results for the LIC, MIC and HIC environments, respectively. (For graph clarity, we separately show the performance of the B-link-based protocols and the other protocols.) To aid in understanding the effects of the undos, we have also included the corresponding results for the "no-undo" model. We first see in these figures that the effect of undos is significant only in the HIC environment (Figures 3e-f) but in the LIC and MIC environments (Figures 3a-b, 3c-d), the undos at most have only a *marginal* adverse impact on performance. Note that this marginal impact phenomenon is *not limited to*


Legend
B−Link (no undo)
LAB−Link (no undo)
AED−Link (no undo)
B−Link (with undo)
LAB−Link (with undo)
AED−Link (with undo)
Legend
X (no undo)
SIX (no undo)
OPT (no undo)
X (with undo)
SIX (with undo)
OPT (with undo)
3a) Low Index Contention (LIC)
3b) Low Index Contention (LIC)
3c) Moderate Index Contention (MIC)
3d) Moderate Index Contention (MIC)
3e) High Index Contention (HIC)
3f) High Index Contention (HIC)
Miss Percent −−>
Arrival Rate −−>

*light loads* – it also occurs for high miss percent values where we would expect the undo overheads to be considerable due to the large number of aborted transactions.

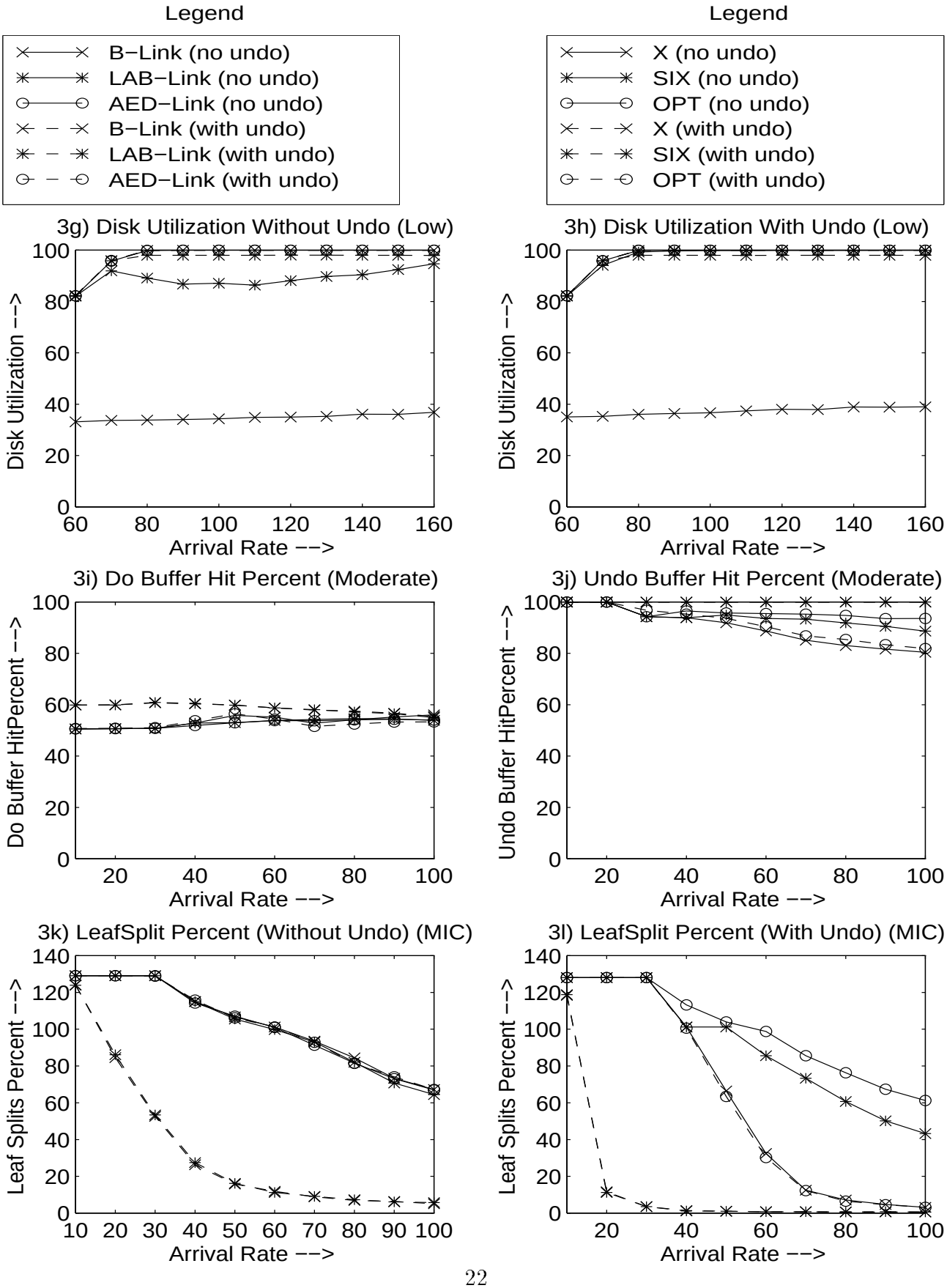
Even more interestingly, we see in the LIC and MIC environments that the protocols often *perform better with undos than in the absence of undos*, especially in the lower range of the loading spectrum! While the improvement is typically small, for SIX we observe a *significant* improvement in the LIC environment (Figure 3b). 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.

First, 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. This was quantitatively confirmed by measuring the buffer hit ratio of undo operations – it was typically about *90 percent*, as against a hit ratio of only around 50 percent for the normal operations (figures 3g-j). 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.

For the load-adaptive protocols, AED-link and LAB-link, an additional factor is at work: Their use of admission control and their completion of most of the transactions that they admit (especially in AED-link) means that the overhead arising out of transactions that are admitted and then miss their deadlines is very small. Moreover, *no* undo actions are necessary for transactions that are denied entry since they do not execute at all. In essence, the undo overhead is *inherently* small in the load-adaptive algorithms. This is confirmed in Figure 3e, where we see that even in the HIC environment, where undos were found to have a significant effect on the classical algorithms, the load-adaptive algorithms are hardly affected.

Second, 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. For example, in the MIC environment (figure 3k-l), where the workload consists of 100 percent inserts, the number of splits in the undo model was much fewer as compared to that for the no-undo model since many of the inserts were undone. 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.

Third, 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. (The only exception is the case when an updater has either already obtained an X latch on a node or has requested an X latch on the node and is of higher priority than the reader. Since the number of page splits and merges is comparatively small in our experiments, this exception does not occur frequently.) 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


Legend
B−Link (no undo)
LAB−Link (no undo)
AED−Link (no undo)
B−Link (with undo)
LAB−Link (with undo)
AED−Link (with undo)
Legend
X (no undo)
SIX (no undo)
OPT (no undo)
X (with undo)
SIX (with undo)
OPT (with undo)
3g) Disk Utilization Without Undo (Low)
Disk Utilization −−>
Arrival Rate −−>
3h) Disk Utilization With Undo (Low)
Disk Utilization −−>
Arrival Rate −−>
3i) Do Buffer Hit Percent (Moderate)
Do Buffer HitPercent −−>
Arrival Rate −−>
3j) Undo Buffer Hit Percent (Moderate)
Undo Buffer HitPercent −−>
Arrival Rate −−>
3k) LeafSplit Percent (Without Undo) (MIC)
Leaf Splits Percent −−>
Arrival Rate −−>
3l) LeafSplit Percent (With Undo) (MIC)
Leaf Splits Percent −−>
Arrival Rate −−>

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 in their undo processing they, by virtue of their "golden" priority, preempt 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 preempted leaf-level-latch-holder-updater will not request the root latch until its disk operation is completed.)

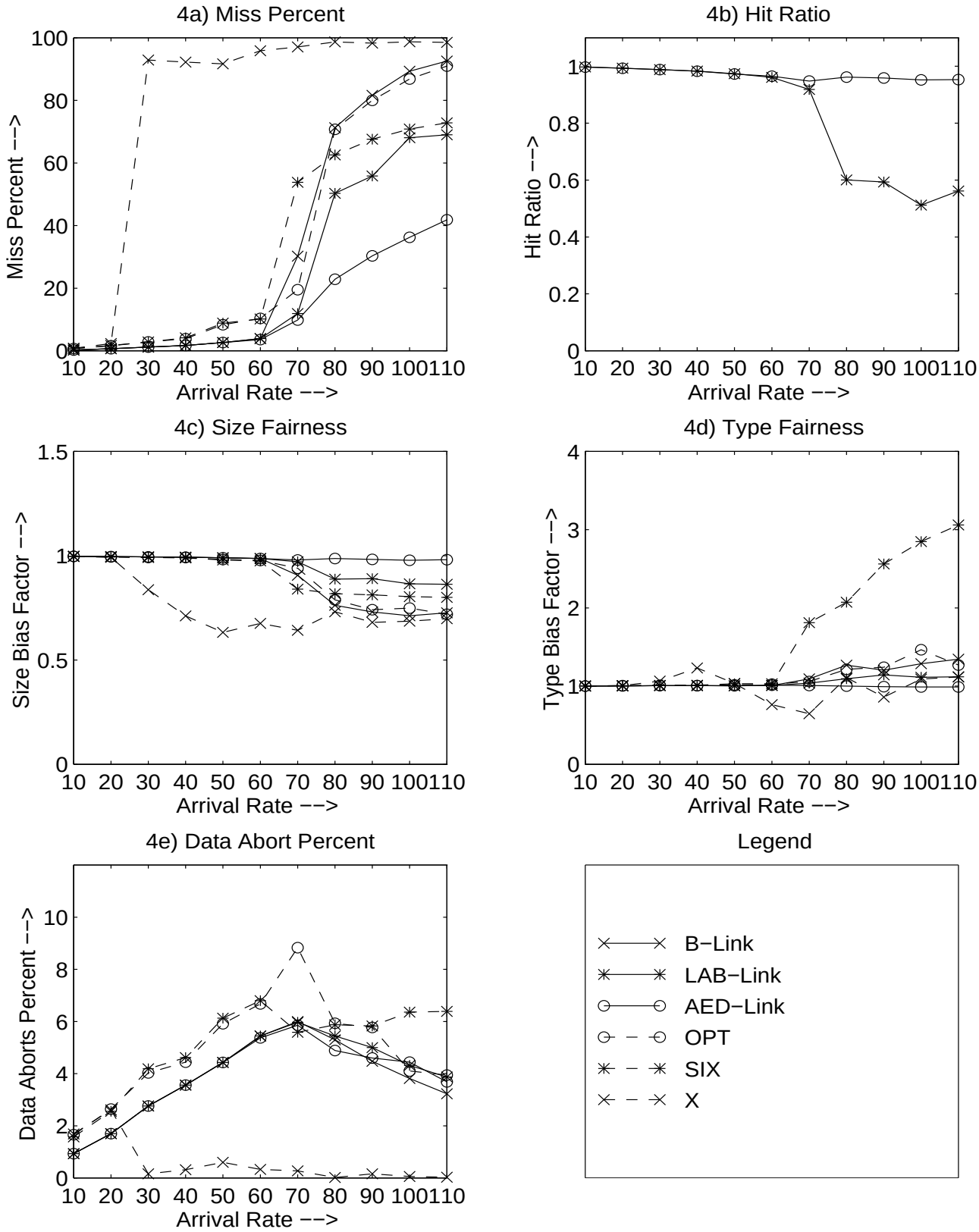
The benefit of convoy elimination is felt only under light and moderate loads but not under heavy loads. This is because, under heavy loads, the disk is almost fully utilized and becomes the primary bottleneck. In this situation, removing the latching bottleneck proves ineffective and is actually harmful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

## 6.6 Experiment 4: Range Queries

We now move on to considering the impact of *range* queries on the performance of the ICC protocols. For this experiment, we used an operation mix similar to that of the LIC experiment, which was composed of 80 percent searches, 10 percent inserts and 10 percent deletes. The only difference here is that each search now is a *range search* which retrieves *RangeKeys* key values, set equal to 10 for this ejperiment. The SlackFactor value was set to 2.0 for this experiment (The range search index action is estimated to take a time equal to *RangeKeys* disk I/Os although the disk access is not explicitly modeled).

For this environment, the MissPercent performance of all the ICC protocols is shown in Figure 4a. In this figure, we observe that the performance differences between B-link, LAB-link and AED-link, are *significantly* larger than those seen in the corresponding experiment with point searches. The reason is that range searches significantly increase the level of *data contention* in the system since they hold locks on several key values simultaneously (this can be seen clearly in Figure 4e, which shows the percentage of transactions that get restarted due to key-value contention. The figure shows a relatively high value of data contention in the system as compared to the case where there was no range queries i.e. in the Low Index Contention experiment.) In this environment, LAB-link by virtue of being cognizant of only resource contention fails to apply sufficiently strict admission control. This is brought out clearly in Figure 4b, which shows the hit ratios in the ADMIT group for both the LAB-link and AED-link protocols. We see here that the hit ratio of LAB-link goes almost as low as 0.5, thereby resulting in many more missed deadlines. In contrast, AED-link, due to its output-based feedback retains a close to 1.0 hit ratio in the ADMIT group as desired. Also note that SIX is still able to maintain the MissPercent value of nearly 75%. Again this is due to SIX giving preference to readers. This is clearly seen in the TypeFairness graph in Figure 4d.

In summary, this experiment is qualitatively very much similar to the Low Contention Exper-


4a) Miss Percent
Miss Percent −−>
Arrival Rate −−>
4b) Hit Ratio
Hit Ratio −−>
Arrival Rate −−>
4c) Size Fairness
Size Bias Factor −−>
Arrival Rate −−>
4d) Type Fairness
Type Bias Factor −−>
Arrival Rate −−>
4e) Data Abort Percent
Data Aborts Percent −−>
Arrival Rate −−>
Legend
B−Link
LAB−Link
AED−Link
OPT
SIX
X

iment except that range searches increase the data contention in the system, thereby glaringly showing the negative aspects of the "only-Resource-Utilization-based" approach of LAB-link. AED-link is able to perform well in this experiment too again due to its "end-user-performance-metric-based" feedback policy.

# 7 Conclusions

In this paper, we investigated the problem of index concurrency control for real-time database systems supporting transactions with firm deadlines. We extended and improved on the research presented in a previous study. In particular, a new load-control-based B-link ICC protocol called AED-link was presented, the fairness aspects of ICC protocols were examined in detail, the modeling of undos of index actions of aborted transactions was included in the experimental framework, and the performance effects of range queries were evaluated. The experiments were conducted using a detailed simulation model of an RTDBS and considered the entire range of index contention environments – low, moderate and high. The protocols in the evaluation suite included real-time variants of three different classes of index CC algorithms: Bayer-Schkolnick, Top-Down and B-link. To ensure transaction serializability, a simplified version of the Next-Key-Locking algorithm used in the ARIES system [12] was implemented.

Our experimental results showed that AED-link provides noticeably better MissPercent performance than LAB-link, the load-adaptive B-link algorithm proposed in [5]. This was primarily due to two reasons: First, AED-link is impervious to the source of contention in the system since its feedback is based on system output performance, unlike LAB-link which is limited to measuring resource utilization. Second, AED-link is based on the well-established real-time principle that the Earliest Deadline policy minimizes the miss percentage in a lightly loaded system. Apart from its good MissPercent performance, AED-link also provided close to *ideal fairness*, not discriminating either based on transaction size or on transaction type. In contrast, all the classical protocols showed bias in favor of smaller-sized transactions and in favor of read-only transactions with increasing loading levels. Finally, we showed that unlike LAB-link whose performance is quite sensitive to the settings for its algorithmic parameters, AED-link is relatively *robust* to its parameter settings.

Incorporating undos of index actions of aborted transactions was expected, in [5], to significantly increase, but not qualitatively alter, the performance differences between the protocols. However, our results indicate that the adverse performance impact of making undos is only felt when the CPU is a bottleneck, and that too only for the classical protocols. For the load-adaptive-protocols, due to their admission control policy, undos have virtually no effect under any contention level. In addition, undos can have an unanticipated beneficial impact of (a) reducing the number of index node splits and merges, and (b) for the SIX protocols, preventing extended formation of updater convoys.

Finally, our experiments involving range queries showed AED-link to perform far better than both B-link and LAB-link. This is because unlike LAB-link which was blind to the steep increase in data contention created by the range queries, AED-link was able to successfully limit its transaction admission to a sustainable number and thereby minimize the miss percentage.

In summary, we suggest that RTDBS designers may find AED-link to be a good choice for

real-time index concurrency control.

**Acknowledgements**

This work was supported in part by a research grant from the Dept. of Science and Technology, Govt. of India.

# References

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

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

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

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

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

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

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

[8] E. Jensen, C. Locke and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", *Proc. of 6th IEEE Real-Time Systems Symposium*, December 1985.

[9] 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.

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

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

[12] 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 1990.

[13] 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.

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

[15] Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences*, 33(2), 1986.

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