# Index Concurrency Control
# in
# Firm Real-Time DataBases

A Project Report

Submitted in partial fulfilment of the

requirements for the Degree of

**Master of Engineering**

in

Faculty of Engineering

by

**Narayanan S. R.**

Department of Computer Science and Automation

Indian Institute of Science

Bangalore – 560 012 (INDIA)

January 1997

# Contents

# Acknowledgements

I would like to thank my project guide, *Dr. Jayant Haritsa*, for his constant support and encouragement throughout the course of the project. The long discussions I had with him, the untiring enthusiasm he showed in it and his insightful comments were a rewarding experience. The Database Systems Lab, with its great facilities and a wonderful working environment has contributed a lot to the final shape of this work.

I would like to thank my lab-mates shri, nalin, mouli, vidya, ramesh and binto for their help and enjoyable company. Their presence in the lab created a co-operative environment.

I take this opportunity to thank all the faculty I have learnt under, for the stimulating intellectual environment they provided.

I also wish to thank all my class-mates for their warm friendship. Particularly, the trip to Nandi Hills is an unforgettable experience.

Special thanks go to my friends abala, arvs, ganesh, jaianand, mahesh, ms, paththu, rajesh, rajkumar, ramesh, srini, subbu, sv, varadhan, venu and viswa for making my stay in IISc a memorable one.

Finally, I would like to record my gratitude to my parents and my brother, for their affection and concern that made me what I am.

# Abstract

A Real Time Database System (RTDBS) is a DBMS which in addition to supporting data integrity constraints has to support timing constraints in the form of transaction deadlines. A Firm RTDBS is one where transactions are immediately aborted and discarded from the system on missing their deadline. The objective in a firm RTDBS is to minimize the number of transactions missing their deadlines as opposed to minimizing the average response times of transactions in a conventional DBMS. Due to the difference in objectives, the scheduling policies in an RTDBS differ from those of a conventional database.

In this project, we study the issue of Index Concurrency Control (ICC) in a firm RTDBS. We start off from the work done in an earlier study [GHSS95]. The earlier study had proposed a load-adaptive real-time variant of an ICC algorithm (B-link) called LAB-link which was found to provide the best real-time performance in that study. 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. We also make a detailed study of the fairness charactersitics of ICC protocols. Our results indicate that AED-link provides almost ideal fairness for a wide range of workloads. We also explicitly model the "undos" of index actions of aborted transactions. An unexpected and interesting result here is that performing undos usually has only a marginal adverse impact on real-time performance and can, in a few scenarios, actually result in better performance. We also modeled the performance effects of range index operations and experimented with different real-time situations.

# Chapter 1

# Introduction

## 1.1 Real Time DataBase System

A *Real Time DataBase System* (RTDBS) is a transaction processing system which is designed for workloads where transactions have completion deadlines. It is similar to a conventional (non-real-time) database system in that it is also a transaction processing system. On the other hand, in an RTDBS, transactions have deadlines (time constraints). Deadlines may be of two kinds

- Release deadline
- Finish deadline

*Release deadline* corresponds to the time before which the transaction is not supposed to *start*. An example of an application involving deadlines of this kind is a transaction processing system where transactions have dependency relationships of the form $\mathcal{A} \rightarrow \mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are transactions, and $\mathcal{B}$ is not supposed to start before $\mathcal{A}$ finishes. *Finish deadline* corresponds to the time before which the transaction is supposed to complete (commit). One can also consider such applications to have their release deadline equal to their arrival time. Although an application can have both release and finish deadlines, we consider systems involving only finish deadlines.

Although an RTDBS has to support time constraints, it differs from a *Real Time System* (RTS) in that RTSs typically deal with simple transactions (called processes) which have predictable data or resource requirements. This makes the problem of scheduling transactions much easier in an RTS as compared to an RTDBS.

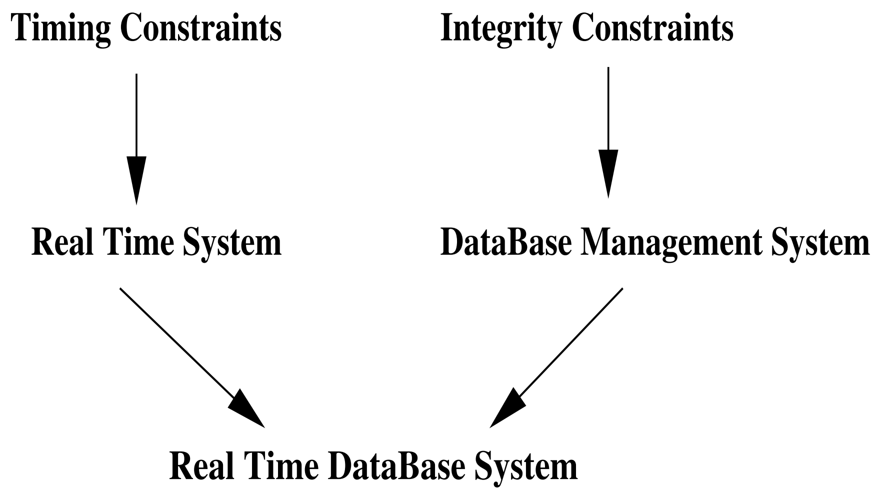



Figure 1.1: Real Time DataBase System

As shown in Figure 1.1, an RTDBS has to support features of both conventional database systems as well as real time systems. It has to maintain *data integrity constraints*, which is usually accomplished by the notion of *serializability* where a concurrency control protocol is used to schedule the data access of concurrent transactions so as to produce a schedule equivalent to a serial schedule (a schedule obtained if transactions were executed one after another in some order). It also has to meet *timing constraints*. The important point is not how much earlier a transaction finishes before its deadline, but whether or not it finishes before its deadline. Transactions which miss their deadlines are referred to as *late* transactions. RTDBSs can be classified into three kinds based on their response to a transaction-time constraint violation (the following examples of RTDBS applications are from [AG92]).

- *Hard*

  In this case, missing the deadline of a single transaction is equivalent to a catastrophe. e.g. Threat analysis using a radar system, to track objects, with a computer to perform image processing and control. The data collection and signature look-up must be done in real time.

- *Firm*

  Here, a transaction missing its deadline is aborted and discarded immediately from the system. e.g. A stock market where transactions have to be completed before the stock values change.

- *Soft*

  A transaction missing its deadline is not discarded from the system. In such an application, lesser *utility* is added for completing transactions after their deadlines. Utility of a completing transaction is 1.0 (normalized) while for a late transaction the utility reduces as its finish time is delayed. One issue to be considered in this case is the assignment of priority to late transactions. One possibility is, the transaction missing its deadline may be executed with lesser priority since it has already missed its deadline and other unfinished transactions have to be given priority over this. The other is, it may be executed with higher priority since it may clog up system resources which may affect incoming transactions. e.g. A banking transaction processing system. Here, the correct thing to do, for a transaction that misses its deadline, is to continue its execution with a higher priority. In this case, finishing within time is a desirable feature while finishing it is an essential feature.

## 1.2 Firm Deadline RTDBS

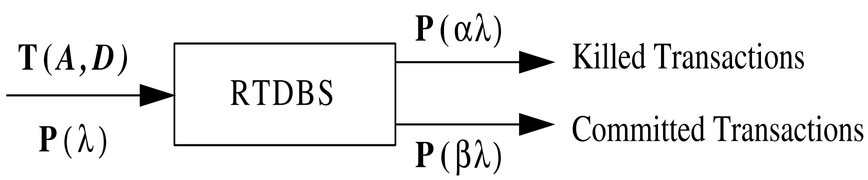
We focus our attention on firm RTDBS for the following reasons [HCL92].

1. Database systems for supporting hard deadline applications appear to be infeasible due to the large variance between the average case and worst case execution times of a typical database transaction [SZ88].

2. Understanding the issues involved in the firm deadline case can provide insights into understanding the more complex framework of soft deadline applications, where

transactions may retain some utility if completed after their deadlines (firm deadline case is a particular case of the soft deadline case with utility of a late transaction being zero however soon it finishes after its deadline).

As mentioned earlier, a firm deadline RTDBS aborts and discards immediately from the system transactions that have missed their deadlines. Transactions may abort also due to the underlying concurrency control algorithm, data integrity violation, etc. These aborts involve a restart of the transaction. To differentiate such aborts from an RTDBS abort, where the transaction is thrown out of the system immediately, we refer to the RTDBS abort as a *kill*. In the conventional database, the goal was to minimize the average response times of transactions. On the other hand, the goal here is to minimize the number of transactions that miss their deadlines. One may also consider it this way, the goal is to maximize the total utility of the transactions exiting from the system. Because of this difference in objectives, the results obtained in the conventional domain may not carry over to the real-time domain (eg. In transaction concurrency control, optimistic algorithms which perform worse than locking algorithms in conventional DBMS [ACL87], outperform locking algorithms in the firm real-time domain [HCL92]).




$\alpha + \beta = 1$

$\lambda$ = Transaction Arrival Rate

$\mathbf{P}(\lambda)$ = Arrival Process with parameter $\lambda$

$100\alpha$ = Miss Percent

Figure 1.2: Firm RTDBS Model

The model of the firm deadline database used is shown in Figure 1.2. Here transactions arrive as a random process with mean arrival rate $\lambda$. Each transaction has an arrival time $A$ and a deadline $D$. The RTDBS has knowledge of only the transaction's deadline. The output of the RTDBS consists of two streams of transactions, ones which miss their deadlines and ones which finish before their deadline. The stream consisting of killed transactions has a rate $\alpha\lambda$ where ($\alpha$ varies from 0.0 to 1.0), while the other stream consisting of committed transactions has a rate $(1 - \alpha)\lambda$. The value $\alpha$ expressed as a percentage is called the *Miss Percent*. The objective of the firm RTDBS is to minimize $\alpha$.

## 1.3 Minimizing Miss Percent

The performance of the RTDBS is a function of several parameters including the database system architecture, underlying hardware speed, etc. While the performance of the real-time database may definitely be improved by additional hardware functionality in the system, the interesting question is to find out how to improve the performance, given a system configuration. One way of improving performance is to consider the policies used for scheduling transactions at resources. These policies affect *when* the transaction gets service and hence directly impact the completion of the transaction before its deadline.

By resources, we mean both physical and logical resources [HCL92]. Physical resources of a system consist of the set of hardware available i.e. processors, disks, memory, etc. These resources are fixed for a given system configuration. Logical resources are the data objects and metadata (data about data) comprising the database. Transactions contend at both types of resources. We henceforth refer to the contention at the physical resources of the system, data and index as *resource contention, data contention and index contention* respectively.

Much of the research on real-time database systems had focused on the various database system policies like priority assignment, data concurrency control, memory management, etc. (e.g. [AG92, HCL92, PCL94, HJC93]). However, the design and evaluation of concurrency control algorithms for *indexes*, which are an integral part of database systems since they are used for efficient access of data, has been considered for the real-time domain till now only in [GHSS95].

## 1.4 Index Concurrency Control

In any database, for efficient processing of queries, indexes are used. Thus almost all transactions have to contend for index access. This implies that the index might become a bottleneck even at light workloads. Concurrency control of the index thus becomes important in a conventional database itself. In a real-time database, transactions have to meet their deadlines. Hence it is all the more likely that the system will heavily rely on indexes to meet transaction timing constraints. At this point, a poor index concurrency control algorithm may lead to more transactions missing their deadlines. Thus the problem of Index Concurrency Control (ICC) becomes even more important in an RTDBS scenario.

Although a variety of access methods have been proposed in the literature, commercial database systems typically implement indexes using a *B-tree* [Com79]. In particular, they use a minor variant of the B-tree called the $B^+$-tree where all the data values are stored in the leaves of the tree. We use the term B-tree to refer to this variant. The B-tree is a multiway, height-balanced search tree. Data is accessed by traversing down from the root of the tree, following the pointers present in the intermediate nodes of the tree, to the leaf where the data value is stored. In 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, many concurrency control algorithms were proposed for the B-tree (e.g. [BS77, Bil86, KW82, LS86, LS92, LY81, ML89, Moha90, Sag85, SG88]).

A comparative study of the above algorithms was done in [JS90, SC91]. However, this was done for the conventional database domain. [JS90] compared the performance of B-tree concurrency control algorithms using an analytic model of an open queuing system, while [SC91] did the same using a simulation model. The results of both the studies showed that *B-link* [LY81] algorithms (one class of index concurrency control algorithms) performed the best over a wide range of workloads and system operating conditions.

Recently, a study of a representative set of the above algorithms was done for the real-time domain in [GHSS95]. The findings of this study showed that some results which were considered true in the conventional case (conventional database case) didn't hold true in the real-time case (real-time database case). An example is the case where B-link algorithms performed badly than others under heavy load. Analysis of the experiments showed a surprising result in that the very feature of the B-link algorithms i.e. its high resource utilization capacity, that enabled it to perform the best in the conventional case, contributed to its performing badly in the real-time case.

This study also proposed a real-time variant of the B-link algorithm called LAB-link (Load Adaptive B-link) which enforces an admission control policy based on system resource utilization. This was found to perform better than the other algorithms in all cases. The admission control policy used in LAB-link is orthogonal to the ICC algorithm used. By this we mean that the admission control policy can be used for any other ICC algorithm too. We refer to this admission control policy as Load Adaptive Policy (LAP). The limitations of this study are

1. The proposed LAB-link algorithm is not a robust solution in that it is dependent on the system resource utilization parameters rather than on end-user performance metrics.

2. Undoing the index actions of an aborted transaction, required for maintaining database consistency, was not modeled.

3. The only performance metric that was considered was Miss Percent. The *fairness* aspects of the protocols, in terms of bias for or against a particular class of transaction, were not considered quantitatively.

4. 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.

## 1.5 Contributions

This project attempts to overcome the limitations of the above study and also deal with some issues which were not considered by the same. We present a new real-time ICC protocol called **AED-link** which employs an adaptive feedback-based admission control mechanism termed the *Adaptive Earliest Deadline* (AED) algorithm [HLC91]. This algorithm performed load control via priority assignment thus acting as an alternative to LAB-link, though being more rational in the sense that it is only dependent on the output process of the RTDBS for its feedback and not on any system dependent parameter like resource utilization as was the case in LAB-link. Also, as earlier mentioned, LAB-link was found to be sensitive to the resource utilization parameter. Such sensitivity was not noticed in the AED-link algorithm.

The earlier study, though didn't model undos, had 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 significantly increase, but not qualitatively alter, the observed performance differences between the various protocols. However, as we will show, the impact of undos is not so simply characterized and that unexpected effects can show up in certain situations.

In addition to the important performance metric of miss percent, we also compared the ICC algorithms based on two other metrics, one termed *SizeFairness*, which basically gives a measure of the bias that the ICC algorithm shows towards transactions of different sizes and another termed *TypeFairness*, which gives a measure of the bias that the ICC algorithm shows towards transactions of different types (e.g. readers, updaters). The size of a transaction refers to the number of index actions constituting the transaction while the type of a transaction refers to the number of update index actions in the transaction.

We found AED-link to perform better than the other algorithms in terms of these performance metrics too. Since AED is a load control policy orthogonal to the ICC algorithm used, we applied it to the other ICC algorithms also and found it to improve their performance as well. We also experimented with different deadline formulae in order to model different real-time situations and obtained similar results as those obtained in the baseline case.

We also experimented the impact of *range* queries on the performance of the ICC protocols. In this experiment too, we found AED-link to provide the best real-time performance.

## 1.6 Other RTDBS Related Work

In this section, we highlight RTDBS research related to other scheduling policies such as memory management, priority assignment, transaction concurrency control, etc.

- *Transaction Concurrency Control*

  By transaction concurrency control, we mean data concurrency control. Data concurrency control in a conventional database was achieved by using locking protocols (pessimistic), validation protocols (optimistic) or timestamping protocols. The pessimistic real-time protocols developed were,

  - 2PL-HP [AG88]
  - 2PL-WAIT [SRL87]

  Their optimistic counterparts [HCL92] were,

  - OPT-SACRIFICE
  - OPT-WAIT
  - WAIT-50

  Also speculative protocols which were a mix of the pessimistic and optimistic approaches were developed [BB94]. The results obtained in [HCL92] were that optimistic methods outperform their locking counterparts at high data contention workloads and vice versa at low data contention.

- *Priority Assignment*

  [AG92] studied four methods of priority assignment,

  1. First Come First Serve

2. Earliest Deadline
3. Least Slack with Static Evaluation
4. Least Slack with Continuous Evaluation

Earliest Deadline was found to perform better in lightly loaded situations, while it performed badly in overloaded situations. [HJC93] proposed a new dynamic priority assignment policy called the *CCA* (Cost Conscious Approach) and found it to perform better than Earliest Deadline.

- *Memory Management*

  [PCL94] introduced a *Priority Memory Management* (PMM) algorithm which attempts to minimize the miss percent by adapting both its multiprogramming level and its memory allocation strategy to the characteristics of the offered workload.

## 1.7 Organization of the Report

The rest of the report is organized as follows. Chapter 2 presents a survey of various B-tree concurrency control algorithms. Chapter 3 presents the real-time adaptation that needs to be done to the existing ICC algorithms. Chapter 4 presents our contributions. Chapter 5 discusses the simulation model of our RTDBS. Chapter 6 deals with the experiments and the results that we obtained. We conclude in Chapter 7 with a summary of the work done and a discussion of the scope for future research.

# Chapter 2

# B-Tree Concurrency Control Algorithms

Queries are the heart of any database management system. The aim of the DBMS is to process the data items involved in the query efficiently. The value that a data item takes is referred to as the data value. Since huge amounts of data items are stored in any non-trivial database, efficient processing of the data items is possible only by carefully organizing them. A structure that does this organization is called an *Index*.

The data items can be organized based on the

- *Embedding Space*

  The embedding space is the range of values that each tuple in the relation can take. This space is partitioned into many disjoint, mutually-exclusive intervals and data items in the database whose data values occur in one interval are grouped together. However, the inherent problem with this approach is that, if the data value distribution in the database is skewed, some intervals may not have a data item at all. Thus, this may result in no organization at all for extremely skewed data.

- *Data Values*

  Here the organization is done with respect to the data values currently existing in the database. Hence this approach doesn't suffer from the skewness drawback.

Typical commercial database systems implement indexes using a B-tree. A B-tree is an index that uses the latter approach. Here the index stores and retrieves information associated with a search key. The index can be one-to-one or one-to-many, i.e. there may be only one or many tuples with the given key value in the database.

## 2.1 B-tree review

The material for this section is taken from [Sri92]. The B-tree is a multiway, height-balanced search tree. It is page-oriented in the sense that each node of the tree is stored in one page. All the key values are stored only in leaf-nodes. A B-tree is said to be of order $d$ if every node of the tree has at most $2d$ separators and every node except for the root has at least $d$ separators. A separator defines one step in a search path to leaf pages that contain actual keys and their associated information. The root has at least two children. The leaf is at the lowest level (level 1) and the root is at the highest. The height of the tree is the number of levels in the tree. A non-leaf node with $k$ separators contains $k + 1$ pointers to children index nodes. Within a node, the separators are stored in ascending order of their values. The pointer to the left of a separator is the subtree containing all key values less than the value of the separator and greater than the value of the previous separator at the same level. Thus at every level, the separators divide the data values into many intervals, the size of each interval decreasing as the level decreases till it becomes one at the leaf level. Figure 2.1 illustrates a B-tree fragment.

B-trees provide efficient access because of the height-balance they maintain. This is what makes them do searches, inserts, deletes all in time which is of order logarithmic in the number of data values. Since the separators partition the data value range into intervals, a search for a key value proceeds from the root index page and follows the pointer that corresponds to the range containing the searched key value till it reaches the leaf. The searched key value may or may not be present in the leaf. Inserts and deletes are done in a similar fashion except that when an insert occurs in a full leaf page or a delete occurs in a leaf page with $d$ entries, a page split or page merge occurs. A B-tree page split is illustrated in Figure 2.2.

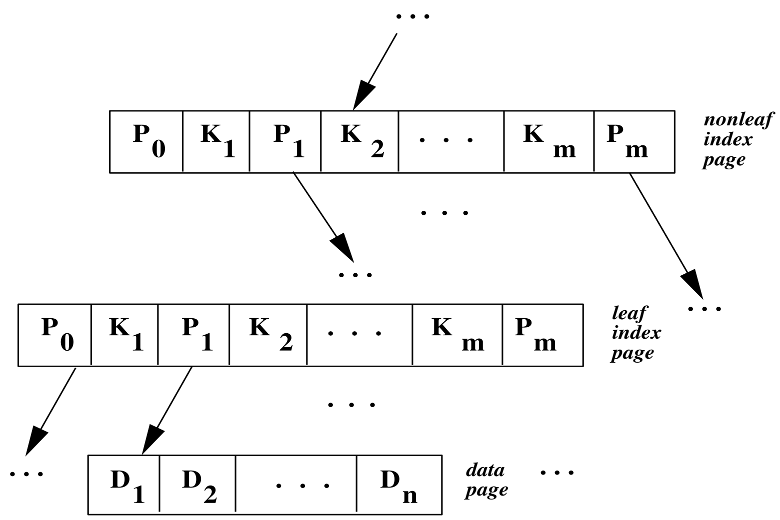



Figure 2.1: A B-Tree Fragment

Page merges, in practice, are done only when pages become empty, since this simplifies implementation and also is found, to not decrease occupancy by much, for practical workloads [JS89]. We employ this strategy for B-tree merges in our study. A node is said to be *safe* for an insert if it is not full and safe for a delete if it has more than one entry. Page splits or merges proceed from the leaf to the lowest level unsafe node on the path to the root. If all the nodes in the path from the node to the root are unsafe, then the tree increases or decreases in height. The set of unsafe nodes that are modified in an update operation is called the *scope* of that update.

## 2.2 Index Node Locks

Transactions access the index concurrently. B-tree concurrency control (CC) algorithms typically ensure index consistency in the face of concurrent access by maintaining index node locks. Actually, in many commercial DBMSs, index node locks are implemented using *latches* which are equivalent to fast locks. A lock is required to maintain *logical* consistency while a latch is required to maintain *physical* consistency of index nodes. Also a lock is held for a long time (usually till the commit time of the transaction), while a latch (lock on an index node) is held usually for a short duration. This optimization is

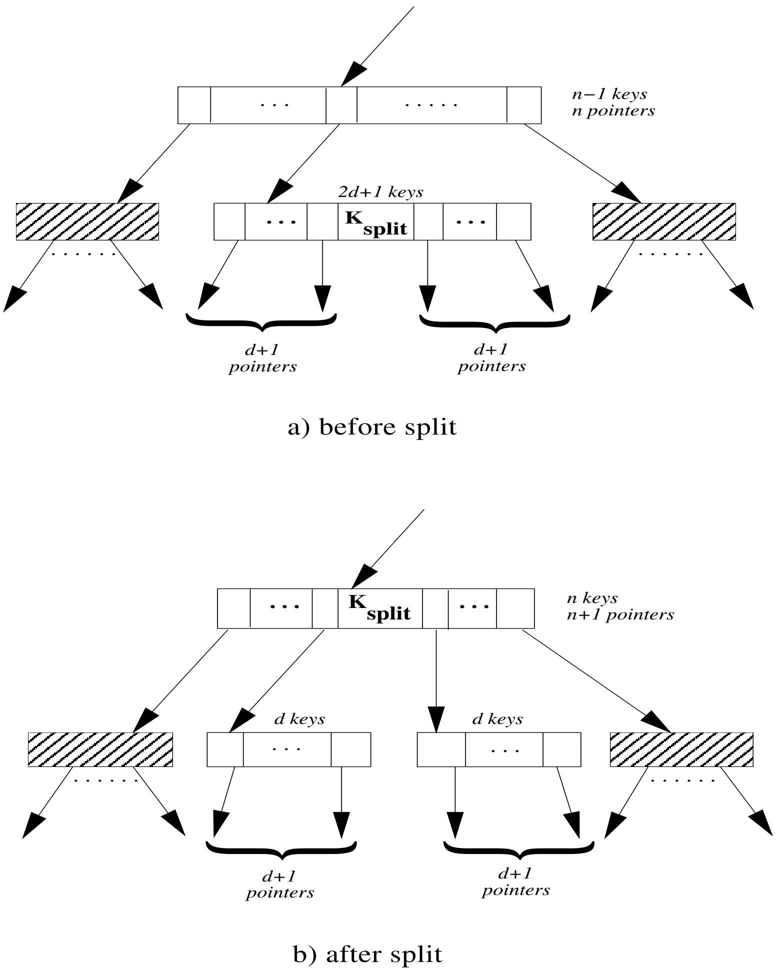



Figure 2.2: A B-Tree Page Split

also taken into account in our study. 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 if it requests a lock on an index page while already holding a lock on the page's parent, releasing the parent lock if the new page is found to be safe. The index lock modes used by the B-tree CC algorithms considered in this study and their compatibility relationships are given in Figure 2.3. In this table, IS refers to "intention shared", IX refers to "intention exclusive", SIX refers to "share and intention exclusive" and X refers to "exclusive" locks [Gray79].

There are three well-known classes of B-tree concurrency control algorithms (which we will discuss),

**Holding mode** →

**Request Mode** ↓

| mode | S | IX | SIX | X |
|---|---|---|---|---|
| S | ✓ | ✓ | ✓ | |
| IX | ✓ | ✓ | | |
| SIX | ✓ | | | |
| X | | | | |

Figure 2.3: Index Node Lock Compatibility Table

1. Bayer-Schkolnick Algorithms
2. Top-Down Algorithms
3. B-link Algorithms

Each class differs essentially in the granularity of operations into which the scope of an updater is split.

## 2.3 Bayer-Schkolnick Algorithms

This class of algorithms was proposed by Bayer and Schkolnick [BS77]. In this class, we consider three algorithms B-X, B-SIX and B-OPT. In all these algorithms, readers follow the same locking protocol. They first obtain an IS lock on the root index node. Then they lock-couple to the leaf using IS locks. The algorithms differ in the locking strategy used by updaters.

In B-X, updaters obtain an X lock on the root and then lock-couple to the leaf using X locks. The problem with this algorithm is that an updater may temporarily shut off a reader who is not even in the scope of that update. To overcome this problem, updaters can lock-couple using SIX locks to the leaf and can then convert the SIX locks in its scope to X locks bottom up. In this case, readers are preferred over updaters since IS locks of readers are compatible with SIX locks of an updater, while SIX locks of two updaters are not compatible. This causes queuing of updaters behind one updater, while the readers are allowed to pass through without any hindrance.

In both the above algorithms, updaters which do not conflict with each other in their scope may still interfere at higher levels of the tree. The third algorithm, called B-OPT tries to overcome this by having updaters descend with IX locks instead of SIX locks. Also here updaters don't lock-couple. They release IX locks of the parent after obtaining an IX lock on the child, regardless of the safety of the child, till they reach the leaf, where they obtain an X lock. If the leaf is found to be safe, then the update is done. Otherwise, the updater releases the X lock and descends again using the B-SIX algorithm. The performance of this algorithm depends on the number of updaters that undergo a second pass. It is usually expected to be less, since in a large B-tree, page splits or merges occur very rarely.

One problem with this class of algorithms is that updaters update the entire scope at the same time making it necessary to hold several X locks at the same time. The Top-down and B-link algorithms break the updating of the scope into several atomic operations.

## 2.4 Top-Down Algorithms

In this class of algorithms [MR85, LS86], readers behave in exactly the same fashion as in the Bayer-Schkolnick algorithms. Updaters, on the other hand, perform *preparatory* splits and merges during their index descent. An inserter, on finding an unsafe node during its descent, performs a preparatory page split and inserts an appropriate index entry into the parent of the newly split node. Similarly, a deleter, on finding an unsafe node, performs a preparatory merge with its adjacent node and deletes the appropriate entry from the parent. The parent is guaranteed to be safe since if it were unsafe, a preparatory split or merge would have been done on that itself during the descent.

Based on the preparatory operations described above, we consider three algorithms TD-X, TD-SIX and TD-OPT. In TD-X, updaters obtain an X lock on the root and then lock-couple using X locks. All preparatory operations are done before releasing the parent's X lock. This can be improved by using SIX locks instead of X locks for the descent, and converting the SIX locks to X locks only if a preparatory operation has to be done. This variation is called TD-SIX. In TD-OPT, updaters descend using IS locks in the first pass, and lock-couple using IS locks till they reach the leaf where they get an X lock. If the leaf is found to be unsafe, then they release all locks and restart the update operation using the TD-SIX protocol. One problem with this approach is that there is a possibility of deadlock between first pass updaters and second pass updaters [GHSS95]. A first pass updater which holds an IS lock on some parent node of a leaf and wants an X lock on the leaf node may deadlock with a second pass updater who has SIX locks on both the nodes and requests X locks on both of them for updation. To overcome this problem, we use IX locks instead of IS locks for updaters in the first pass.

The top-down algorithms split the update of the scope into sub-operations, that involve only nodes at two adjacent levels of the tree. The B-link algorithms go one step further and limit each sub-operation to nodes at a single tree-level. Also they differ from the top-down algorithms in that they do their updates in a bottom-up fashion.

## 2.5 B-link Algorithms

The B-tree we considered till now had sibling pointers at only the leaf level. A B-link tree [LY81, Sag85, LS86] is a modification of the basic B-tree in which every node in addition to the ordinary B-tree page, contains a high key (the highest key of the subtree rooted at this page) and a pointer to its right sibling at each level. The rightmost node at every level has its pointer field set to null. The link enables a page split to occur in two phases, a half-split followed by the insertion of an index entry into the appropriate parent. After the page split and before the index entry corresponding to the new page has been inserted into the parent page, the new page is reachable through the right link of the old page. This process is illustrated in Figure 2.4 ([Sri92]).

Operations arriving at a newly split node with a search key greater than the high key use the right link to reach the appropriate page. Such a sideways traversal is termed a

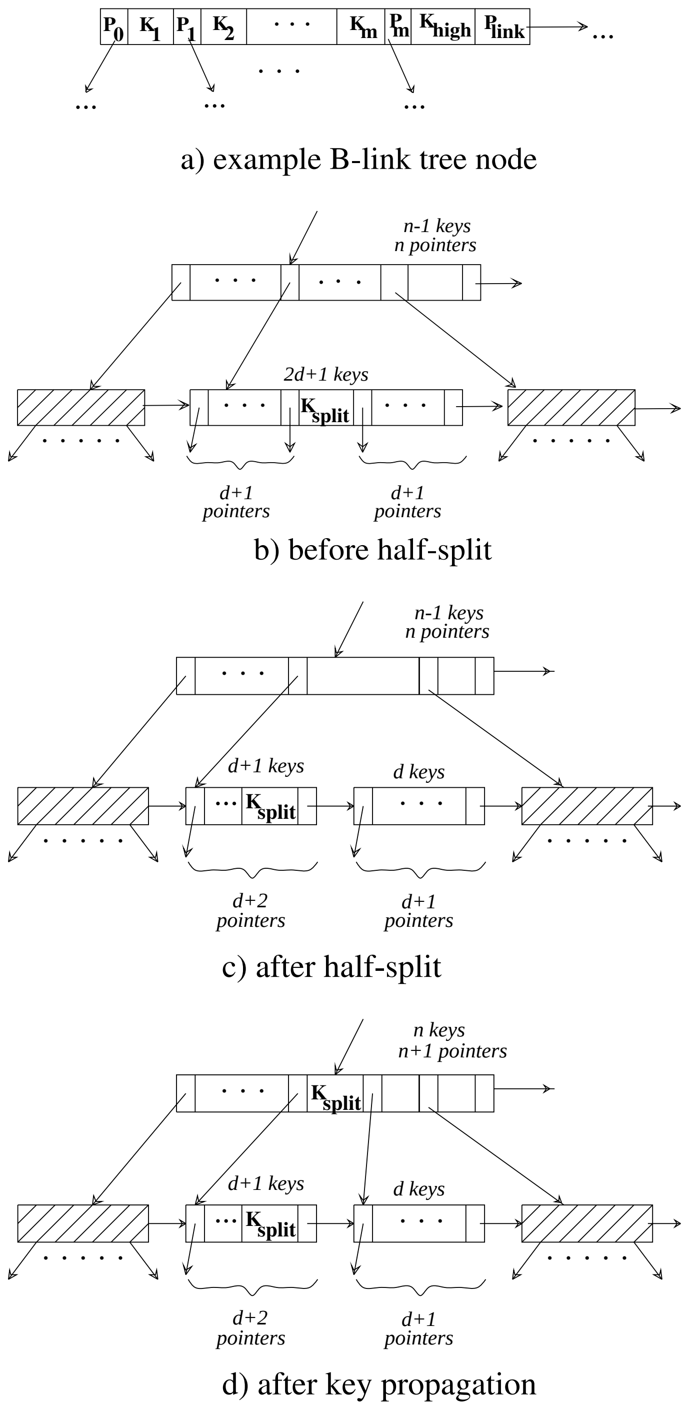



Figure 2.4: A B-link Tree Page Split

*link-chase*. Merges can also be done in two steps [LS86], via a half-merge followed by an entry deletion at the next higher tree level. The B-link algorithms differ from the earlier ones in that they don't lock-couple in their descent.

We consider one B-link algorithm in which readers descend the tree using IS locks, releasing each lock before getting a lock on the next node. Updaters also behave like readers till they reach the leaf node when they release their IS lock and try to acquire an X lock on the same leaf. After acquiring the X lock, 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 one node before asking for the next. If the leaf is unsafe, then they do a half-split or a half-merge. They then release the lock on the leaf before searching (again using link-chases) for the parent node from the last node (at the next higher level) that they used in their descent. Thus readers and updaters don't lock-couple at any stage and hold a maximum of one index node lock at any time.

We consider only this algorithm, referred to as the *LY* algorithm, since this was found to have the best performance among all other algorithms in this class [SC91].

## 2.6 Other Algorithms

### 2.6.1 ARIES/IM Algorithm

This algorithm was proposed for high-concurrency index management in [ML89]. This uses a B-tree similar to the B-link tree except that higher level pointers to sibling nodes are not present here. At the leaf level, nodes have both left and right links. Updaters in ARIES/IM make an initial descent to the leaf using IS locks, and at the leaf level they perform link-chases as in the B-link algorithm. However, unlike the B-link algorithm which performs link-chases at all levels, ARIES/IM uses a complex protocol based on recursive restarts to perform the equivalent of a link-chase at higher levels. This mechanism does 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. In our experiments, we observed that all link-chases were at the leaf level. Since the leaf-level ARIES/IM algorithm resembles the B-link algorithm in most respects, we can expect this algorithm to perform close to the LY (B-link) algorithm [SC91] and hence don't explicitly simulate this algorithm.

## 2.7 Summary

In this chapter, three classes of B-tree concurrency control algorithms were discussed, Bayer-Schkolnick, Top-Down and B-link (here afterwards, we refer to the LY (B-link) algorithm as the B-link algorithm). In addition the ARIES/IM algorithm was also discussed, but it is not simulated in our experiments since it is expected to work as well as the B-link algorithm. All the above algorithms are proposed for the conventional database domain. This necessitates their adaptation to the real-time domain. This issue is explored further in the next chapter.

# Chapter 3

# Real Time Index Concurrency Control

## 3.1 Introduction

In the previous chapter, we saw a representative set of B-tree concurrency control algorithms. All those algorithms work in the conventional database case. These have to be adapted to the real-time domain, since in the real-time (RT) domain, the objectives are very different. As stated earlier, this is achieved by modifying the transaction resource scheduling policies, to suit the real-time domain.

Contention can be resolved by means of a scheduling policy. Contention comes in two forms, at physical resources (system resources) and at logical resources (data and index). Resource contention is a system dependent property in the sense that it can be decreased by increasing the hardware functionality of the system. Data contention arises due to the data concurrency control protocol. Index contention arises due to transactions having to acquire index node locks before performing any operation on the index.

In the conventional database case, all transactions were given the same priority. The real-time domain, however, typically involves a priority-driven environment, where transactions which are closer to their deadlines have to be given higher priority than those that are far away from their deadlines. In such an environment, higher priority transactions have to be given preference over lower priority transactions at any place where there is contention in the system.

## 3.2 Priority Inversion

All the B-tree algorithms discussed so far have no notion of the priority of a transaction. Also every algorithm involves a blocking either at data or index for acquiring locks or latches respectively or at the system resources. This leads to what is called as *priority inversion* [SRL87], where a high priority transaction is blocked by a low priority transaction. Priority inversion can also occur if a higher priority transaction is restarted by a lower priority transaction. This may lead to the high priority transaction missing its deadline because of the wait or restart and the low priority transaction finishing much before its deadline. This is clearly not desirable. Thus the B-tree CC algorithms have to be adapted to handle priority inversion conflicts properly. The importance of handling priority inversion conflicts correctly has already been shown by many simulation experiments [AG92].

Priority inversions can be handled by two mechanisms [AG92],

- *Priority Inheritance*

  Here, the low priority transaction that is holding the resource, inherits the priority of the high priority transaction for the duration it holds that particular resource. Priority inheritance is transitive in the sense that if A, B and C are transactions with priority descending in that order, and A waits for B and B waits for C, then C inherits the priority of A through B. The problem with this approach, is that at workloads involving high contention, the transitive property is applied many times resulting in most or all of the transactions being at the same priority [HSRT91] (since most of them would have risen to the highest priority). Thus it may reduce to a situation where all transactions have equal priority, or in other words, a conventional database.

- *High Priority*

  Here, the low priority transaction is preempted from holding the resource, which is then scheduled to the high priority transaction. If the resource is preemptible, e.g. CPU, then the low priority transaction can be resumed at a later point of time. If the resource is not preemptible, e.g. disk, then that particular operation on that resource has to be restarted. Moreover, if a logical resource like data is preempted, then the whole transaction might need to be restarted. The problem with this restart is that a transaction B might restart another transaction C, and in turn be restarted by another transaction A, thus wasting the restart of C. This problem is called the problem of *wasted restarts* [HCL92].

Blocking at system resources is handled by the system and the method used to handle priority inversion at system resources will be outlined in Chapter 5. In the following sections, we describe, in detail, how we handle priority inversions due to data or index contention.

## 3.3 Index Node Conflicts

Priority in the index CC algorithms has been implemented in the following manner, similar to [GHSS95]. Each index node has a queue of waiting transactions maintained in priority order. If a transaction requests a lock on an index node that is currently conflicting with the lock already held on that node, and the node is locked by a higher priority transaction then the requester waits in the queue of that index node. On the other hand, if the requester is of higher priority than the transaction or set of transactions holding the index node lock and the lock mode conflicts with the request lock mode, then all transactions currently holding the index node lock are preempted and the lock awarded to the requester. The preempted transactions have to only restart the current index operation (not the entire transaction). An exception to this case is when the holder is *physically* making a modification to the index node. In this case, the holder is not preempted. The holder is allowed to continue its update and other transactions, whatever their priority be, wait till the holder finishes its operation and releases the index node locks. The effect of this special case is not expected to be significant, since these latching operations are pretty fast thereby making the priority inversion periods negligible.

Data locks are usually acquired and held till the commit time of the transaction. On the other hand, index node locks are generally held for very short durations. Commercial database systems implement index node locks as *latches* [ML89] (henceforth a reference to a lock implies a data lock).

A latch is similar to a semaphore and is used to maintain physical consistency. It can also be considered as a fast lock and can be used for things such as locking an index node, accessing a page of the buffer pool. Since the number of latchable items is generally very small, latches don't have the problem of name translation as in locks where the number of values that data items take is numerous. Also they are deadlock free and hence don't have the overhead of deadlock detection. Since all the index CC algorithms discussed earlier don't have a possibility of deadlock due to an index node lock, an index node lock can be implemented using a latch.

Now the other question that occurs is, if index node locks are very short duration locks, then is there any benefit in adding preemption to such locks. It was shown in [GHSS95] that

- Adding preemption to index node locks is indeed advantageous in terms of performance
- Adding preemption doesn't lead to any inefficiency in latch requests since the only extra work to be done is in comparing transaction priorities, which is readily available.

We, therefore, adopt the same line of thought and add preemption to index node locks.

## 3.4 Data Conflicts

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. Ensuring transaction serializability in this case is much more complicated than in the case where transactions consist of single index actions only. For this, we use a real-time variant of a simplified form of the well-known ARIES *Next-Key-Locking* (KVL) protocol proposed in [Moha90] to provide transaction data concurrency control. In formulating this simplified algorithm, we assume that all indexes are unique.

The next-key-locking-based data concurrency control 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, the preemption mechanism that is used is the same as that used in the index concurrency control algorithm, i.e. a higher priority requester preempts a lower priority holder if the lock modes conflict. The only difference between this preemption and index node lock preemption is that a preemption of an index node lock means a restart of that index operation only, while preemption of a key-value lock means the restart of the *complete* transaction itself.

A study conducted in [HCL92] regarding transaction concurrency control in firm real-time database systems had found that optimistic algorithms perform better than locking protocols. In particular, WAIT-50 was found to perform better than 2PL-HP. However, the result holds true only in workloads where there is very high data contention. In the workloads considered in this study, there is very less data contention, and there is more of index contention. Thus the choice of transaction concurrency control algorithm can be expected to have very less impact on the results. Also, as cited in [GHSS95], integrating optimistic protocols with index management and recovery are open problems [Hard84]. However, one positive feature of the optimistic protocols is that the effects of the transaction that misses its deadline, and is hence aborted, need not be undone since the updates are delayed till commit time.

## 3.5 Summary

In this chapter, we considered how the conventional B-tree CC algorithms were adapted to the real-time domain. Index node conflicts and data conflicts are handled by preemption if the holder is of lower priority than the requester and are handled by waiting in the other case. In the next chapter, we will be dealing with the earlier work done in [GHSS95], and also discuss in detail the limitations of that study, and our solutions to them.

# Chapter 4

# Contributions

In the earlier chapter, we discussed how to adapt the B-tree Concurrency control algorithms to the real-time domain. In this chapter, we will see the results of the earlier work, its limitations and our contributions. The only earlier work (to the best of our knowledge) is the one done in [GHSS95]. In fact, this study starts off from where [GHSS95] left.

## 4.1 Earlier Work

The study in [GHSS95] considered the Bayer-Schkolnick, Top-Down and B-link algorithms, with adaptation to the real-time domain. They modeled index node locks as latches, used a simplified form of KVL for transaction serializability. The performance metric considered was the *Miss Percent*. They used Earliest Deadline for transaction priority assignment. The study didn't model undos of index actions of aborted transactions. On the other hand, it surmised that modeling undos had a positive feedback, i.e. implementing undos would in turn increase the miss percent, since more of misses implies more of undo work. A single B-tree was considered for the experiments. Since the B-tree height was only three, they didn't notice any significant difference in the results for the corresponding Bayer-Schkolnick and the top-down algorithms. Thus the algorithms considered were X, SIX, OPT and B-link. The B-tree operations considered were *searches, inserts, deletes,* *appends*. They conducted the following experiments,

- *Low Index Contention*

  In this experiment, the workload consisted of 80% searches, 10% deletes and 10% inserts. The high percentage of searches resulted in a low index contention environment. The results they observed were that at low arrival rates, the B-link algorithm performed better than all others which was in agreement with the earlier non-real-time experiment results. At high arrival rates, however, SIX algorithms performed better than B-link. The reason was that SIX algorithms favoured readers more than updaters, a fact already explained in Chapter 2. Thus SIX algorithms completed nearly all the readers (percentage of readers was 21%) whatever the arrival rates be while B-link completed lesser readers and updaters with increasing arrival rates. In the corresponding non-real-time experiment, the slower updaters used to clog up the system, leading to higher contention, resulting in SIX performing badly, while in the firm deadline model, the updaters missing deadlines are thrown immediately out of the system resulting in the clogging inherently being prevented. The B-link algorithm performed badly at very high arrival rates because of the high resource utilization that is inherent in the algorithm. The SIX algorithms implemented a form of *load control* by favouring readers. The B-link algorithms on the other hand do not favour any particular class of transactions. The study then proposed a real-time variant of the B-link algorithm termed *Load Adaptive B-link* (LAB-link). This algorithm monitored the system resource utilization and applied load control by not admitting a transaction if the utilization at any resource at the time of arrival of that transaction was more than *MaxUtil* and the miss percent was more than *MinMiss*. In this experiment, the bottleneck resource was the disk. This was found to perform better than SIX as well as B-link.

- *Moderate Index Contention*

  The workload here consisted of 100% inserts. The OPT and B-link algorithms were found to perform better than SIX and X. In the corresponding non-real-time experiment, OPT was found to perform badly. The reason OPT performed well in this case was attributed to the effect of index node preemption in the real-time case. The preemption of index node locks resulted in a leaf node being unsafe for a shorter period of time resulting in lesser number of updaters executing the second pass. Here too LAB-link performed the best.

- *High Index Contention*

  This workload consisted of 25% searches and 75% appends. The appends are concentrated in the rightmost portion of the index tree thereby resulting in a high contention environment. Here too, B-link and OPT performed better than SIX, while LAB-link performed the best. However, B-link performed slightly better than OPT, because of a higher probability of multiple transactions restarting in this case, since all index actions are directed to the right end of the tree. Here too LAB-link performed the best, the difference however being, the bottleneck resource was the CPU.

- *No Resource Contention*

  Miss percent, can be attributed to three types of contention, data index and resource. In the workloads considered, the data contention effect was negligible since there were very few number of transaction restarts due to data conflicts. In order to isolate the effect of index contention, they removed the resource contention part by having infinite resources, cpu, disk and memory thereby removing any waiting at the resource level. This helped in isolating the effect of index contention. They carried out the previous experiment without resource contention and found B-link and OPT to perform significantly better than SIX. This means that B-link and OPT saturate their resources more than SIX and their index contention component is very less compared to SIX.

- *No Index and Data Contention*

  To study the effect of index contention on the algorithms, they conducted a 100% search workload experiment. Since this workload has no data or index contention, the performance improvement that one gets in this experiment directly translates to the index contention component (since the effect of data contention is negligible). They observed that the effect of index conflicts is significant (20% improvement in miss percent for high arrival rates) even for B-link algorithms.

Although this study was the first of its kind, it has its share of limitations too.

## 4.2 Limitations

As stated briefly earlier, the limitations of this study are,

1. The LAB-link algorithm, as we will show in this study, is effective only for a restricted class of operating environments. Moreover, its performance is sensitive to the choices made for its algorithmic parameters (i.e. not robust).

2. Undoing of an aborted transaction, which is essential to maintain database consistency, was not modeled. However they surmised that for the performance with undos, a large miss percent implies more clean-up work thus increasing the miss-percent. In other words, the assumption was that there is a positive feedback between undos and miss percent.

3. The only performance metric considered was miss percent. Other metrics like fairness of the ICC algorithm with respect to the completing transactions, were not considered. These metrics give more insight into the working of the algorithm, as to whether the algorithm inherently gives preferences to transactions of a particular class.

4. The study was limited to point (single key) index operations. Range operations which are a basic form of index access, were not modeled at all.

In the next few sections, we attempt to overcome these limitations and consider few other issues too.

## 4.3 AED-link

In this section, we present the new load-control-based ICC protocol we developed, termed **AED-link**. In formulating the AED-link real-time ICC algorithm we use the description of the AED algorithm proposed in [HLC91] as an improvement over Earliest Deadline. Both these algorithms deal with the question of assigning priorities to transactions. Earliest Deadline operates by assigning the highest priority to a transaction that has the nearest deadline, i.e. which has the least amount of remaining time left to complete. Earliest Deadline [Liu73] has been observed to minimize the number of late transactions in the system if data or resource contention is low or moderate. On the other hand, it has been observed practically in [AG88, AG89] that the performance of Earliest Deadline steeply degrades in an overloaded system. This is because, in a highly overloaded system, a transaction gets high priority only when it comes very near to its deadline thus making it impossible to finish it within time. Because of this, other transactions, which can be completed if time is not wasted on the nearest deadline transaction, get delayed.

Thus there is a need for an algorithm which behaves as well as Earliest Deadline in lightly loaded situations and performs better in overloaded situations. *Adaptive Earliest Deadline* (AED) achieves this by a feedback process which does not require any knowledge of transaction characteristics. The problem with Earliest Deadline in overloaded times is that it tries to find a schedule for all transactions existing in the system when it is definite that all of them cant be finished in time. Therefore the idea is to find out the largest set of transactions that can be finished in time, and then apply an Earliest Deadline schedule to them.

### 4.3.1 Group Assignment

The AED algorithm dynamically splits all transactions entering the system into two groups, *HIT* and *MISS*. The group assignment algorithm is illustrated in Figure 4.1. The idea is to collect into the HIT group the largest set of transactions which can be completed in time and leave the rest to the MISS group and finish all the transactions in the HIT group using an Earliest Deadline schedule and not do anything in the MISS group. This it

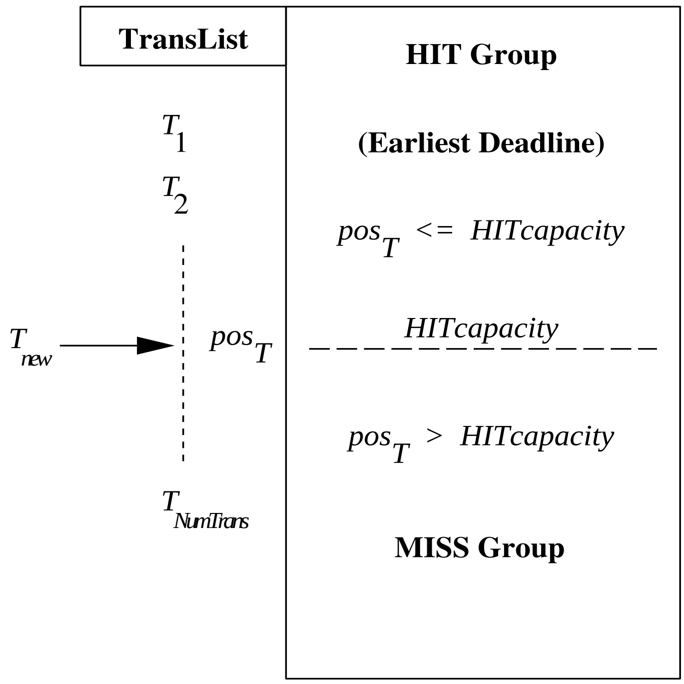



Figure 4.1: AED Priority Mapping

does by the following method. It maintains a dynamically monitored variable *HITcapacity* which refers to the size of the HIT group. A transaction that comes into the system is assigned a unique 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 *HITcapacity* then the transaction is assigned the HIT group else it is assigned the MISS group. Transactions that cannot be accommodated in the HIT group are estimated to miss their deadlines and are therefore assigned to the MISS 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).

### 4.3.2 Priority Assignment

Essentially priority assignment is done in two levels. The group assignment itself acts as one level of priority assignment. All transactions belonging to the HIT group have a higher priority than all transactions belonging to the MISS group. The priority assignment for transactions within the same group is done by the following formula,

$$P_T = \left\{ \; (0, D_T, \; I_T) \quad if \; Group = HIT \right.$$

where $A_T, D_T, P_T$ denote the arrival time, deadline and priority of the transaction respectively.

The priority of a transaction $P_T$ is thus a vector with three components, with priority comparisons made in lexicographic order, i.e. the first components of the two transactions' priorities are compared and if both are equal, only then is the second component examined and so on. If they are unequal, the one with the lesser component has the higher priority. Thus, with the above scheme, all transactions in the HIT group have a higher priority than those in the MISS group. Within the HIT group, transaction priorities are assigned according to Earliest Deadline. The third component serves to break ties in the HIT group, i.e. if two transactions have the same deadline, then their $I_T$ determines their priority.

### 4.3.3 Feedback Process

As already stated, the goal of AED is to find the size of the largest set of transactions that can be completed. It tries to accommodate that set in the HIT group and it estimates the rest of the transactions as those which cannot be completed within their deadlines and puts them in the MISS group. The transactions admitted into the MISS group are denied entry into 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. If we define *hit ratio* within a transaction group as the number of transactions within that group which meet their deadline, then the goal of AED can be stated as achieving a hit ratio of 1.0 in the HIT group and a hit ratio of 0.0 in the MISS group, thereby doing *only* those transactions that *can* be done. To achieve this, it has to predict the value of *HITcapacity* accurately, which is difficult since the transaction characteristics are not known. Hence it tries to compute the *HITcapacity* value accurately by a feedback process.

The feedback process operates as follows. It uses two parameters *HITbatch* and *ALL-batch*. The meaning of the parameters will be clear from the following discussion. Initially the *HITcapacity* is set to an arbitrary value (it was 100 in our experiments, the results were negligibly different for different initial values of *HITcapacity*). The incoming transactions are assigned a group by the above group assignment algorithm. Each transaction that is assigned to the HIT group is marked. After a set of *HITbatch* transactions have been assigned to the HIT group, the status of these marked transactions at the output is monitored. Two quantities are measured : *HitRatio(HIT)* and *HitRatio(ALL)*. *HitRatio(HIT)* refers to the fraction of the marked transactions that were completed in time. *HitRatio(ALL)* refers to the fraction of the last *ALLbatch* transactions which complete before their deadline. In the following discussion, *NumTrans* refers to the number of transactions that are present currently in the system. Once the last marked transaction exits the system, the above two ratios are measured and the HITcapacity value is changed accordingly as follows,

*(1)* $HITcapacity := \lceil HitRatio(HIT) * HITcapacity * 1.05 \rceil;$

*(2)* *if* $HitRatio(ALL) < 0.95$ *then*

$HITcapacity := Min(HITcapacity, \lceil HitRatio(ALL) * NumTrans * 1.25 \rceil);$

In the first step, based on the value of *HitRatio(HIT)*, the size of the HIT group is proportionally changed. The aim, however, is not just to have a *HitRatio(HIT)* of 1.0, but also to have the largest *HITcapacity* value. So a factor of 1.05 is introduced to increase the *HITcapacity* value. The *HITcapacity* value goes on increasing till *HitRatio(HIT)* is reduced to less than 0.95 (approximately, since $0.95 * 1.05 \simeq 1.0$). After that *HITcapacity* becomes lesser and this in turn increases *HitRatio(HIT)*. Thus the first step takes care of the *HITcapacity* setting pretty accurately.

One problem with this approach is that if the system faces a very high period of no overload, then the *HITcapacity* value is bound to increase (due to step *(1)*) to a very large value, since *HitRatio(HIT)* would be 1.0 throughout. Suddenly if the system becomes overloaded, it would take the system a lot of time to get the *HITcapacity* down to the actual value by step *(1)*. In the meantime, since the *HITcapacity* is very large, all transactions may be accommodated in the HIT group, which is the same as the unstable high-miss region of Earliest Deadline. This is clearly undesirable. So we have to bring the *HITcapacity* down to the appropriate value, quicker than step *(1)*. The second step takes care of this. The *HitRatio(ALL)* value becoming less than 0.95 signals this overloaded situation. In this case, the *HITcapacity* is not allowed to go more than an upper bound. This upper bound is set to be 25% greater than the *right* value of *HITcapacity* 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* *HITcapacity* value). After *HITcapacity* is brought down close to the *right* value, step *(1)* acts as a fine tuning mechanism to adjust the *HITcapacity* value.

One should note that the feedback in AED is not the same as the feedback used by the LAB-link algorithm. There the feedback involved measuring system parameters. Here the feedback involves measuring *HitRatio(ALL)*, *HitRatio(HIT)* and *NumTrans* all of which can be done at the output of the RTDBS. Also, the feedback in AED is based on end-user performance metrics rather than system performance metrics as in LAP. This way it is a more robust solution than the load control implemented by LAB-link. One can also compare the two in the following manner : While AED looks at miss percent, thereby taking into account the effect of index contention, data contention and resource contention LAP looks at only resource contention. LAP wont work in cases, where there is less resource contention and more of index or data contention.

### 4.3.4 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* parameters 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*.

## 4.4 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 the 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 update action involves performing its *compensating* action (insert for a delete, delete for an insert or apend) on the B-tree. On completion of this compensating action, the corresponding key value lock is released. (except when another to-be-undone action involves the same key value).

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. Within the undo transactions, the relative priority ordering is the same as that which was existing before they started their undo (i.e. their original priorities).

## 4.5 Performance Metrics

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. We have also considered other performance metrics which are important from a fairness viewpoint. The two other performance metrics we have dealt with are

- *SizeFairness*

  This parameter tries to capture the extent to which bias is exhibited towards transactions based on their sizes. This is evaluated using the formula

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

  Thus an algorithm which shows no bias towards transactions of a particular size will have a SizeFairness of 1.0, and one which shows bias towards transactions of shorter size will have a SizeFairness less than 1.0.

- *TypeFairness*

  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 this formulation, a protocol is ideally type-fair if it has a TypeFairness factor of 1.0, and favours readers to updaters if its TypeFairness is less than 1.0.

## 4.6 Different Deadline Formulae

In the earlier study, the deadline of a transaction is calculated using the following formula. If a transaction arrives at time $A_T$ and its expected resource time is $R_T$, then its deadline is given by the formula,

$$D_T \ = \ A_T \ + \ SF \ * \ R_T$$

$SF$ denotes the slack factor (a term signifying the tightness/slackness of a transaction with respect to its deadline). The earlier study used a single slack factor for all their experiments. For completeness sake, we tried out other slack factors too for the above deadline formula. We also considered a uniform distribution for the slack factor, where a transaction's slack ratio is chosen uniformly over the range given by *LSF* and *HSF* [HCL92]. The corresponding deadline formula becomes,

$$D_T \;=\; A_T \;+\; Uniform(LSF, HSF) \;*\; R_{T_{max}}$$

where $R_{T_{max}}$ is the expected resource time of the transaction with the maximum resource requirement.

## 4.7 Range Queries

The earlier study had considered only point queries, i.e. queries involving single key actions. We also consider the effect of range queries (involving multiple key-values) on the different ICC protocols. The index operations now considered are point search, range search, insert and delete. The range search is implemented as follows. Instead of specifying the two end-key-values for the range search, we specify only the start value. The end key value is computed as a fixed number of keys after the start key value. The range search returns all the keys starting from the start-key-value till the smallest key after the end-key-value. The tree is initially traversed down to the leaf where the index node corresponding to the start-key-value is latched in S mode. An S lock is then obtained on the key value (or the next key value if the key is not present). A data access operation is modeled on the key value returned. The range search then continues by locking the next key value in S mode. The range search follows right sibling pointers as and when required to find the next key value. Inserts now operate as follows. They first descend the tree to the leaf and then obtain an X lock on the key value to be inserted as well as the next key value, perform the insert and then release the lock on the next key value. Deletes descend the tree to the leaf and then obtain an X lock on the to-be deleted key value as well as the next key value, delete the key value and then release the X lock on the deleted key value.All the key-value locks obtained are held till commit time.

## 4.8 Summary

This chapter presented earlier related work, its limitations and our contributions. We used AED for priority assignment instead of Earliest Deadline. We modeled the undoing of index actions of aborted transactions. We also considered two other performance metrics which measure the biases that the ICC algorithm shows towards transactions of a particular class. For completeness sake, we also considered different real-time situations which we model using different slack factors and different distributions for the slack factor. We also considered the performance impact of range queries on the different ICC protocols. We present the results of our experiments in Chapter 6. In the next chapter, we discuss the simulation model used.

# Chapter 5

# Simulation Model

In the earlier chapters, we considered various index concurrency control algorithms and their real-time variants. In order to perform a comparative study of them, we used a detailed simulation model of a firm-deadline RTDBS [GHSS95]. We explain this model in the following sections.

## 5.1 RTDBS Model

The RTDBS consists of six components.

A transaction is generated at the the *source* and goes to the *transaction manager*. The *transaction manager* controls all the index and data operations that need to be done on the transaction. Before each index operation, the *transaction manager* obtains the appropriate lock for the data item from the *concurrency control manager*. The *concurrency control manager* controls the acquiring and releasing of key-value-locks and index-node-locks (latches). For each index operation, the *transaction manager* calls the *B-tree manager*. *B-tree manager* controls all single index operations, i.e. implements the index concurrency algorithms, and is modeled as in [SC91]. It is this part that varies with the concurrency control algorithm used. This calls the *concurrency control manager* to request latches on index node pages. It also calls the *buffer manager* for buffer requests and *resource manager* for CPU time. The *buffer manager* on its part calls the *resource manager* for CPU

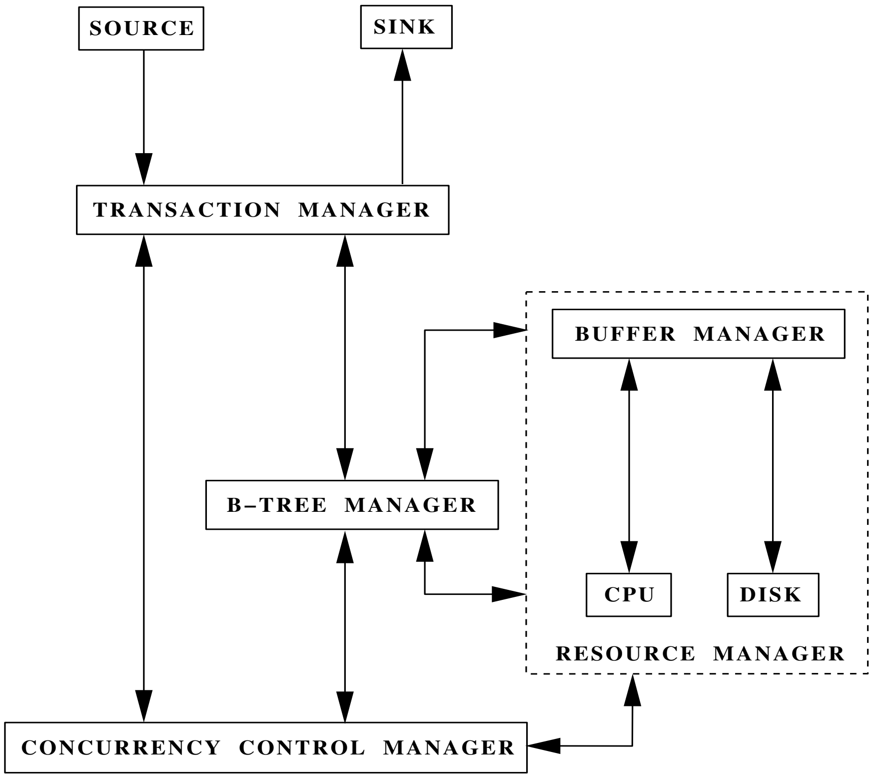



Figure 5.1: Simulation Model

and disk I/O. The *concurrency control manager* calls the *resource manager* for CPU time. After each index operation is finished, the *B-tree manager* returns back to the *transaction manager*. After all index operations are finished, the *transaction manager* returns the transaction to the *sink* where statistics regarding the transaction are collected.

A summary of the parameters used in the model is given in Table 5.1.

## 5.2 Workload Model

As mentioned earlier, transactions arrive as a *poisson* process and each transaction has a deadline associated with it. Each transaction consists of a sequence of index operations such as *search (point or range), insert, delete or append* of a key value. After each index operation, the corresponding data access 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 time equal to one disk I/O. A transaction that misses its deadline

| Parameter | Meaning |
|---|---|
| ArrivalRate | Transaction arrival rate |
| TransSize | Average transaction size |
| SlackFactor | Deadline Slack Factor |
| LSF | Low Slack Factor |
| HSF | High Slack Factor |
| SearchProb | Proportion of searches |
| InsertProb | Proportion of inserts |
| DeleteProb | Proportion of Deletes |
| AppendProb | Proportion of Appends |
| RangeKeys | Size of range search |
| InitKeys | Number of keys in the initial tree |
| MaxFanout | Key entries per node |
| NumCPUs | Number of processors |
| SpeedCPU | Processor speed in MIPS |
| LockCPU | Cost for lock/unlock |
| LatchCPU | Cost for latch/unlatch |
| BufCPU | Cost for buffer call |
| SearchCPU | Cost for page search |
| ModifyCPU | Cost for key insert/delete |
| CopyCPU | Cost for page copy |
| NumDisks | Number of disks |
| PageDisk | Disk page access time |
| NumBufs | Size of buffer pool |

Table 5.1: Model Parameters

is immediately aborted and discarded from the system (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 earlier in Chapter 3. If a transaction gets restarted due to data conflicts with higher priority transactions, then it does the same sequence of operations as its original incarnation.

The *ArrivalRate* parameter specifies the mean arrival rate of the transactions. The number of index operations in a transaction, termed its size, varies between half and one-and-a-half times *TransSize*. The overall proportion of searches, inserts, deletes and appends in the workload is given by the *SearchProb, InsertProb, DeleteProb* and *Append-Prob* parameters respectively. *RangeKeys* specifies the number of keys fetched per range search.

Deadlines are assigned to the transaction using the following 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 the transaction *T*, while *SF* is the slack factor. As discussed earlier, we tried out a uniform distribution for the slack factor too. The deadline formula used in this case is

$$D_T \ = \ A_T \ + \ Uniform(LSF, HSF) \ * \ R_{T_{max}}$$

where $R_{T_{max}}$ is the expected resource time of the transaction with the maximum resource requirement.

The *resource time* gives the time the transaction will spend on an average at the system resources i.e. Buffer, CPU, Disk. Since it is the disk time that dominates the others, we take the disk time involved for all index actions into account as the transaction's resource time. For a search, we assume *two* disk I/Os, since due to buffer caching, all the index nodes except those of the leaf will be present in the buffer, and getting the required page into the buffer may require a page replacement. For an insert or delete, we assume, in addition to the time that it takes for a search, a time of $1/MaxFanout$ disk I/Os since $1/MaxFanout$ is the probability (approximately), that a split will occur and in that case, one more disk I/O is required since an existing page has to be replaced to create the new index page or deleting an existing one. For the *no resource contention* experiment, however, since there are infinite buffer pages, and the whole of the index tree is in memory, there is no disk I/O involved. Here we consider the time spent in the Buffer and CPU for each index action to compute the resource requirement of the transaction.

The *SlackFactor* is a constant that specifies whether a transaction's deadline is tight or slack. A slack factor of *1.0* means a tight deadline, of $<$ *1.0* means an infeasible deadline and a slack factor of $>$ *1.0* means a slack deadline. Here, although the workload generator uses the transaction resource requirements to generate deadlines, the RTDBS has no knowledge of the transaction characteristics. Thus, the transaction is discarded from the system only when it misses its deadline. On the other hand, if the RTDBS knows the resource requirements of the transaction, then it could discard the transaction, the moment it realizes that the remaining time left before the transaction's deadline is less than the transaction's remaining required resource time To use the terminology of [AG92], the overload detection is *observant* and not *predictive*.

## 5.3 B-Tree model

We model only a single B-tree, for simplicity, and all transaction accesses are made onto this B-tree. The key values of this B-tree are assumed to be of the same size, and the key is assumed to be a candidate key of the underlying relation, i.e. no two tuples of the relation can have identical values on the indexed attribute. Each index node corresponds to a disk block is assumed to have a maximum of *MaxFanout* number of <separator, pointer> entries (recall that a separator defines one step in the search path to a leaf page). The total number of keys in the initial B-tree is given by the *InitKeys* parameter. We assume, the index to be a unique index. Unlike in the previous study [GHSS95], index operations of update transactions that are aborted are undone. As discussed earlier, complementary update index operations of the aborted update transaction are carried out in reverse chronological order.

## 5.4 Resource Model

The resources in the system consist of processors, disks and memory. Since this is a firm RTDBS where we want to meet the maximum number of deadlines, resource scheduling has to be priority driven. There is a single queue of transactions waiting for the processors. If there are $n$ processors, then at any point of time, the processors will be servicing the $n$ highest priority transactions. All the others will be waiting in the single queue, which is also priority-ordered. If a higher priority transaction arrives, thus becoming one of the $n$ highest priority transactions, the least priority transaction that was being serviced is preempted to accommodate the newcomer. Similarly, each of the disks has its own priority-ordered queue and the highest priority transaction at any point of time is serviced.

Buffer management is implemented through a two level priority LRU mechanism. Higher priority transactions steal buffers from the lowest priority transaction that has one or more pages currently in the buffer pool. The least recently used clean buffer of this transaction is the one chosen for replacement. If there is no clean buffer, then the least recently used dirty buffer is chosen for replacement, flushed to disk, and then transferred to the higher priority transaction. The lowest priority transaction, uses a similar LRU mechanism within its set of buffers.

The number of resources in the system is determined by the *NumCPUs, NumDisks* and *NumBufs* parameters. The processing cost of each of these resources are determined by the other parameters listed in Table 5.1.

## 5.5 Summary

In this chapter, we discussed the simulation model used for our experiments. We have incorporated into our model different real-time situations by assuming a uniform distribution for the slack factor. Now that the real-time index CC algorithms and the simulation model used to implement them have been explained, we discuss in the next chapter, the experiments we performed, state the results we obtained and validate those results with supporting statistics.

# Chapter 6

# Experiments

This chapter presents in detail the simulation experiments that we performed, the results obtained and the explanations for those results. The simulator was written in C++ [Str86]. The index concurrency control algorithms that we compare are B-X, B-SIX, B-OPT, TD-X, TD-SIX, TD-OPT, B-link, LAB-link, AED-link. The priority assignment used in all the algorithms (except for AED-link) is Earliest Deadline, where a transaction with a closer deadline is given higher priority than a transaction with a later deadline. In AED-link, the priority assignment used is Adaptive Earliest Deadline (discussed in Chapter 4). The initial index generation and the parameter settings are the same as those considered in [GHSS95].

## 6.1 Initial Index Generation

As stated earlier, we model only a single B-tree, whose key is assumed to be a candidate key of the underlying relation. Each index operation is either an insert, delete, append or search and is a point query (operates on a single key). The key range space for the insert, delete and search operations consists of integer values from 1 to 300,000. In order to avoid interference at the level of key values, deletes take key values which are perfect multiples of 3, while inserts take key values that are not multiples of 3. This simplifying assumption is reasonable since we are interested in the relative performance of the ICC algorithms, and one can assume that contention due to key-value interference will mostly be the same for all the algorithms. Searches can use all the key values in the key range space. Appends use key values from 300,001 onwards. In order to avoid a situation where a key to be deleted is not present in the tree, we initialize the tree with all the key values that delete operations can take. The tree is formed by inserting, one by one, a random permutation of all the delete key values. Thus the number of keys in the initial B-tree, *InitKeys*, is 100,000.

## 6.2 Performance Metrics

The performance metrics considered (as explained earlier) are MissPercent, SizeFairness and TypeFairness. While MissPercent measures the percentage of transactions that are not able to complete before their deadline, SizeFairness measures the bias shown towards the size of a transaction and TypeFairness measures the bias shown towards the type of a transaction. A value of 1.0 for SizeFairness indicates the algorithm shows no bias towards a particular sized transaction class and similarly a value of 1.0 for TypeFairness indicates the algorithm as fair towards readers and updaters.

All the graphs 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 20,000 transactions were completed by the system. The simulator was programmed to generate a lot of other information, including resource utilizations, number of link-chases for B-link algorithms, number of restarts for optimistic algorithms, etc. These secondary measures help explain the performance behaviour of the B-tree algorithms under various workloads and system conditions.

## 6.3 Parameter Settings

The values that the parameters of our simulation model take are as follows. The *MaxFanout* value, the maximum number of <separator, pointer> pairs in an index node, was fixed at 300. With this setting, the B-tree generated in the above section was 3 levels deep, with 1 root, 2 nodes at the second level, and 506 index nodes at the leaf level. The tree nodes are uniformly distributed across all of the disks. The values for specific constant parameters are given in Table 6.1. Other parameters are set according to the experiment under consideration.

| Parameter | Value |
|---|---|
| TransSize | 8 |
| InitKeys | 100,000 |
| MaxFanout | 300 |
| SpeedCPU | 20 MIPS |
| LockCPU | 1000 instructions |
| LatchCPU | 100 instructions |
| BufCPU | 1000 instructions |
| SearchCPU | 50 instructions |
| ModifyCPU | 500 instructions |
| CopyCPU | 1000 instructions |
| PageDisk | 20 ms |

Table 6.1: Parameter Values

The set of parameters that are varied in our experiments are shown in Table 6.2. *ArrivalRate* is the basic parameter that is varied across all experiments. All performance metrics are plotted as a function of the *ArrivalRate* only. The experiments considered are low index contention (LIC), moderate index contention (MIC), high index contention (HIC), no index contention (NIC), no resource contention (NRC), sensitivity analysis, effect of AED and LAB on other ICC algorithms, different real-time situations, index contention with undo and low index contention with range queries.

The *SearchProb, InsertProb, DeleteProb, AppendProb* parameters are varied according to the degree of index contention required. The *RangeKeys* parameter is used only in the range queries experiment and is assigned the value = 10. The system resource related parameters are assigned values $NumCPUs$ = 1, $NumDisks$ = 8 and $NumBufs$ = 250 for all the above experiments except for the *No Resource Contention* experiment, where all of them are set to *infinity*.

In addition to the above mentioned model parameters, there are other parameters

| Parameter | Range of Value |
|---|---|
| ArrivalRate | $0 - \infty$ |
| SlackFactor | 2, 4, 8 , Uniform [2, 6] |
| SearchProb | 0.0 - 1.0 |
| InsertProb | 0.0 - 1.0 |
| DeleteProb | 0.0 - 1.0 |
| AppendProb | 0.0 - 1.0 |
| RangeKeys | $1 - \infty$ |
| NumCPUs | $1 - \infty$ |
| NumDisks | $1 - \infty$ |
| NumBufs | $1 - \infty$ |

Table 6.2: Range of Values for Variable Parameters

related to the load control policy used.

- Load Adaptive Policy

    1. MaxUtil

    2. MinMiss

- Adaptive Earliest Deadline Policy

    1. HITbatch

    2. ALLbatch

*MaxUtil* was set to 98%, and *MinMiss* was set to 9% (the same values as those used in [GHSS95]), for all the experiments except for the *Sensitivity Analysis* experiment where *MaxUtil* was varied from 96% to 99% and *MinMiss* was varied from 7% to 9%. *HITbatch* and *ALLbatch* were set to 20 in all the experiments except in the *Sensitivity Analysis* experiment where *HITbatch* was varied between 16 and 24 in step sizes of 4 and *ALLbatch* too was varied from 16 to 24 in step sizes of 4.

As mentioned earlier, there was no statistically significant difference observed between the performance of the corresponding algorithms from the Bayer-Schkolnick and Top-Down class (B-X and TD-X, B-SIX and TD-SIX, B-OPT and TD-OPT). This was due to the height of the B-tree considered, being very small (only three levels deep), which means that there is little difference between the number of exclusive locks held at any time on the scope of an update, in the two cases. These algorithms will henceforth be referred to as X, SIX and OPT respectively.

## 6.4 Low Index Contention

In this experiment, the workload consisted of 80% of searches, 10% of deletes and 10% of inserts (no appends). Since there is predominantly more number of read index operations than update operations, the result is a low index contention environment. For this experiment, Figure 6.1 shows the performance of the various index concurrency control algorithms with respect to MissPercent, SizeFairness and TypeFairness. Considering the MissPercent curves, it is clear that the X algorithm performs very poorly missing more than 90% of the transactions. This is because of the root becoming a bottleneck as pointed out in [GHSS95]. At low arrival rates, B-link performs better than SIX while the opposite holds true at high arrival rates. OPT performs the same as B-link due to the very small number of merges and splits (because of the low number of update index operations).

SIX performs better than B-link at high arrival rates, because of the preference they give to reader transactions by which they are able to complete most of the reader transactions at any arrival rate. Since the percentage of reader transactions is nearly 21% ($= \frac{1}{9}\sum_{i=4}^{12} 0.8^i$, since 80% of operations are searches, and transaction sizes vary from 4 to 12 uniformly), SIX algorithm maintains its MissPercent close to 79% throughout. The preference that SIX gives to readers can be seen clearly from the type fairness graph, where SIX has a TypeFairness of nearly 4.0, while all other algorithms have a type bias factor nearly equal to 1.0. A larger value of TypeFairness indicates that the algorithm gives preference to read operations over write operations.

Note that AED-link performs better than X, SIX and OPT. It also performs better than LAB-link in terms of MissPercent, SizeFairness and SizeFairness.

Generally the bias factor curves will start at 1.0 for an arrival rate corresponding to zero miss percent, increase with increasing arrival rates, indicating increased bias towards

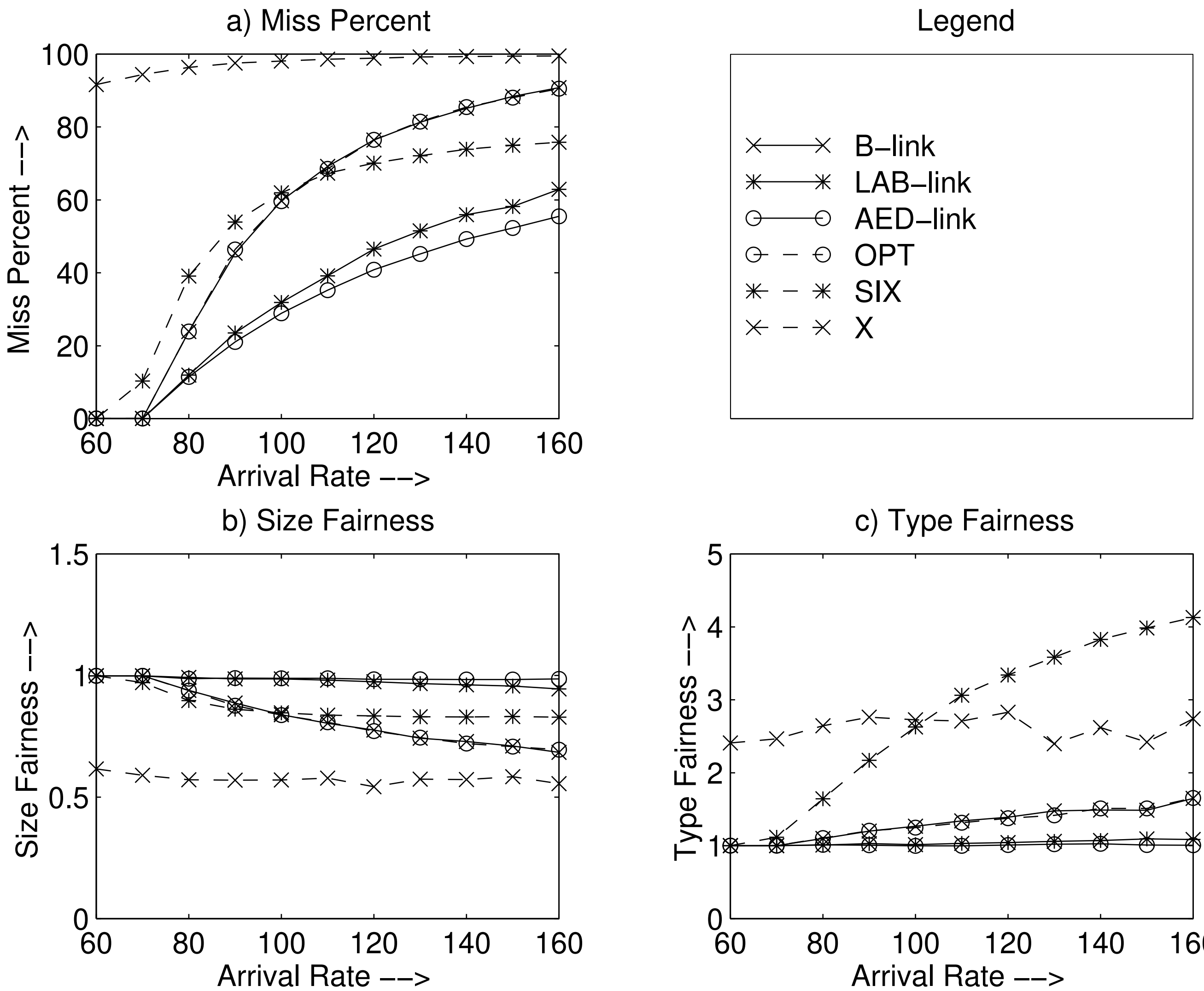



Figure 6.1: Low Index Contention

shorter and more reader-oriented transactions. With even more increase in arrival rates, the miss percent increases to a very high value, which means only a few are completed, and the bias among the few that are completed will be very negligible, making the fairness curves descend till they reach a fairness of 1.0. Note that this is not true for SIX in this experiment, since it finishes almost 21% of the transactions at any arrival rate for the workload in this experiment, since it is able to finish all the readers at any arrival rate (there are nearly 21% of readers).

## 6.5 Moderate Index Contention

This experiment consisted of a workload of 100% inserts resulting in a moderate index contention. For this experiment, Figure 6.2 shows the performance metrics, as a function of the arrival rate for the various algorithms. In this experiment too X algorithms perform badly due to the high index contention existing at the higher levels of the tree. SIX algorithms perform the same as X, since in this experiment, there is no read operation, resulting in a SIX lock effectively acting as an X lock (note that a SIX lock is compatible with a SIX lock of a reader but not with an IX lock of an updater).

OPT and B-link perform nearly the same with respect to MissPercent and the fairness curves, although in the corresponding conventional experiment B-link was found to perform far better than OPT. This is due to the fact that in the real-time case, the number of index concurrency restarts (due to index node conflicts) for OPT is much lower than in the non-real-time case (stated in [GHSS95], but not verified with any statistical measure). We ran the experiment with preemption and without preemption and found out that the number of restarts is relatively less in the with preemption case (refer Table 6.3, the number of transactions processed were 20,000).

| ArrRate | Without Preemption | With Preemption |
|---|---|---|
| 10 | 657 | 654 |
| 20 | 658 | 655 |
| 30 | 656 | 659 |
| 40 | 684 | 628 |
| 50 | 801 | 590 |
| 60 | 847 | 593 |
| 70 | 833 | 550 |
| 80 | 797 | 489 |
| 90 | 740 | 442 |
| 100 | 704 | 422 |

Table 6.3: Number of Restarts Without and With Preemption for OPT

The reason for the lesser number of restarts is the following. In the conventional case, there is no preemption, and hence, lot of first-pass updaters see an unsafe leaf, before

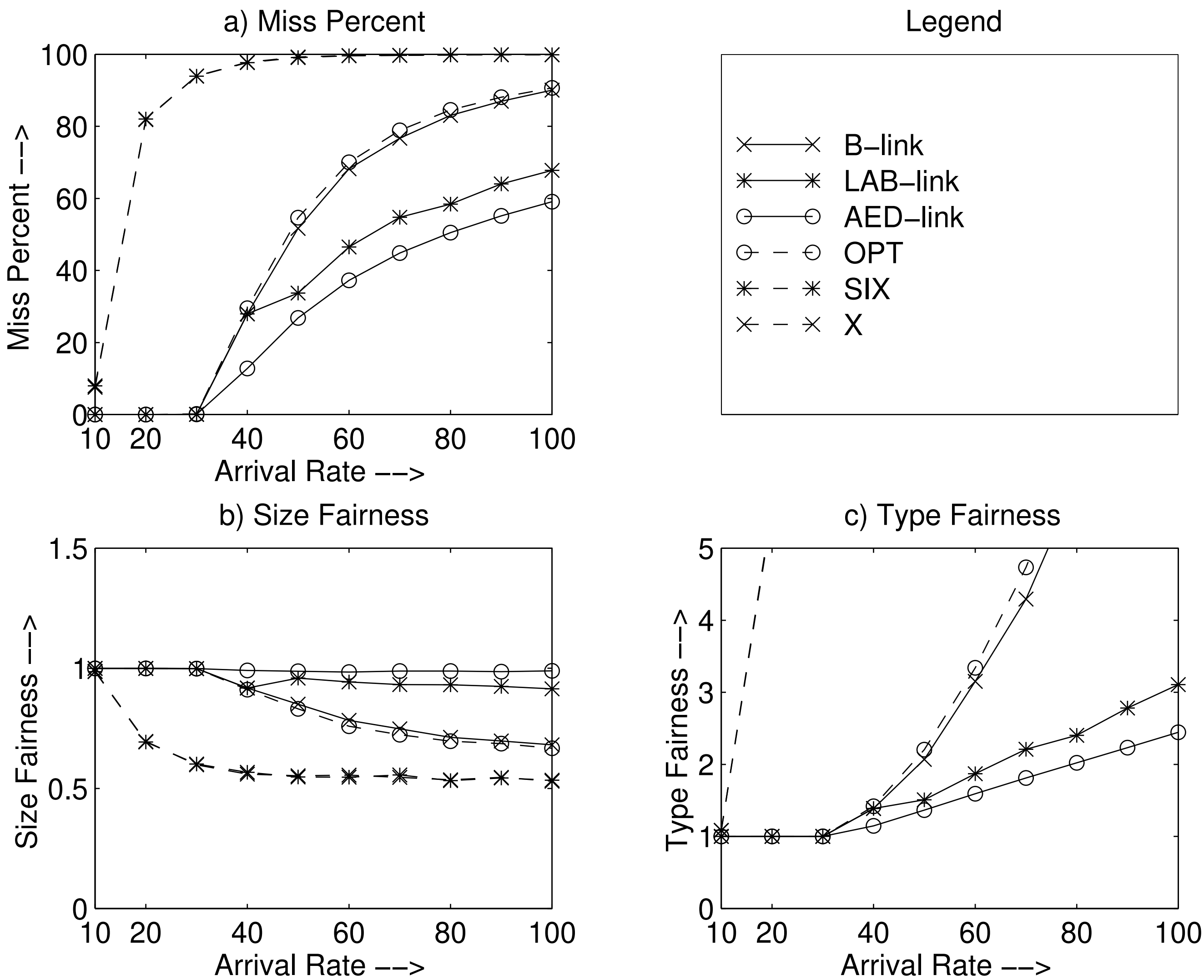



Figure 6.2: Moderate Index Contention

the *first* transaction that saw it as unsafe has completed its second pass and made it safe. In the real-time case, however, there is preemption of index nodes, the second-pass updater completes its pass quickly thereby lessening the number of transactions that see that particular leaf unsafe in their first pass. To verify this, we measured the average time that a leaf is unsafe, for both cases with and without preemption (refer Table 6.4). We found that the average time that a leaf is unsafe is lesser in the with-preemption case.

Considering the other algorithms, AED-link performs the best in this experiment, bettering LAB-link too. Here, load control begins to take effect only after an arrival rate

| ArrRate | Without Preemption | With Preemption |
|---|---|---|
| 10 | 282.69 | 259.67 |
| 20 | 141.78 | 130.98 |
| 30 | 87.83 | 85.38 |
| 40 | 69.61 | 68.33 |
| 50 | 66.08 | 59.05 |
| 60 | 61.82 | 53.61 |
| 70 | 58.66 | 50.28 |
| 80 | 56.22 | 45.92 |
| 90 | 49.83 | 43.33 |
| 100 | 49.76 | 39.91 |

Table 6.4: Average Unsafe Time Without and With Preemption (in sec)

of 40 in LAB-link, while it is taken into effect earlier itself, after an arrival rate of 30 in AED-link. Also, one can see the graceful increase in miss percent in AED-link, while LAB-link is not so graceful. AED-link is better than LAB-link in terms of both fairness curves. One can also note the smooth fairness curves for AED-link, while those for LAB-link are not so. Here the type fairness curve is plotted with the assumption that the fraction of readers completed is 1.0 (in this experiment, the number of readers is 0). Thus effectively the type fairness curve plots the reciprocal of the fraction of updaters completed.

## 6.6 High Index Contention

The workload in this experiment consisted of 25% searches and 75% appends. The appends, because of the key values starting from 300,001 onwards, are directed towards the rightmost part of the B-tree. Since most of the operations are appends, there is very high index contention in that part of the tree. The results of this experiment are shown in Figure 6.3.

The results of the X algorithm are not plotted since they have a larger miss percent than the others at earlier arrival rates itself and it is clear from the previous experiments itself that X performs worse than all the others due to its restrictive locking. SIX performs

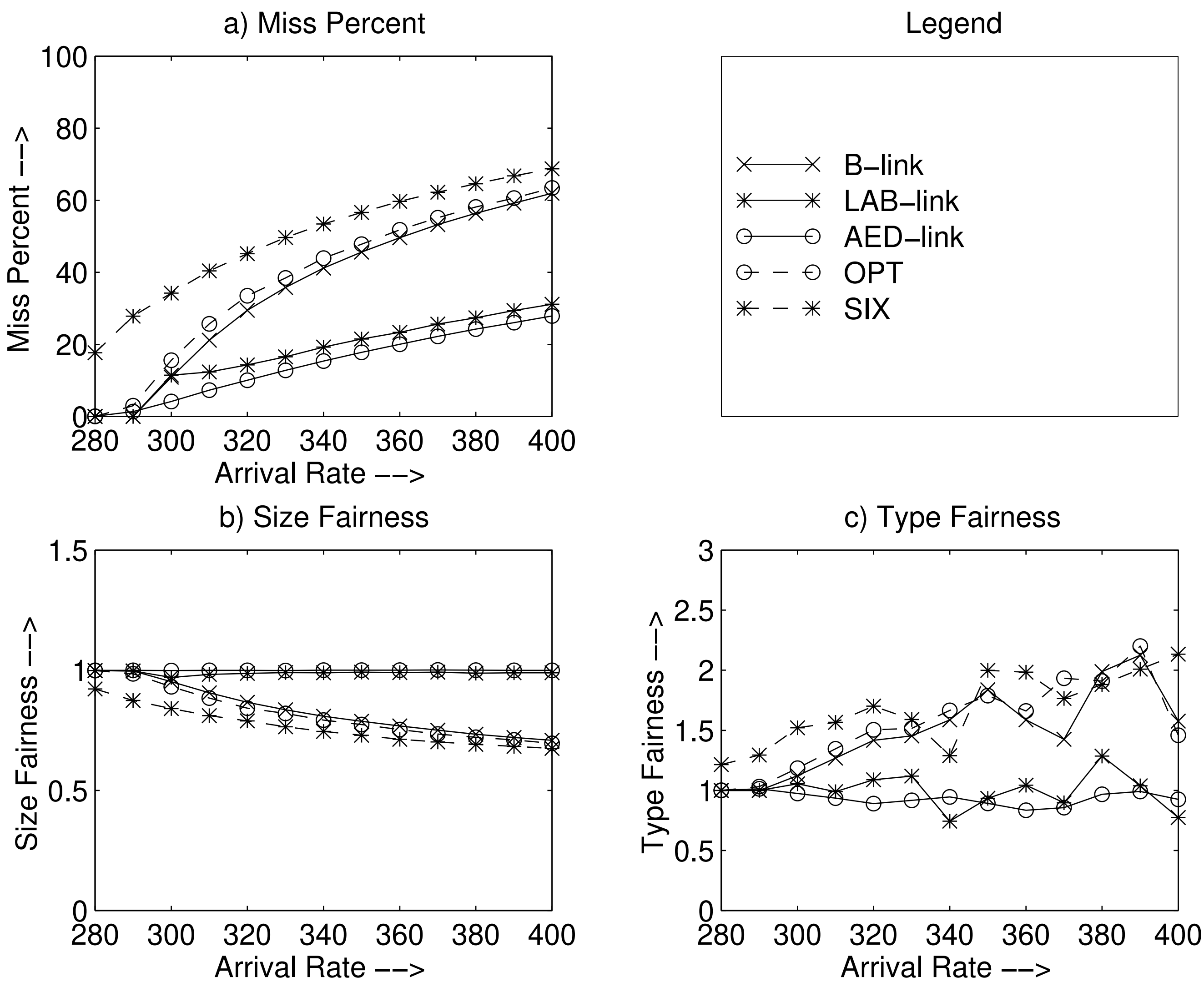



Figure 6.3: High Index Contention

the worst among the remaining, since, again as in the case of the moderate index contention environment, the root becomes a bottleneck, due to more number of updaters than readers. While in the previous experiment, SIX and X had the same performance, here SIX performs better because there are reader transactions in this case (SIX locks of readers are compatible with each other). Similarly, B-link and OPT perform very similarly as in the previous experiment, but the gap between them is more than before. The reason is that here, there is more possibility of restarts due to the high index contention, resulting in OPT performing worse than in the previous experiment. The same reasoning given to the previous experiment works out here too for B-link and OPT. Again, AED-link performs better than LAB-link in terms of miss percent and both the fairness curves. Load control kicks off in LAB-link only after an arrival rate of 300 while in AED-link it starts off after an arrival rate of 290 itself. Here the bottleneck resource is the CPU (for LAB-link).

Regarding the fairness curves, AED-link and LAB-link both perform nearly the same in the SizeFairness curve. The TypeFairness curves are shaky due to the very few number of reader transactions in the workload (only 15 out of a total of 20,000).

## 6.7 No Index Contention

All the previous experiments had a certain amount of index contention. To study the effect of index contention, we repeated the first experiment (low contention experiment), but with a workload of 100% searches thereby removing index contention totally. The performance of the algorithms in this experiment as compared to the low contention experiment give a fair measure of the effect of index contention on the performance of an algorithm. The results obtained are shown in Figure 6.4

The algorithms compared are B-link, LAB-link and AED-link. From the figure, index contention is seen to have a profound impact on the performance of all the algorithms. For example, at an arrival rate of 160 TPS, the performance enhancement due to no contention at index nodes is about 20% for B-link, and even more for LAB-link and AED-link. Once again one can observe that AED-link starts applying load control earlier than LAB-link and outperforms LAB-link for all arrival rates. With respect to the fairness curves, there doesn't seem to be much difference between the performance of AED-link and LAB-link.

To compare the feedback policies of AED-link and LAB-link, we measured the Hit Ratio (i.e. the percent of transactions admitted into the system that are completed within their deadlines). The results are shown in Figure 6.5. From the figure, it is clearly seen that AED-link has a better HitRatio than LAB-link thus deriving its better performance. This is because LAB-link tries to over-estimate the number of transactions that can be completed and hence allows more into the system.

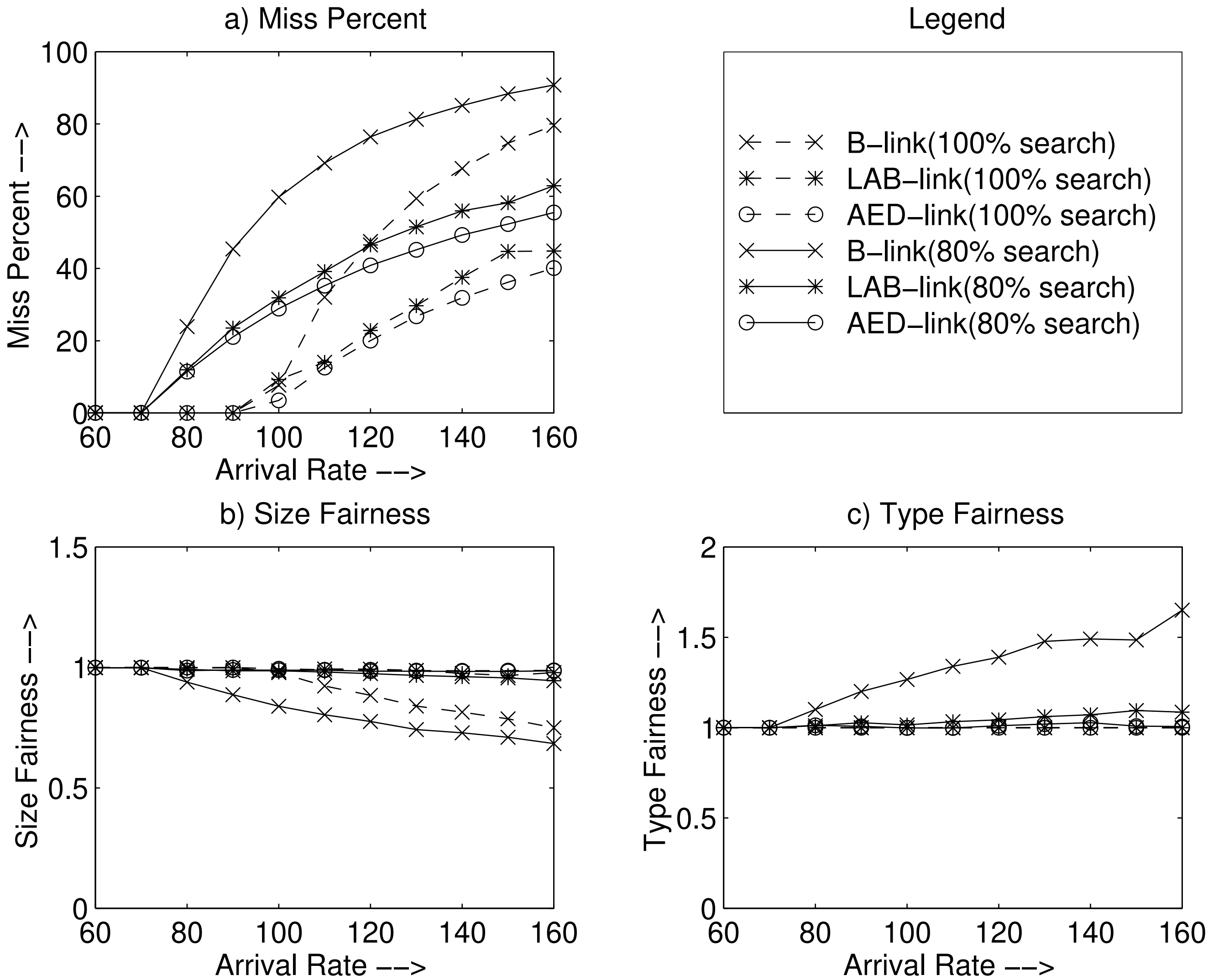



Figure 6.4: No Index Contention

## 6.8 No Resource Contention

This experiment studied the effect of removing the resource contention component. Here the workload was assumed to be the same as the low contention experiment. The resources were assumed to be infinite. Thus, there will be no waiting at the resources. While this assumption is unrealistic of a conventional system, this can be expected of a real-time system, where resources may not be a constraint at all. The only constraint in such a system would be to achieve good performance. For this experiment, the *NumCPUs, NumDisks* and *NumBufs* parameters are set to *infinity*. Thus all index nodes will stay in the buffer

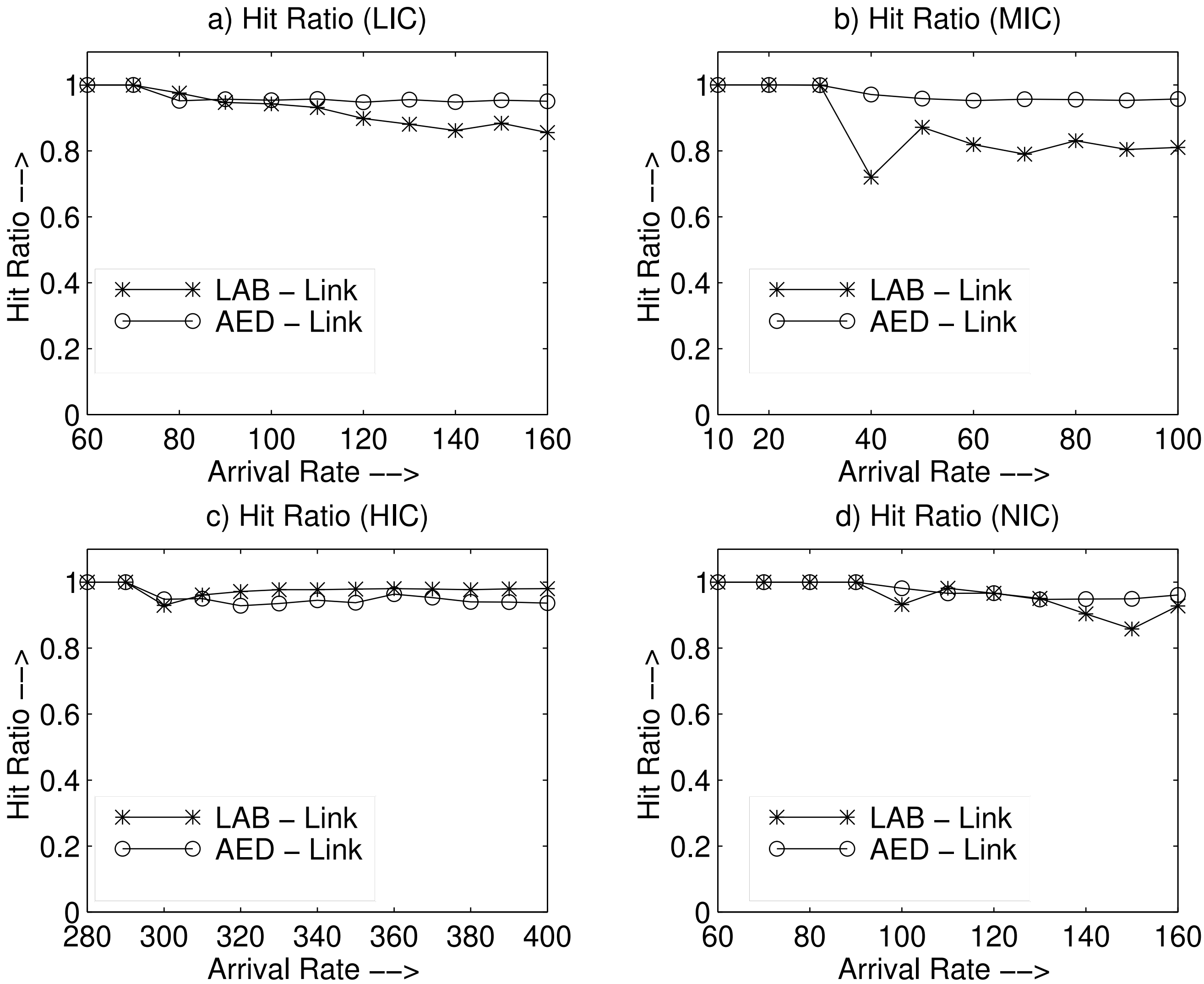



Figure 6.5: Hit Ratio for HITGroup (AED-link without undos)

pool, and paging to disk is not required. Also, data contention in our experiments was minimal, since the number of transaction restarts due to key-value conflicts didn't exceed 30 out of the 20,000 transactions processed. Thus this experiment involves only index contention. The results of this experiment are shown in Table 6.5.

From the table, it is clear that B-link and OPT perform well, finishing almost all of the transactions even at an arrival rate of 20000 TPS. SIX on the other hand manages to finish only nearly 21% of the transactions. This directly relates to the fact that the percentage of readers in the input is nearly 21%. SIX manages to complete all the readers, while it does

| ArrRate | B-link | OPT | SIX | AED-SIX |
|---|---|---|---|---|
| 1000 | 0.02 | 0.02 | 0.02 | 0.02 |
| 2000 | 0.025 | 0.025 | 0.02 | 0.02 |
| 3000 | 0.035 | 0.035 | 25.735 | 11.81 |
| 4000 | 0.040 | 0.040 | 59.355 | 27.77 |
| 5000 | 0.05 | 0.05 | 70.21 | 40.62 |
| 10000 | 0.06 | 0.07 | 79.23 | 60.44 |
| 15000 | 0.09 | 0.09 | 79.70 | 67.38 |
| 20000 | 0.18 | 0.14 | 79.83 | 70.55 |

Table 6.5: No Resource Contention

not do almost all of the updaters. AED-SIX, which is AED priority assignment applied to SIX, does better than SIX. LAB-link is the same as B-link for this experiment, since there is no resource contention, and hence *MaxUtil* is always zero. For that matter, Load Adaptive Policy wont improve the performance of any index concurrency control algorithm, in this experiment. LAB-link uses admission control based on the system resource utilization parameters. The resource utilization can be measured in the system. On the other hand, LAB-link cannot capture the data or index contention in the system while AED-link can do the same since it is measuring an end-user performance metric (miss percent at the output of the RTDBS) which is composed of all the contention components. Thus AED is able to improve the performance of all the algorithms. The improvement obtained in B-link and OPT is negligible since they themselves miss only a very few. There is appreciable improvement obtained in SIX due to AED. This is an example of a situation where the Load Adaptive Policy doesn't work, while AED works.

## 6.9 Effect of AED and LAP on other ICC algorithms

This experiment investigated the relative effect of AED and LAP on the other index concurrency control algorithms. The results of the experiment are shown in Figure 6.6. The workload we consider is similar to the one in the low contention experiment. AED and LAP are found to improve the performance of all the algorithms appreciably although

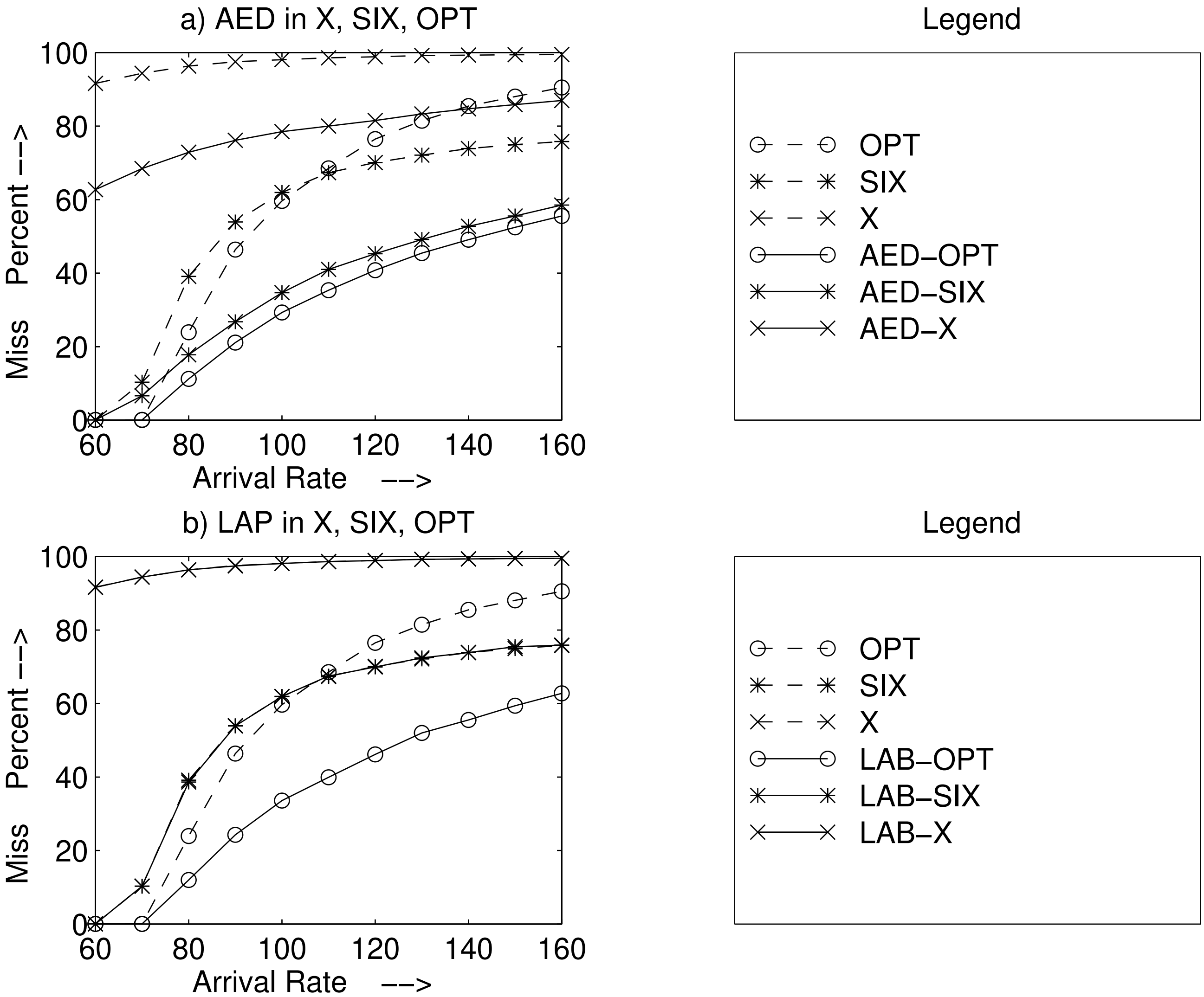



Figure 6.6: Comparison of AED & LAP for X, SIX, OPT

AED improves the performance of the ICC algorithms more than LAP does. In fact, LAP improves the performance of only OPT, while AED improves the performance of X, SIX and OPT. The corresponding algorithms obtained by applying LAP are termed LAB-X, LAB-SIX and LAB-OPT. Note that while AED is able to significantly improve the performance of all the algorithms, LAP is able to improve the performance of OPT only since, OPT is the only one that uses its resources to near full capacity. SIX and X don't use their resources to full capacity, but have a very high index contention component, which is not identified by LAP due to its measurement of only system related parameters.

This illustrates another case where load control through AED works while that through LAP doesn't.

## 6.10 Sensitivity Analysis

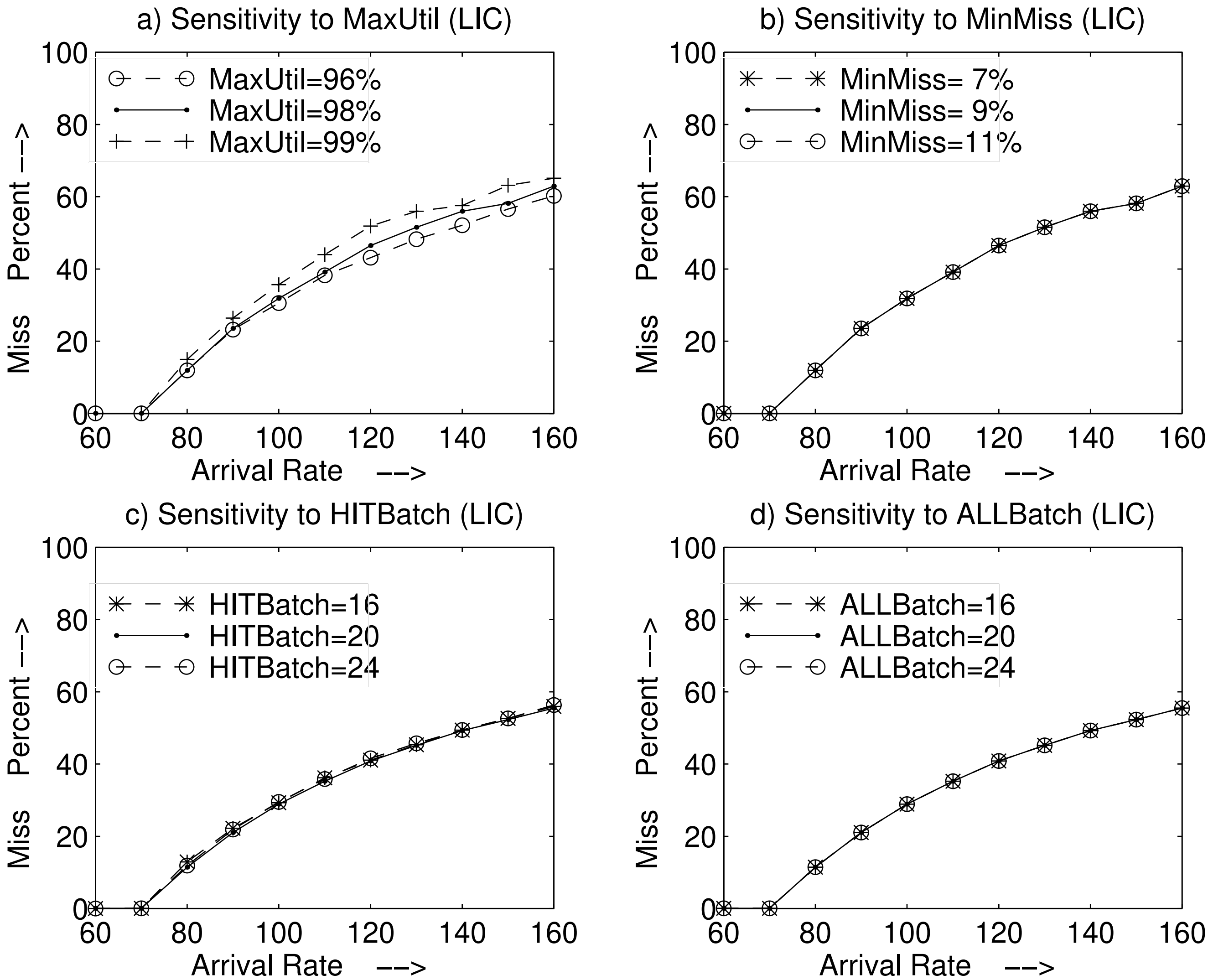



Figure 6.7: Sensitivity Analysis

In this experiment, we analyze the sensitivity of the two load control policies to their parameters. The workload that we consider is the same as the low contention experiment. Load Adaptive Policy uses two parameters *MaxUtil* and *MinMiss* while AED-link uses *HITbatch* and *ALLbatch*. We varied MaxUtil from 96% to 99% for LAB-link and varied

HITbatch from 16 to 24 in steps of 4 for AED-link. Also we varied MinMiss from 7% to 11% in steps of 2% for LAB-link and ALLbatch from 16 to 24 in steps of 4 for AED-link. The results are shown in Figure 6.7. The upper graph shows the effect of variation of *MaxUtil* on LAB-link, while the lower one shows the effect of variation of *HITbatch* on AED-link.

From the figure, it is clearly seen that LAB-link is sensitive to the value of *MaxUtil*. There is a difference of nearly 4% in miss percent between *MaxUtil* values of 96% and 98%. On the other hand AED-link behaves the same for all *HITbatch* range of values considered. This clearly shows the stability of AED-link as opposed to LAB-link. There is no difference in the results for different values of *MinMiss* for LAB-link due to the first non-zero miss percent for LAB-Link being above the maxiumum value of *MinMiss* (11%) considered in this experiment. Also, AED-link is stable to changes in *ALLbatch*.

## 6.11 Different Real-Time Situations

We experimented with different real-time situations by varying the slack factor of the transaction. We assigned values of 2.0 and 8.0 for SF, and also tried out a uniform distribution of slack factor with LSF = 2.0 and HSF = 6.0. Figure 6.8 shows the miss percent curves of the various algorithms for varying slack factors with all other parameters assuming the same values as the low index contention experiment.

At low arrival rates, increasing the slack factor improves the performance of all algorithms, since a transaction now gets more time to finish and transactions which were originally missing their deadlines very narrowly will be able to complete now. However, at high arrival rates, since transactions are not discarded and new transactions keep coming very soon, the system becomes highly loaded and the performance of all algorithms except SIX, AED-link and LAB-link comes down. SIX doesn't perform worse with increasing slack factor because of its preference to reader transactions which makes it complete almost all the readers whatever the arrival rate or slack factor be. Also LAB-link is able to perform better with increasing slack factor, this is because, it now admits only lesser transactions into the system and finishes a greater percent of those admitted. Anyway,

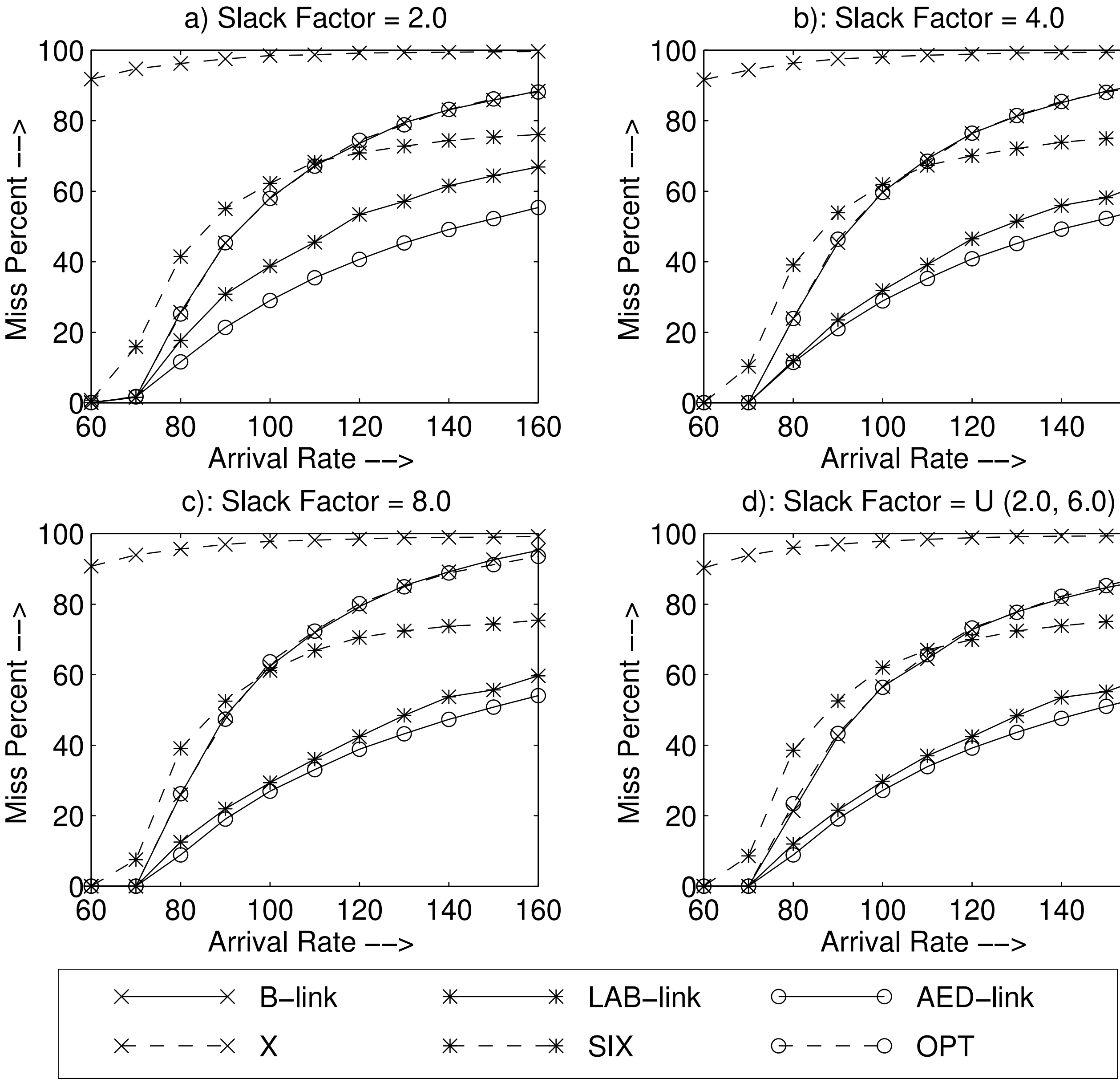



Figure 6.8: Different Slack Factors

LAB-link performs worse than AED-link in all the cases. AED-link is able to perform nearly the same for all slack values. This shows that it chooses exactly the set of transactions that can be completed and nothing more whatever be the real-time situation (slack factor). The curves for the uniformly distributed slack factor (between 2.0 and 6.0) are similar to the curves for the constant slack factor case where SF = 4.0 except that the performance is slightly better in the former case.

This is because the resource time estimated is that of the largest transaction thereby most of the small transactions have effectively a greater slack factor than in the SF = 4.0 case, thus giving them more time to finish and completing more of them at the cost of a few larger transactions.

## 6.12 Index Contention With Undo

We now move on to considering the performance impact of modeling the *undos* of the index actions of aborted transactions. In [GHSS95], 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 6.9, 6.10, 6.11 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 (Figure 6.11) but in the LIC and MIC environments (Figures 6.9, 6.10), the undos at most have only a *marginal* adverse impact on performance. Note that this marginal impact phenomenon is *not limited to light loads* – it also occurs for high miss

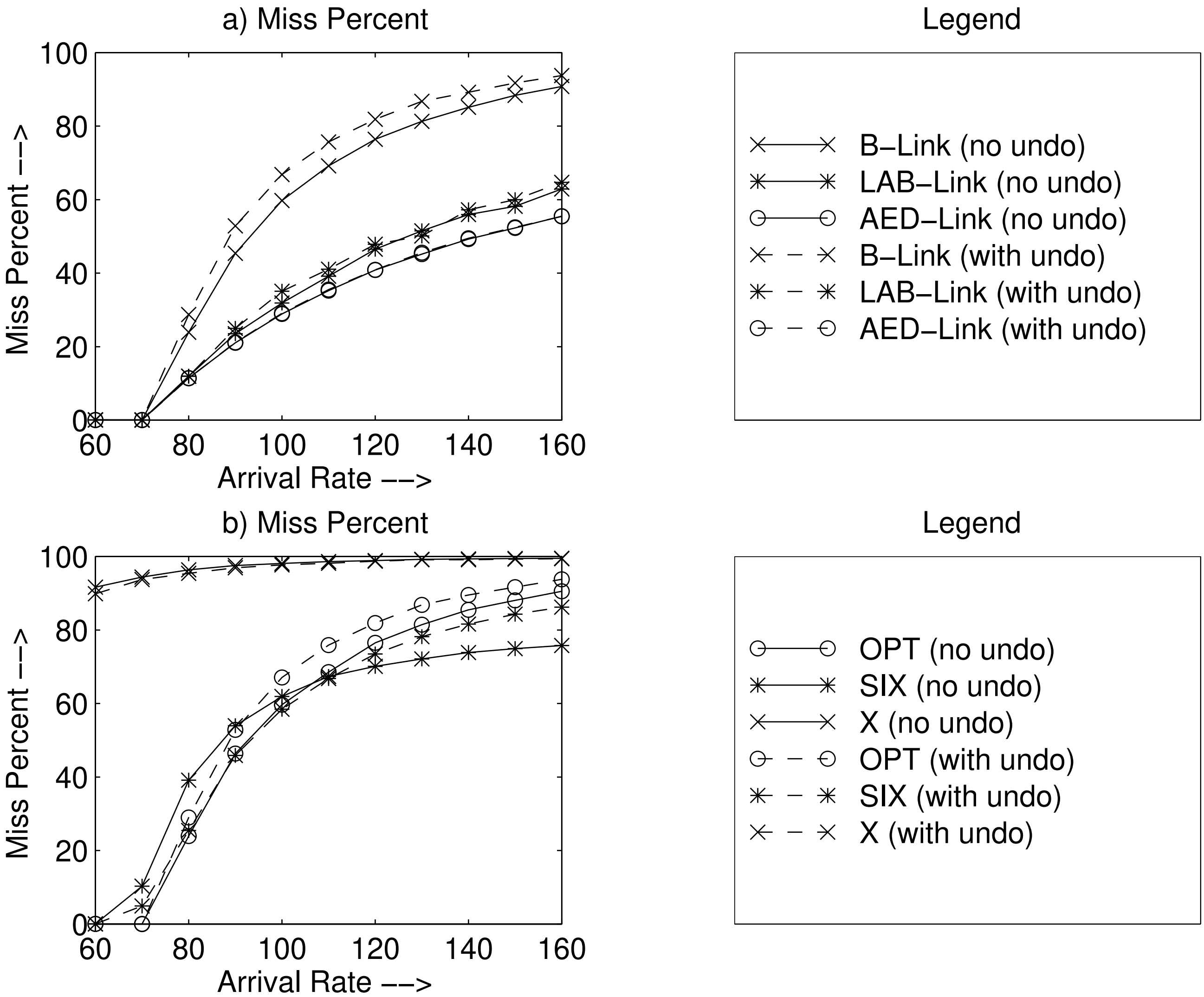



Figure 6.9: Low Index Contention (Without and With Undo)

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 6.9b). 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

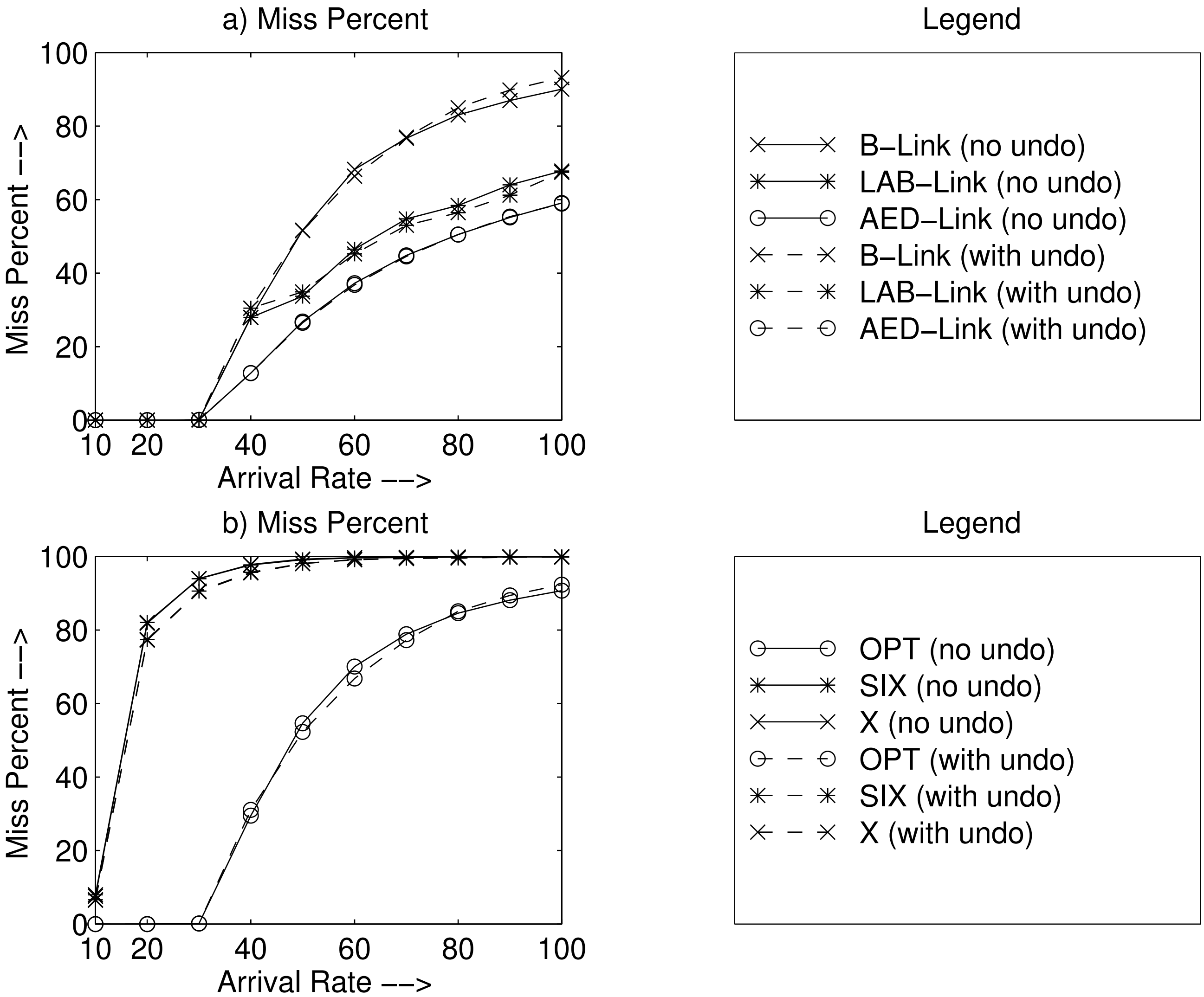



Figure 6.10: Moderate Index Contention (Without and With Undo)

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 is typically about *90 percent*, as against a hit ratio of only around 50 percent for the normal operations

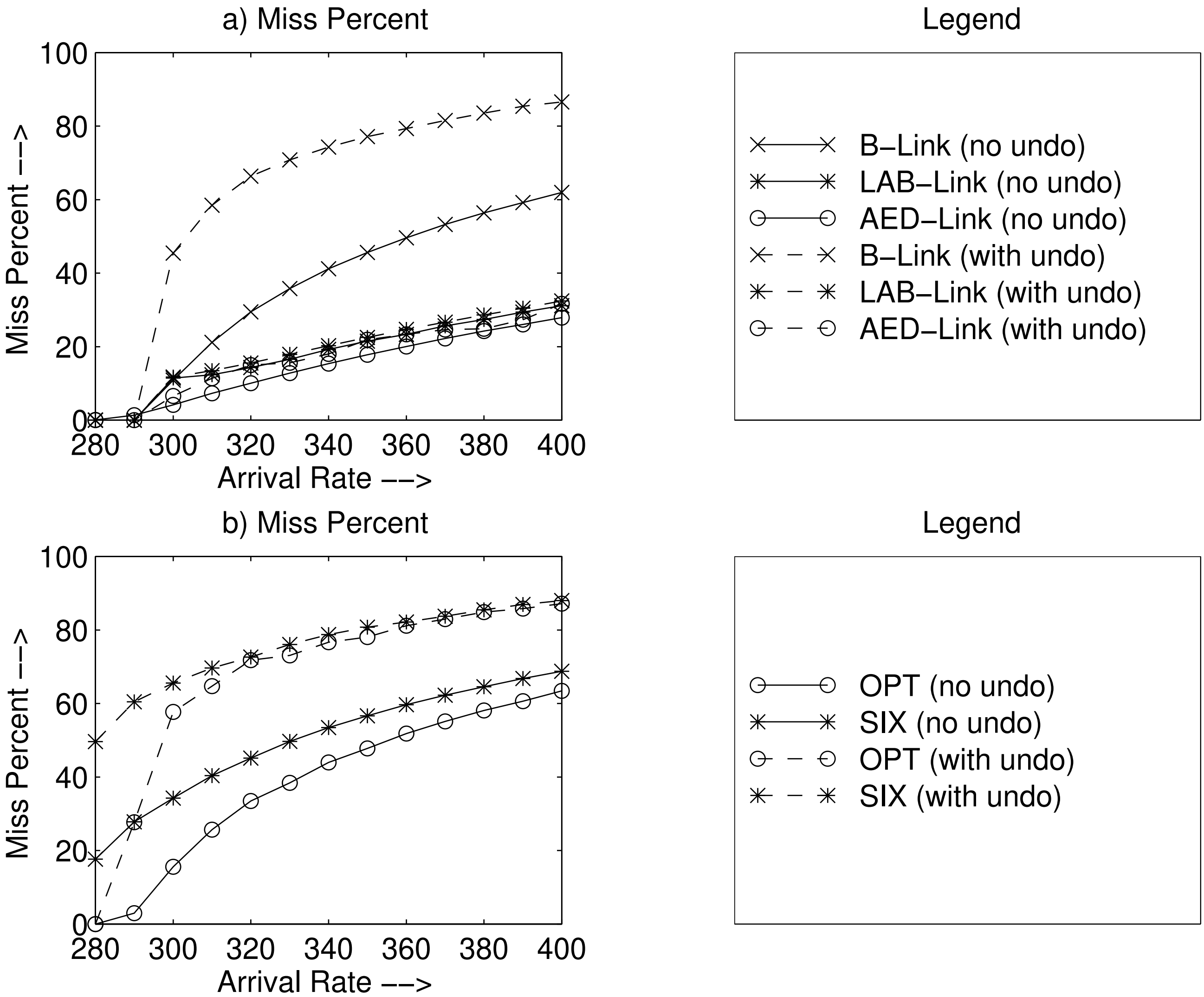



Figure 6.11: High Index Contention (Without and With Undo)

(Figures 6.12c-d). 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 6.11a, 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, 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 (refer Figures 6.12e-f. Here Leaf Splits Percent refers to the percentage of new leaf nodes created). 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 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.

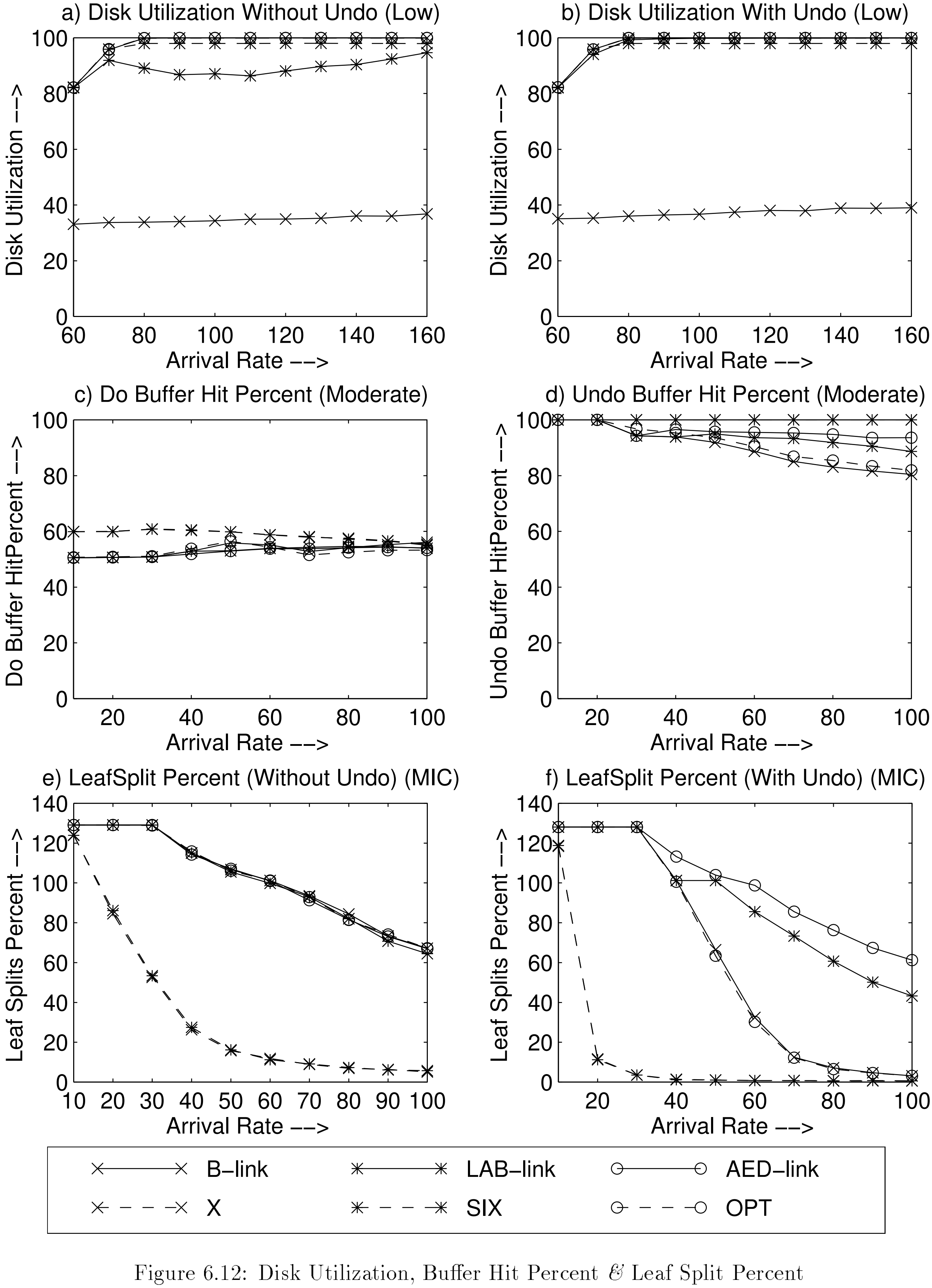



Figure 6.12: Disk Utilization, Buffer Hit Percent & Leaf Split Percent

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.

## 6.13 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 Low Index Contention 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 experiment.

For this experiment, the MissPercent performance of all the ICC protocols are shown in Figure 6.13a. 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 6.13e, 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 experiment, LAB-link

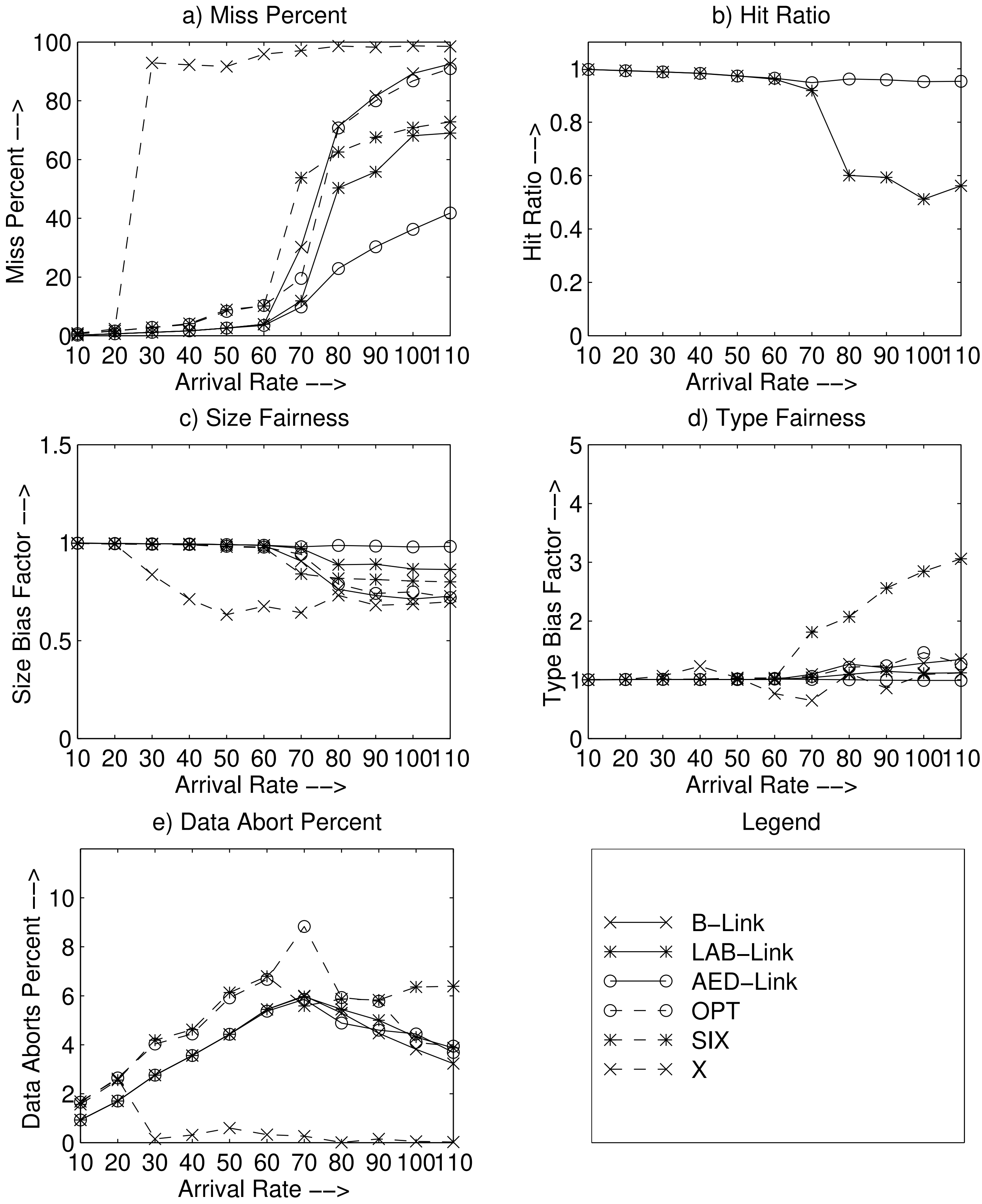



Figure 6.13: Range Queries (LIC)

by virtue of being cognizant of only resource contention fails to apply sufficiently strict admission control. This is brought out clearly in Figure 6.13b, which shows the hit ratios in the HIT 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 HIT 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 6.13d.

## 6.14 Summary

This chapter discussed the experiments and the results obtained. AED-link was found to perform the best in terms of all the performance metrics in all the cases. We showed how sensitive LAB-link is to the system resource utilization parameters. We also experimented with different real-time situations and obtained similar results. We also observed the effect of modeling undos of index actions of transactions. We applied AED and LAP on the other ICC algorithms and found AED to perform better in all the cases while load control through LAP didn't even come into effect for some algorithms although the algorithms were performing badly. This is because of LAP considering only the resource utilization and not able to quantitatively measure the index contention thus failing in cases where there is less resource contention and more of index contention. We summarize our work in the next chapter and conclude with the scope for future work.

# Chapter 7

# Conclusion

Index concurrency control algorithms for the real-time domain have received little attention. To the best of our knowledge a performance study has been done till date only in [GHSS95]. There are a lot of issues to be addressed. We have attempted to address a few in this study. In the following sections we summarize our contributions and discuss avenues for further work.

## 7.1 Summary of the Work

This study 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, high and no contention. 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 [Moha90] 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 [GHSS95]. 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 transaction type. In contrast, all the classical protocols showed bias in favour of smaller-sized transactions and in favour 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 [GHSS95], 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.

We also tried out different real-time workloads by considering the slack factor of the transaction as a uniformly distributed random variable, and obtained results similar to that obtained in the baseline case.

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 blink 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.

## 7.2 Future Work

In this section we outline areas of index concurrency control in the firm RTDBS domain that can be addressed. One issue deals with how to perform load control. Both AED and Load Adaptive Policy implemented load control through a feedback process. Since the basic question is to separate the arriving transactions into groups (not-admitted and admitted groups in Load Adaptive Policy, HIT and MISS group in AED), an Artificial Neural Network (ANN) based approach to load control seems to be a good alternative. We had incorporated data concurrency control through 2PL-HP (a real-time variant of 2PL, a pessimistic concurrency protocol). Earlier studies [HCL92] had shown optimistic protocols to perform better than locking protocols in a firm RTDBS. Since the data contention in our experiments was low, we didn't expect any significant change in the results due to the data concurrency protocol. However, one positive feature of the optimistic protocols is that the effects of the transaction that misses its deadline and is hence aborted, need not be undone since the updates are delayed till commit time. We conclude with the view that although this study has investigated issues related to load control, different real-time workloads, etc. in the context of real-time ICC, a lot of issues still remain to be addressed.

# Bibliography


[ACL87] R. Agrawal, M. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications", *ACM Transactions on Database Systems*, December 1987.

[AG88] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation", *Proceedings of the 14th Intl. Conf. on Very Large Database Systems*, August 1988.

[AG89] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data", *Proceedings of the 15th Intl. Conf. on Very Large Database Systems*, August 1989.

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

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

[BB94] A. Bestavros and S. Braoudakis, "Timeliness via Speculation for Real-Time Databases", *Proceedings of IEEE Real-Time Systems Symposium*, December 1994.

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

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

[Esw76] K. Eswaran et al., "The Notions of Consistency and Predicate Locks in a Database System", *Communications of ACM*, November 1976.

[GHSS95] B. Goyal, J. Haritsa, S. Seshadri and V. Srinivasan, "Index Concurrency Control in Firm Real-Time DBMS", *Proceedings of the 21st Intl. Conf. on Very Large Database Systems*, Zurich, Switzerland, 1995.

[Gray79] J. Gray, "Notes on Database Operating Systems", *Operating Systems: An Advanced Course*, eds. R. Bayer, R. Graham and G. Seegmuller, Springer-Verlag, 1979.

[Hard84] T. Haerder, "Observations on Optimistic Concurrency Control Schemes", *Information Systems*, 9(2), 1984.

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

[HLC91] J. Haritsa, M. Livny and M. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems", *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

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

[HSRT91] J. Huang, J. Stankovic, K. Ramamritham and D.Towsley, "On using priority inheritance in real-time databases", *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

[JS89] T. Johnson and D. Shasha, "Utilization of B-trees with Inserts, Deletes and Searches", *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, April 1990.

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

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

[LS86] V. Lanin and D. Shasha, "A Symmetric Concurrent B-tree Algorithm", *Proceedings of Fall Joint Computer Conference*, 1986.

[Liu73] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, January 1973.

[LS92] D. Lomet and B. Salzberg, "Access method concurrency with recovery", *Proceedings of ACM SIGMOD Conference*, June 1992.

[LY81] P. Lehman and S. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, 6(4):650-670, December 1981.

[Moha90] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes", *Proceedings of the 16th Intl. Conf. on Very Large Database Systems*, September 1990

[ML89] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-Ahead Logging", *IBM Research Report RJ6846*, 1989.

[MR85] Y. Mond and Y. Raz, "Concurrency Control in $B^+$-tree Databases Using Preparatory Operations", *Proceedings of the 11th Intl. Conf. on Very Large Database Systems*, September 1985.

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

[PLC92] H. Pang, M. Livny and M. Carey, "Transaction Scheduling in Multiclass Real-Time Database Systems", *Proceedings of IEEE Real-Time Systems Symposium*, December 1992.

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

[SG88] D. Shasha and N. Goodman, "Concurrent search structure algorithms", *ACM Transactions 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.

[SC91] V. Srinivasan and M. Carey, "Performance of B-Tree Concurrency Control Algorithms", *Proceedings of ACM SIGMOD Conference*, May 1991.

[Sri92] V. Srinivasan, "On-Line Processing In Large-Scale Transaction Systems", *Ph. D. Thesis*, Dept. of Computer Science, Univ. of Wisconsin (Madison), January 1992.

[SZ88] J. Stankovic and W. Zhao, "On real-time transactions", *ACM SIGMOD Record*, 17(1), March.

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

[Triv82] K. Trivedi, *Probability & Statistics with Reliability, Queuing and Computer Science Applications*, Prentice-Hall Inc., 1982.