Real-Time Index Concurrency Control

Jayant R. Haritsa, S. Seshadri

Abstract— **Real-time database systems are expected to rely heavily on indexes to speed up data access and thereby help more transactions meet their deadlines. Accordingly, high-performance index concurrency control (ICC) protocols are required to prevent contention for the index from becoming a bottleneck. In this paper, we develop real-time variants of a representative set of classical B-tree ICC protocols and, using a detailed simulation model, compare their performance for real-time transactions with firm deadlines. We also present and evaluate a new real-time ICC protocol called GUARD-link that augments the classical B-link protocol with a feedback-based admission control mechanism. Both point and range queries, as well as the undos of the index actions of aborted transactions are included in the scope of our study. The performance metrics used in evaluating the ICC protocols are the percentage of transactions that miss their deadlines and the fairness with respect to transaction type and size.**

Our experimental results show that the performance characteristics of the real-time version of an ICC protocol could be significantly different from the performance of the same protocol in a conventional (non-real-time) database system. In particular, B-link protocols, which are reputed to provide the best overall performance in conventional database systems, perform poorly under heavy real-time loads. The new GUARD-link protocol, however, although based on the B-link approach, delivers the best performance (with respect to all performance metrics) for a variety of real-time transaction workloads, by virtue of its admission control mechanism. In fact, GUARD-link provides close to ideal fairness in most environments. These and other results presented here represent the first work in the area of real-time index concurrency control.

Keywords— **Real–Time Database, Index Concurrency Control, B-tree, B-link tree.**

I. Introduction

REAL-TIME database systems (RTDBS) cater to data-intensive applications that are faced with timing constraints, typically in the form of transaction completion deadlines. Such applications include electronic commerce, stock trading, mobile computing, network management and factory automation. For example, in the Flash Auction at `http://www.firstauction.com`, a bid is valid only if registered in the database within five minutes of the registry of the previous bid.

The ability of an RTDBS to meet transaction timing constraints depends on several factors such as the database system architecture, the processor and disk speeds, etc. For a given system configuration, however, the primary real-time performance determinants are the policies used for *scheduling* transactions at the system resources. These policies determine *when* service is provided to a transaction, and therefore directly impact whether or not a transaction completes before its deadline.

Research on developing real-time database scheduling policies has been underway for close to a decade now (see [5], [34] for surveys). In particular, high-performance policies for priority assignment, processor and disk scheduling, memory management, data concurrency control, distributed commit processing, etc., have been developed. A notable lacuna in this research is that there has been *no study* of the role of *index concurrency control* in determining real-time performance. This lacuna is rather surprising since it appears highly reasonable to expect that RTDBS will make extensive use of indexes to quickly access the base data and thereby help more transactions meet their deadlines. For example, it is estimated that the rapid spread of cellular communication technology will soon result in database systems having to support environments with a large number of highly mobile users [14]. Due to the users mobility, their interactions with the database system have to be processed in real-time and indexes will be essential for quickly processing frequent operations such as location updates ("I am here") and location queries ("Where is X").

With heavy usage of the index, the contention among transactions concurrently using the index may *itself* form the primary performance bottleneck, rather than other forms of resource contention. In particular, note that while contention for the physical resources can always be reduced by purchasing more and/or faster hardware, there exists no equally simple mechanism to reduce index contention since the base data, and hence indexes, cannot be "manufactured". Similarly, even in environments where the contention on base data is largely absent due to transactions accessing mostly disjoint sets of data, index contention may continue to pose performance problems since a common access structure is used to access this base data. Therefore, in this sense, *index contention is a more "fundamental" determinant of RTDBS performance as compared to other shared system resources.*

For the above situation, employing an "off-the-shelf" index concurrency control (ICC) protocol that has not been specifically designed for the real-time environment would only aggravate the problem and result in many more transactions missing their deadlines. Therefore, there is a clear need for developing ICC protocols that are *tuned* to the objectives of the real-time domain. We address this issue here.

A. Real-Time Index Concurrency Control

While a large variety of index structures have been proposed in the literature, commercial database systems typically use **B-tree** indexing [4] as the preferred access method. In particular, they implement the B$^+$ variant in which all data key values are stored at the leaf nodes of

J. R. Haritsa is with the Database Systems Lab, SERC, Indian Institute of Science, Bangalore 560012, INDIA. E-mail: haritsa@dsl.serc.iisc.ernet.in.

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

the index. In the remainder of this paper, our usage of the term B-tree refers to this variant.

A number of protocols have been proposed in the literature for maintaining the consistency of the B-tree index in the face of concurrent transaction accesses (e.g. [2], [3], [16], [17], [18], [20], [21], [23], [30], [28]). The performance of a representative set of these protocols was profiled in [15], [31] and their results indicate that **B-link** protocols [18] are the best choice. These studies were conducted in the context of a *conventional DBMS* with transaction throughput being the performance metric. Therefore, their results are not directly applicable to RTDBS where performance is measured in terms of the ability of the system to complete transactions before their deadlines expire, necessitating a fresh investigation of this issue.

There are two major issues that need to be explored with regard to real-time index concurrency control: First, how do we adapt the conventional ICC protocols to the real-time domain? Second, how do these real-time variants compare in their performance? In this paper, we address these questions in the context of real-time applications with "firm-deadlines" [10] – for such applications, completing a transaction after its has expired is of no utility and may even be harmful.[1] Therefore, transactions that miss their deadlines are "killed" – that is, immediately aborted and permanently discarded from the RTDBS without being executed to completion. Accordingly, the performance metric is *KillPercent*, the steady-state percentage of killed transactions.[2]

B. Contributions

For the above environment, we develop real-time variants of the set of classical ICC protocols considered in [15], [31] and, using a detailed RTDBS simulation model, evaluate their performance for a variety of transaction workloads and index contention environments. We also present and evaluate a new real-time ICC protocol called **GUARD-link** that augments the real-time variant of the B-link protocol with a feedback-based *admission control* mechanism called GUARD. To the best of our knowledge, our results represent the *first work* in the area of real-time index concurrency control.

A feature of our study is the level of detail in the index processing model – it includes multiple index action transactions, range operations and undos of aborted index operations. In contrast, in [15], [31] each transaction consisted of only a *single point index operation.* Further, apart from KillPercent, an additional performance-related issue that we consider is the *fairness* of the protocols, in terms of bias for or against a particular class of transactions, an issue that has largely been ignored in most RTDBS research. Our motivation for including these additional complexities are described below.

B.1 Transaction and Index Model

The KillPercent performance metric applies to *entire* transactions, not to individual index actions. Therefore, in our model each user transaction consists of multiple index actions and mechanisms for ensuring transaction serializability are implemented. Further, both *point* (single key) index operations and *range* (multiple key) index operations are supported since range queries form an important component of index usage.

In an RTDBS, transactions are usually assigned priorities to reflect to the system resources each transaction's urgency relative to other concurrently executing transactions – the Earliest Deadline policy [19], for example, assigns priorities in the order of transaction deadlines. When these priorities are used in resolving *data conflicts*, a lower priority transaction is usually forcibly *aborted*[3] if it happens to possess the data item required by a higher priority transaction.

In addition to the aborts caused by the priority resolution of data conflicts, transaction aborts also arise in a firm-deadline RTDBS due to transactions missing their deadlines and therefore being killed. Taken as a whole, therefore, transaction aborts are *much more common* in RTDBS than in conventional DBMS. (In a conventional DBMS, aborts are usually resorted to only to resolve data deadlocks, and deadlocks occur very infrequently in practice [33].)

Since aborts are common in an RTDBS and since the index actions performed by an aborted transaction need to be *undone*, the processing of these "undos" may have a significant performance impact. Therefore, we explicitly model the undos of the index actions of aborted transactions.

B.2 Protocol Fairness

Characterizing the fairness performance of RTDBS protocols, in terms of bias for or against a particular class of transactions, is important for the following reasons:

1. Fairness is inherently important in RTDBS that serve "public applications" such as stock markets and telecommunications wherein it is essential to treat all users alike.
2. Evaluating fairness helps to identify situations wherein one protocol outperforms another not because it is designed better, but because it chooses to *selectively* complete "easier" transactions (for example, short transactions or read-only transactions). This kind of "cheating" is particularly possible in the RTDBS domain since, as mentioned before, the performance of these systems is typically measured only in terms of the number of transactions that are successfully completed, ignoring the *characteristics* of the completed transactions.

[1] Alternative kinds of real-time applications are "hard-deadline" applications, where *all* transaction deadlines have to be met, and "soft-deadline" applications, where even transactions that miss their deadline are required to be executed to completion. We choose to selectively investigate firm-deadline applications here because database systems for efficiently supporting hard deadline applications are considered to be infeasible [32], while soft deadline applications are difficult to characterize since they have multiple real-time metrics – number of killed transactions and the completion tardiness of late transactions.

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

[3] The aborted transaction may subsequently be restarted by the system.

To address the above observations, we consider two fairness metrics in our study: *SizeFairness* and *TypeFairness*, which measure the bias exhibited by a protocol based on transaction size and transaction type, respectively.

C. Organization

The remainder of this paper is organized as follows: An overview of the major B-tree ICC protocols proposed in the literature is presented in Section II. The issues involved in incorporating real-time priorities in ICC protocols and guaranteeing transaction serializability are discussed in Section III. GUARD-link, our new admission-control-based ICC protocol is introduced in Section IV. The performance model is described in Section V, and the results of the simulation experiments are highlighted in Section VI. Finally, in Section VII, we summarize the conclusions of the study.

II. B-tree ICC Protocols

In this section, we present an overview of the major ICC protocols that have been presented in the literature. We assume, in the following discussion, that the reader is familiar with the basic features and operations of B-tree index structures [4], [31].

The transactional operations associated with B-trees are *search*, *insert*, *delete* and *append* of key values. Search, insert and delete are the traditional index operations. The append operation is a special case of the insert operation wherein the inserted key is larger than the maximum key value currently in the index. In this case, the new key will always be inserted in the rightmost leaf node of the B-tree. We make the distinction between appends and generic inserts since, as shown later in the paper, the performance implications when appends are frequent[4] are quite different since the rightmost leaf node of the index effectively becomes a "hot-spot". Finally, note that search operations arise out of transaction reads while the insert, delete and append operations arise out of transaction updates.

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

B-tree ICC protocols maintain index consistency in the face of concurrent transaction accesses and updates. This is achieved through the use of locks on index nodes. Index node locks are typically implemented in commercial DBMS using *latches*, which are "fast locks" [23]. This optimization is taken into account in our performance study, as discussed later in Section III. An important aspect of latches is that deadlocks involving latches are not detected and ICC protocols have to therefore ensure that latch deadlocks can never occur.

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

The index lock modes discussed in this paper and their compatibility relationships are given in Table I. In this table, IS, IX, SIX and X are the standard "intention share","intention exclusive", "share and intention exclusive" and "exclusive" locks, respectively [7].

TABLE I
Index Node Lock Compatibility Table

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

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

Another technique used by the ICC protocols for their descent from the root to the leaf is *optimistic lock-coupling*. An operation is said to optimistically lock couple if, regardless of safety, the lock at each level of the tree is released as soon as the appropriate child has been locked.

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

A. Bayer-Schkolnick Protocols

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

B. Top-Down protocols

In the Top-Down class of protocols (e.g. [17], [24]), readers use the same locking strategy as that of the Bayer-Schkolnick protocols. Updaters, however, perform *preparatory* splits and merges during their index descent: If an inserter encounters a full node it performs a preparatory node split while a deleter merges nodes that have only a single entry. This means that unlike updaters in the Bayer-Schkolnick protocols who essentially update the entire scope at one time, the scope update in Top-Down protocols is split into several smaller, atomic operations. In particular, a lock on a node can be released once its child is either found to be safe or made safe by the preparatory split or merge mentioned above.

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

C. B-link Protocols

A B-link tree [17], [18], [28] is a modification of the B-tree that uses links to chain together all nodes at each level of the B-tree. Specifically, each node in a B-link tree additionally contains a high key (the highest key of the subtree rooted at this node) and a link to the right sibling, which are used during a node split or a node merge. A node is split in two phases: a half-split, followed by the insertion of an index entry into the appropriate parent. Operations arriving at a newly split node with a search key greater than the high key use the right link to get to the appropriate node. Such a sideways traversal is called a *link-chase*. Merges are also done in two steps [17], via a half-merge followed by the appropriate entry deletion at the next higher level.

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

As discussed above, B-link protocols limit each sub-operation to nodes at a single level. This is in contrast to Top-Down protocols which break down scope updating into sub-operations that involve nodes at *two* adjacent levels of the tree. B-link protocols also differ from Top-Down protocols in that they do their updates in a bottom-up manner. We consider only one B-link protocol here, which exactly implements the above description. This protocol is referred to as the **LY** protocol in [31], and was found to have the best performance of all the above-mentioned protocols with respect to transaction throughput.

III. Real-Time Index Concurrency Control

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

A. Priority Incorporation

Satisfaction of transaction timing constraints is the primary goal in real-time database systems. Therefore, the scheduling policies at the various resources (both physical and logical) in the system can be reasonably expected to be *priority-driven* with the priority assignment scheme being tuned to minimize the number of missed deadlines. The ICC protocols described above do not take transaction priorities into account. This may result in high priority transactions being blocked by low priority transactions, a phenomenon known as *priority inversion* in the real-time literature [29]. Priority inversion can cause the affected high-priority transactions to miss their deadlines and is clearly undesirable. We therefore need to design preemption schemes for ICC protocols in order to adapt them to the real-time environment.

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

The only exception to the above procedure is when a low priority transaction is in the midst of *physically* making updates to a node that is currently locked by it. In this situation, the low priority transaction is not preempted until it has completed these updates and released the lock on the updated node. Since these operations are typically very fast, we expect that the effect of having these extremely short priority inversion periods is negligible.

Locks on index nodes are typically held only for a short duration (in contrast to data locks which are usually retained until EOT) and commercial database systems implement such short duration locks as optimized "fast-locks" or *latches* [23]. In our simulation model also, index node locks are implemented as latches. Since latches are held only for a short duration, it may be questioned as to whether adding preemption to latches is really necessary. The reason latch preemption can make a difference is the following: The time for which computation is performed on a node is usually much shorter than the time it takes to retrieve a node from disk. Given this observation, if two transactions conflict on a parent node but require access to different children, then, in the absence of preemption, a higher priority transaction may have to wait for a lower priority transaction while this lower priority transaction retrieves its desired child from the disk. Our experiments (Section VI) show that adding preemption does have an appreciable performance effect.

In all our protocols, a transaction starts making a physical update only after all the required nodes are in memory. Therefore, when a low priority transaction is in the midst of *physically* making updates to a node that is currently locked by it we do not preempt it (to avoid undoing the physical updates) until it has completed these updates and released the lock on the updated node. Since this operation does not require I/O's, we expect that the effect of having these extremely short priority inversion periods is negligible.

B. *Transaction Serializability*

As mentioned in the Introduction, the KillPercent performance metric applies over entire transactions, not on individual index actions. We therefore need to consider transactions which consist of *multiple* index actions and ensure that transaction serializability is maintained. In our study, we use a real-time variant of the well-known ARIES *Next-Key-Locking* protocol [21] to provide transaction data concurrency control (DCC). In the following discussion of this protocol, we use the term "key" to refer to key values in the index and the term "next key" (with respect to a key value k) to denote the smallest key value in the index that is $\geq k$.

Our variant of the Next-Key-Locking protocol works as follows: A transaction that needs to perform an index operation with respect to a specific key (or key range) first descends the tree to the corresponding leaf. It then obtains the appropriate lock(s) on the associated key(s) from the database concurrency control manager, For point searches an S lock is requested on the search key value (or the next key with respect to the search key, if the search key is not present). A range search operation has to acquire S locks on *each* key that it returns. In addition, it also acquires an S lock on the next key with respect to the last (largest) key in the accessed range. Inserts acquire an X (exclusive) lock on the next key with respect to the inserted key, acquire an X lock on the key to be inserted, insert the key, and then release the lock on the next key. Deletes, on the other hand, acquire an X lock on the next key with respect to the deleted key, acquire an X lock on the key being deleted, delete the key, and then release the lock on the deleted key.

In traditional Next-Key-Locking, all the above-mentioned locks (unless otherwise indicated) are acquired as needed and released only at the end of the transaction (i.e., strict 2PL). For our study we use a real-time version of 2PL called 2PL-HP [1] which incorporates a priority mechanism similar to that described above for the ICC protocols.[5] An important difference, however, is that transactions restarted due to key-value-lock preemptions have to commence the *entire* transaction once again, not just the current index operation.

We wish to clarify here that serializability considerations are relevant with regard to the *data locks* discussed above, not the latches discussed in the previous subsection whose goal is to maintain physical consistency of index nodes during tree traversal.

C. *Undo Transactions*

As mentioned in the Introduction, transactions in an RTDBS 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. We hereafter use the term *undo transaction* to refer to transactions that require undoing of their index actions. The undo transaction removes all the changes that the original transaction had made on the index structure and also releases all its data locks – we explicitly model these operations in our study.

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

IV. Incorporating Admission Control

In the previous section, we discussed how adding priority cognizance is important in the RTDBS context to maximize the number of transactions deadlines that are met. We move on in this section to motivating why augmenting priority cognizance with mechanisms for appropriately *limiting the system multiprogramming level* could help the RTDBS to *further* reduce the number of missed deadlines.

In an RTDBS, a transaction that eventually misses its deadline nonetheless makes use of the physical and logical resources during its sojourn in the system. This resource usage may result in *even more* transactions missing their deadlines, that is, there may be a "cascading" effect. Therefore, if it was possible to design an *admission control* policy that would "magically" recognize (at entry time itself) and shut out those transactions that will eventually miss their deadlines, then it appears likely that the overall system performance would improve. Based on this observation, we have designed an admission policy called **GUARD** (Gatekeeping Using Adaptive eaRliest Deadline), which is described below.

[5] Earlier studies of DCC protocols in firm RTDBS have shown optimistic algorithms to usually perform better than locking protocols (e.g. [10]), especially in environments with significant data contention. However, since almost all the experiments discussed here consider only negligible data contention (but significant index contention) situations, the choice of DCC mechanism has negligible effect on performance and therefore, for simplicity, a locking protocol has been used. Further, open problems remain with respect to integrating optimistic DCC schemes with index management [9], [22].

A. The GUARD Admission Policy

The GUARD admission policy is a modified version of the *Adaptive Earliest Deadline* (AED) priority assignment mechanism proposed in [12]. The mechanism is based on the following observation: The Earliest Deadline (ED) priority assignment policy [19] (transactions with earlier deadlines have higher priority than transactions with later deadlines) *minimizes* the number of missed transaction deadlines when the system is lightly loaded. At heavier loads, however, its performance steeply degrades; in fact, it has been observed to perform worse than even *random* scheduling in this region. The goal of the GUARD policy is therefore to stabilize the overload performance of ED without sacrificing its light-load virtues. It does this by using a feedback process to estimate the number of transactions that are *sustainable* under an ED schedule, as explained below.

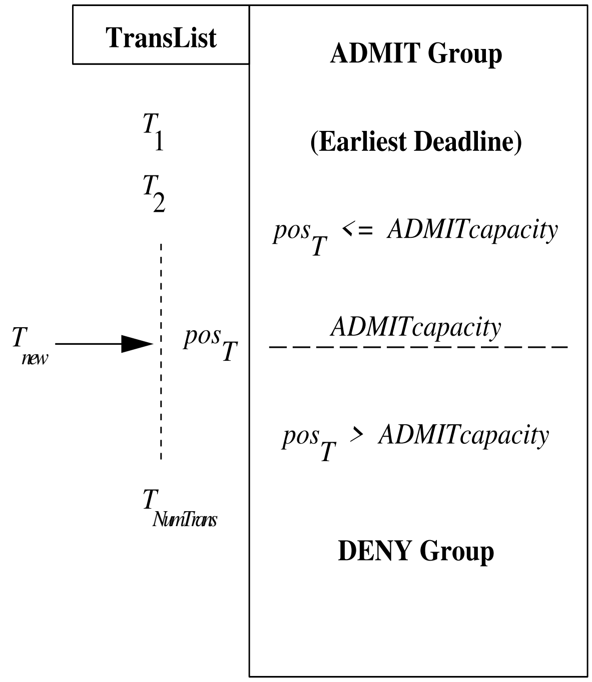

Fig. 1. GUARD Admission Policy

A.1 Group Assignment

In the GUARD policy, all transactions entering the system are dynamically split into two groups, ADMIT and DENY, as illustrated in Figure 1. The assignment of transactions to groups is done in the following manner: Each newly-arrived transaction T is assigned a random integer key, I_T, and then inserted into a I_T-ordered list of transactions. The position pos_T of this transaction in the list is noted. If the value of pos_T is less than or equal to `ADMITcapacity`, which is a dynamic control variable of the GUARD policy, the transaction is assigned to the ADMIT group; otherwise, it is assigned to the DENY group. A transaction is removed from the transaction list either when it completes or when its deadline expires.

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

A.2 Feedback Process

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

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

Step 1
ADMITcapacity :=
$$\lceil HitRatio(ADMIT) * ADMITcapacity * 1.05 \rceil$$

Step 2
if HitRatio(ALL) < 0.95 then
$MaxCapacity := \lceil HitRatio(ALL) * NumTrans * 1.25 \rceil$
ADMITcapacity := Min(ADMITcapacity, MaxCapacity)

A.3 Computation Rationale

We now describe the rationale for the above computation of the `ADMITcapacity` control variable.

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

Step 2 of the `ADMITcapacity` computation is necessary to take care of the following special scenario: If the system experiences a long period where `HitRatio(ALL)` is close to 1.0 due to the system being lightly loaded, it follows that `HitRatio(ADMIT)` will be virtually 1.0 over this extended period. In this situation, the `ADMITcapacity` can become very large due to the 5 percent expansion factor, that is, there is a "runaway" effect. If the transaction arrival rate now increases such that the system becomes overloaded (signaled by `HitRatio(ALL)` falling below 0.95), incrementally bringing the `ADMITcapacity` down from its artificially high value to the right level could take a considerable amount of time (with the feedback process of STEP 1). This means that the system may enter the unstable high-miss region of Earliest Deadline as every new transaction will be assigned to the ADMIT group due to the high `ADMITcapacity` setting. To prevent this from occurring, `MaxCapacity`, an upper bound on the `ADMITcapacity` value is used in STEP 2 to deal with the *transition* from a lightly-loaded condition to an overloaded condition. The `MaxCapacity` is set to be 25 percent greater than an estimate of the "right" `ADMITcapacity` value, which is derived by computing the number of transactions that are *currently* making their deadlines. (The choice of 25 percent is based on our expectation that the estimate is fairly close to the desired value.) After the `ADMITcapacity` is quickly brought down in this fashion to near the appropriate setting, the `HitRatio(ADMIT)` value then takes over as the "fine tuning" mechanism in determining the `ADMITcapacity` setting.

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

B. Comparison with the AED policy

As mentioned at the beginning of this section, the GUARD admission policy is based on the AED priority assignment policy of [12]. The primary difference between GUARD and AED is the following: In GUARD, transactions assigned to the DENY group are not permitted to enter the system. That is, they are "put on the shelf" until their deadline expires and not allowed to use any of the system resources. However, in AED, even these transactions are allowed to utilize the system, albeit with a priority less than that of all those belonging to the ADMIT group. Further, a RandomPriority assignment is used for transactions within the DENY group in the AED policy.

If AED's group assignment is working correctly, most of the transactions of the DENY group will end up missing their deadlines as anticipated and will therefore be finally aborted. Further, these transactions may be repeatedly aborted during their sojourn in the system due to data conflicts with the higher priority transactions of the ADMIT group. A problem with these aborts is that all the index operations completed by the transaction before the abort have to be *undone*, thereby imposing an overhead on the system. It is for this reason that we choose to have an admission policy since transactions that are denied entry into the system obviously do not attract any undo overhead.

We would also like to note here that in our earlier work [6], a different admission policy was utilized – this policy implemented a simple feedback mechanism that monitored the utilization at all the system resources and prevented new transactions from entering the system whenever the utilization of the bottleneck resource exceeded a prescribed amount. For a variety of reasons explained in detail in [26], the GUARD approach is preferable to our earlier policy – this expectation was also confirmed experimentally.

C. Integration with ICC protocols

As mentioned above, the GUARD policy can be applied in conjunction with *any* of the classical ICC protocols. Due to space limitations, we will restrict our attention in this paper to the combination of the GUARD policy with the real-time version of B-link, which we will hereafter refer to as the **GUARD-link** protocol. The results for other combinations are available in [25].

V. Simulation Model and Methodology

To evaluate the performance of the various real-time ICC protocols described in the previous sections, we developed a detailed simulation model of a firm-deadline real-time database system (as mentioned in the Introduction, firm deadlines means that transactions which miss their deadlines are immediately killed). The simulation model is described in the remainder of this section.

The organization of our RTDBS model is based on a combination of the real-time database model of [10] and the B-tree system model of [31], and is shown in Figure 2. There are six components in the model: The *Source* generates transactions, the *Transaction Manager* models the execution of transactions, the *Concurrency Control Manager* controls access to shared data, the *B-tree Manager* implements the B-tree structure and the ICC protocols, the *Resource Manager* models the physical resources in the system, and finally, the *Sink* gathers statistics on transactions exiting from the RTDBS.

A summary of the parameters used in the model are given in Table II. The following subsections describe the

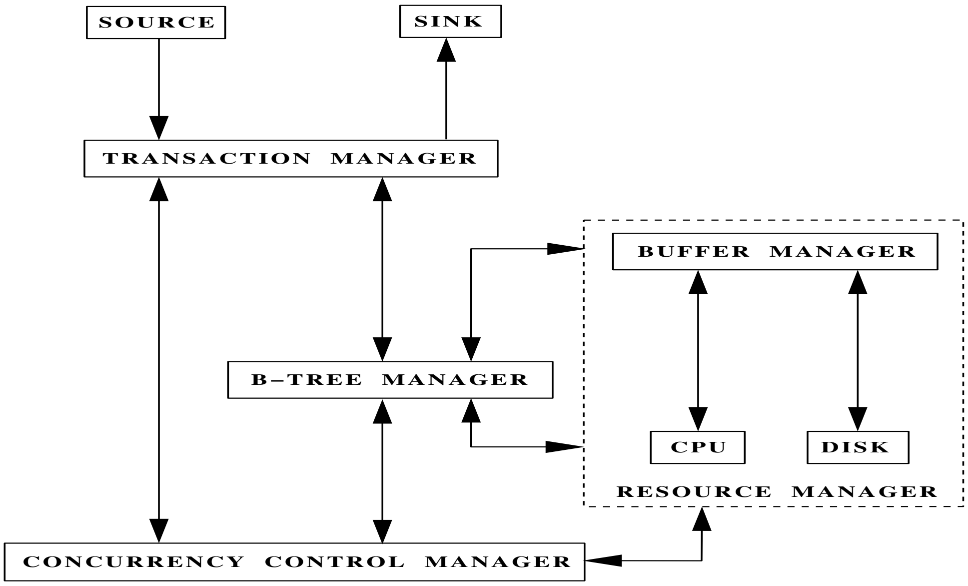

Fig. 2. Simulation Model

workload generation process, the B-tree model and the hardware resource configuration.

TABLE II
MODEL PARAMETERS

Parameter	Meaning	Value
ArrRate	Transaction arrival rate	$0-\infty$
TransSize	Average transaction size	8
SlackFactor	Deadline Slack Factor	4
SearchProb	Proportion of searches	0.0 – 1.0
InsertProb	Proportion of inserts	0.0 – 1.0
DeleteProb	Proportion of deletes	0.0 – 1.0
AppendProb	Proportion of appends	0.0 – 1.0
RangeKeys	Size of range search	$1-\infty$
InitKeys	No. of keys in initial tree	100,000
MaxFanout	Key entries per node	300
NumCPUs	Number of processors	$1-\infty$
SpeedCPU	Processor MIPS	20
LockCPU	Cost for lock/unlock	1000 inst.
LatchCPU	Cost for latch/unlatch	100 inst.
BufCPU	Cost for buffer call	1000 inst.
SearchCPU	Cost for page search	50 inst.
ModifyCPU	Cost for key insert/delete	500 inst.
CopyCPU	Cost for page copy	1000 inst.
NumDisks	Number of disks	$1-\infty$
PageDisk	Disk page access time	20 ms
NumBufs	Size of buffer pool	$1-\infty$

A. Transaction Workload Model

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

The *ArrRate* parameter specifies the mean rate of transaction arrivals. The number of index operations made by each transaction varies uniformly between 0.5 and 1.5 times the value of *TransSize*. The overall proportion of searches, inserts, deletes and appends in the workload is given by the *SearchProb*, *InsertProb*, *DeleteProb* and *AppendProb* parameters, respectively. While generating a new transaction, the type of each operation in the transaction is chosen according to this probability distribution. As mentioned earlier, searches can be either point or range operations, whereas all updates are point operations. For range searches, rather than specifying a range from the domain space, we specify it in terms of the number of keys that are to be retrieved. This number is set using the *RangeKeys* parameter.

Transactions are assigned deadlines with the formula $D_T = A_T + SF * R_T$, where D_T, A_T and R_T are the deadline, arrival time and resource time, respectively, of transaction T, while SF is a slack factor. The *resource time* is the total service time at the resources that the transaction requires for its data processing. The *slack factor* is a constant that provides control over the tightness/slackness of deadlines.[6]

[6]Although the workload generator utilizes information about transaction resource requirements in assigning deadlines, the RTDBS system itself has *no access* to such information since this knowledge is usually hard to come by in practical environments.

B. B-Tree Model

For simplicity, only a single B-tree is modeled and all transaction index accesses are made to this tree. The initial number of keys in the index is determined by the $InitKeys$ parameter. Each index node corresponds to a single disk block and the *MaxFanout* parameter gives the node key capacity, that is, the maximum number of keys in a node. We assume that all keys are of the same size, and that the indexed attribute is a candidate key of the source relation. If an update transaction is aborted, any index modifications that it has made have to be undone to maintain index consistency.

C. Resource Model

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

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

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

D. Priority Assignment

For simplicity, we assume here that all transactions have the same "criticality" or "value" [32].[8] Therefore, the goal of the priority assignment is to minimize the *number* of killed transactions.

The transaction priority assignment used in all the experiments reported here is *Earliest Deadline* [19]. Specifically for the GUARD-link protocol, this priority assignment is operational only within the set of transactions assigned to the ADMIT group, as described in Section IV-A.

E. Performance Metrics

The primary performance metric of our experiments is **KillPercent**, which is the percentage of input transactions that the system is *unable* to complete before their deadlines. A long-term operating region where the KillPercent 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 protocols to brief periods of stress loading. All the KillPercent graphs of this paper show mean values that have relative half-widths about the mean of less than 10 percent at the 90 percent confidence level, with each experiment having been run until at least 20000 transactions were processed by the system. Only statistically significant differences are discussed here.

Apart from the KillPercent metric, we also evaluate two additional performance metrics related to fairness: **SizeFairness (SF)** and **TypeFairness (TF)**. The SF factor captures the extent to which bias is exhibited towards transactions based on their sizes and is evaluated using the formula:

$$SF = \frac{Average\ Size\ of\ In{-}Time\ Transactions}{Average\ Size\ of\ Input\ Transactions}$$

In similar fashion, the TF factor captures the extent to which bias is exhibited towards transactions based on their type. We identify two types of transactions, *read-only* and *update* – read-only transactions are composed exclusively of search index actions while update transactions include at least one update (insert, delete or append) index action. TF is computed with the following formula:

$$TF = \frac{\frac{No.\ of\ In{-}Time\ Read{-}Only\ Transactions}{No.\ of\ In{-}Time\ Transactions}}{\frac{No.\ of\ Input\ Read{-}Only\ Transactions}{No.\ of\ Input\ Transactions}}$$

With the above formulations, a protocol is ideally fair with respect to these metrics if SF and TF are both equal to *one*.

The simulator was instrumented to generate a number of other statistical information, including resource utilizations, number of index node splits and merges, number of link-chases for B-link protocols, number of restarts for optimistic protocols, etc. These secondary measures help to explain the performance behavior of the ICC protocols under various workloads and system conditions.

F. Baseline Parameter Settings

The default settings used in our experiments for the workload and system parameters are listed in Table II. They were chosen to be in accordance with those used in the earlier ICC [31] and RTDBS [10] studies. While the absolute performance profiles of the ICC protocols would, of course, change if alternative parameter settings are used, we expect that the *relative* performance of these protocols will remain qualitatively similar since the model parameters are not protocol-specific.

The initial B-tree is built over a key space that consists of integer values between 1 and 300,000. A random permutation of all the keys that are multiples of 3 in this key space is inserted into the B-tree, that is, the initial number of keys in the B-tree is 100,000. The fanout (key capacity) of each node of the B-tree is set to 300. With this fanout, the resultant initial tree is 3 levels deep, consisting

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

[8] For applications with transactions of varying criticalities, one of the value-cognizant priority assignment mechanisms proposed in the literature can be utilized – see [11] for a detailed study of this issue, including a value-based extension of AED called Hierarchical Earliest Deadline [11].

of 3 internal nodes and 506 leaf nodes. The tree nodes are assumed to be uniformly distributed across all the disks.

Search operations can use all key values in the key space. In contrast, inserts can only use key values that are not exact multiples of 3, whereas deletes can only use the remaining keys (i.e., exact multiples of 3). This operation-to-key assignment scheme is designed to ensure that inserts and deletes do not interfere at the level of key values. Finally, the keys for appends are chosen sequentially from 300,001 onwards.

For all the experiments described here, unless explicitly mentioned otherwise, the resource parameter settings are: $NumCPUs = 1$, $NumDisks = 8$, and $NumBufs = 250$. For the GUARD-link protocol, the feedback-related parameter settings are the same as those used in [12]: `ADMITbatch` = 20 and `ALLBatch` = 20.

VI. Experiments

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

We present our experimental results in four stages: First, in Section VI-A, we evaluate the KillPercent and Fairness performance of the real-time versions of the various ICC protocols, including our new GUARD-link protocol, under varying index contention environments. Then, in Section VI-B, we explore the effectiveness and robustness of the GUARD admission policy. Subsequently, we model the performance effects of incorporating undos in Section VI-C. Finally, the performance effects of range searches are considered in Section VI-D.

A. ICC Protocol Performance

In this section, we study the performance of the real-time versions of the various ICC protocols (X, SIX, OPT, B-link and GUARD-link) under three representative environments that correspond to system conditions of **Low Index Contention (LIC)**, **Moderate Index Contention (MIC)** and **High Index Contention (HIC)**, respectively – these experiments capture environments similar to those considered in [31]. In addition, we present two experiments, **No Index Contention(NIC)** and **No Resource Contention (NRC)**, that attempt to isolate the specific impact of index concurrency control on the overall system performance.

A.1 Low Index Contention (LIC)

In our first experiment, a workload that consisted of 80% searches, 10% inserts and 10% deletes was used. The high percentage of reads as compared to updates results in a low index contention environment. Moreover, the balance of insert and delete index operations limits the number of index node splits and merges. The performance results for this experiment are shown in Figures 3a–c as a function of the transaction arrival rate.

In Figure 3a, which profiles the KillPercent behavior, we first observe that the X protocol performs poorly with respect to the other protocols. This is due to the root of the B-tree becoming a severe bottleneck, as also observed in [15], [31]. The root bottleneck forces transactions to wait much longer to acquire their desired index locks, thereby resulting in many more missed deadlines.

Moving on to the other protocols, we observe that the performance of OPT is almost identical to that of B-link – this is as expected since the number of splits and merges is rather low in this experiment. The relative performance of B-link and SIX shows a more interesting behavior: In the corresponding (non-real-time) experiment in [31], B-link performed much better than SIX throughout the loading range. Here, however, although B-link performs the best for low arrival rates, the situation is reversed at high arrival rates where SIX is noticeably better than B-link. The explanation for this changed behavior is as follows: SIX gives preferential treatment to read index operations over update index operations (by allowing readers to overtake updaters during tree traversal) [31]. This results in commensurately better performance for transactions that have either no updates or only a few updates. A quantitative confirmation of this explanation is shown in Figure 3c, which captures the TypeFairness metric. Here we see that SIX exhibits an extreme bias in favor of read-only transactions, whereas B-link is comparatively much more even-handed. In fact, our measurements showed that at higher loads SIX completes almost exclusively *only* the read-only transactions, which form approximately 20 percent of the workload (in this experiment).

An additional point to note is that in the conventional DBMS of [31], the slower updaters of SIX used to clog up the system, resulting in much higher contention levels and poor performance – in the firm real-time environment, however, that does not happen because transactions are *discarded* as soon as their deadlines expire. Therefore, this type of clogging is inherently prevented.

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

The above observation is incorporated into the design of the GUARD-link protocol, as discussed in Section IV. The

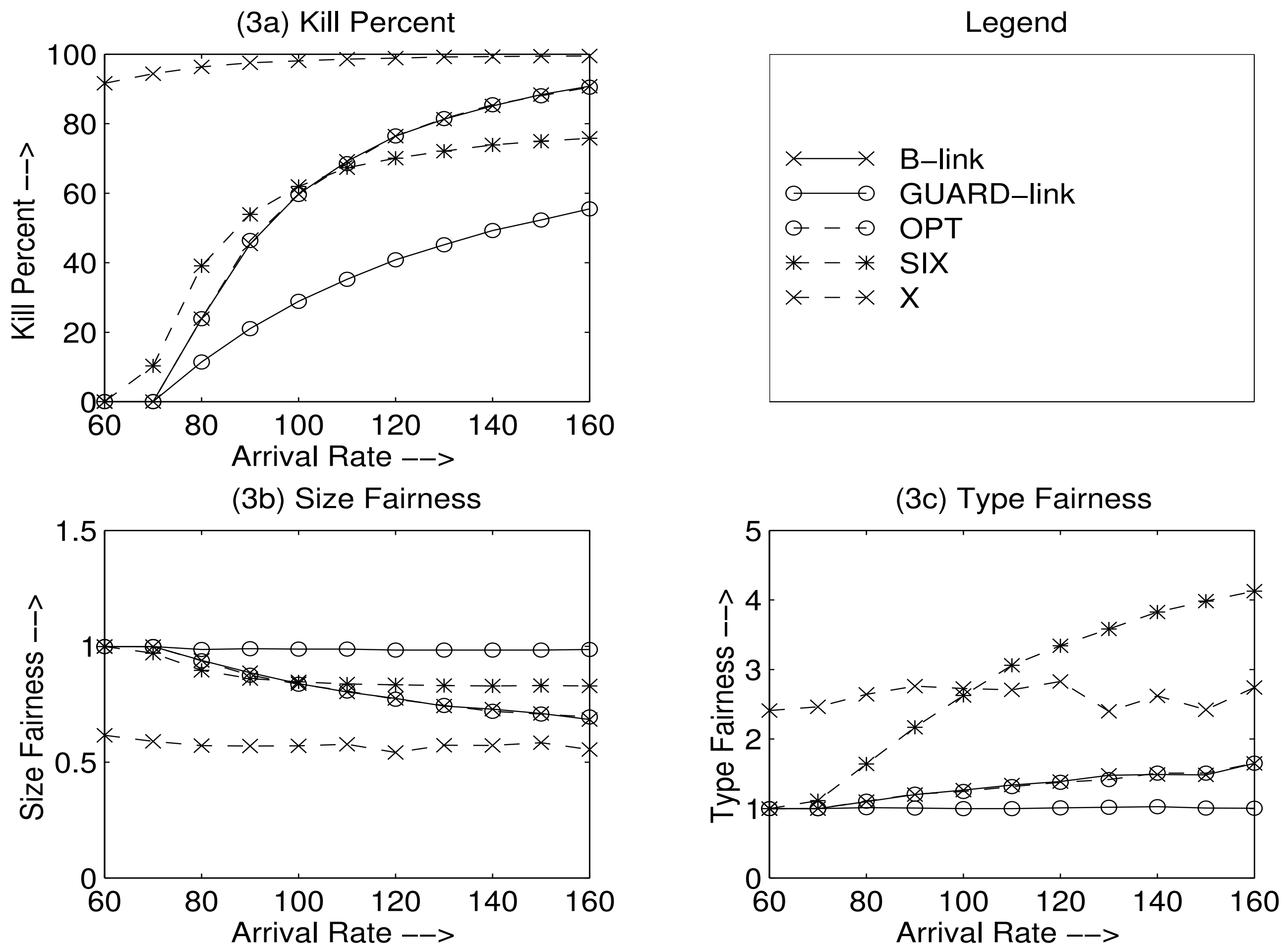

Fig. 3. Low Index Contention

performance of GUARD-link is also shown in Figure 3a and we observe here that a dramatic improvement is achieved – the overload performance of GUARD-link is significantly better than that of both B-link and SIX. This is due to GUARD-link's admission control policy which ensures that the bottleneck resource (in this case, the disk) does not become saturated, thereby comfortably completing the admitted transactions. Also note that the admission control policy of GUARD-link takes effect "gracefully" in that there are no discontinuities in the KillPercent behavior.

From the fairness graphs of Figures 3b and 3c, it is clear that GUARD-link does not *purchase* its good KillPercent performance by selectively completing one or the other class of transactions but, instead, is almost ideally fair with respect to both size and type. In contrast, the classical protocols are noticeably unfair with regard to one or both metrics. For example, B-link and OPT progressively favor smaller-sized transactions with increasing loading levels (Figure 3b). Similarly, SIX shows extreme bias favoring readers (Figure 3c), as explained above.

The reason for GUARD-link's fairness is simple – its admission control policy is non-discriminatory in that it does not selectively admit transactions having a specific characteristic, but instead randomly assigns transactions to the ADMIT or DENY groups. Therefore, the composition of the ADMIT group is a "miniature" replica of the system input workload. Since the protocol ensures that almost all the transactions belonging to the ADMIT group are successfully completed, fairness is automatically ensured.

A.2 Moderate Index Contention (MIC)

In our next experiment, the workload consisted of 100% inserts, resulting in a moderate index contention environment. For this experiment, the performance results are shown in Figures 4a and 4b (the TypeFairness metric is not meaningful here since all transactions are updaters).

In Figure 4a, we observe that SIX and X behave identically with respect to KillPercent – this is only to be expected since, in the absence of readers, SIX locks do not permit any concurrent access, just like X locks. Note also that the performance of these protocols is considerably below that of OPT and B-link.

In the corresponding (non-real-time) experiment in [31], B-link performed noticeably better than OPT. This was because OPT suffered from a large number of restarts caused by high contention, thereby resulting in the root becoming a bottleneck due to the large number of second-pass updaters. In our experiment, however, we observe that the performance of OPT and B-link is very close. The rea-

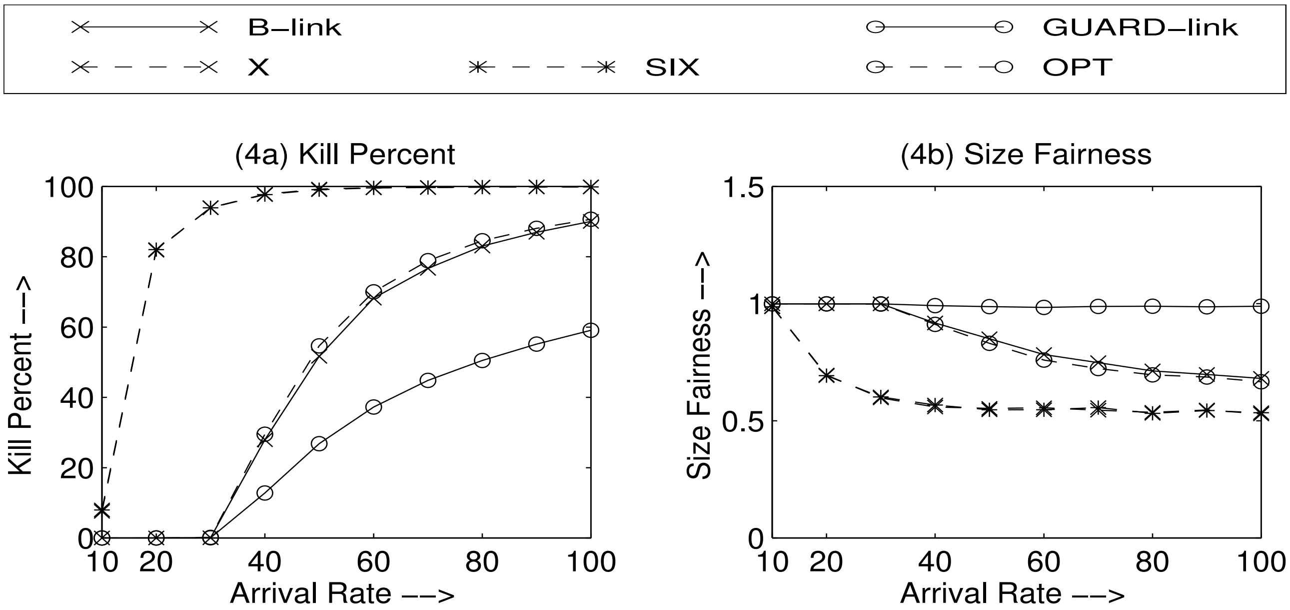

Fig. 4. Moderate Index Contention

son for this surprising result is that the number of restarts for OPT in the real-time environment is, in contrast, quite small. This reduction in restarts arises from the priority feature incorporated in the ICC protocols. To verify this, we ran the same experiment without preemption for index node locks and found that the number of restarts for OPT was significantly higher than in the prioritized case. This shows that adding preemption to index latches can result in tangible performance benefits (the other ICC protocols also exhibited similar improvement due to incorporating preemption).

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

Moving on to the GUARD-link protocol, we observe that its KillPercent performance is again clearly better than that of all the other protocols. In fact, at an arrival rate of 50 transactions per second, GUARD-link reduces the kill percentage by over 20 percent as compared to B-link, and at an arrival rate of 100 transactions per second, by almost 30 percent.

With respect to the SizeFairness metric (Figure 4b), we observe that as in the LIC experiment, GUARD-link provides almost ideal fairness, whereas the other protocols progressively favor the smaller-sized transactions that are relatively easier to complete before their deadlines.

A.3 High Index Contention (HIC)

In the next experiment, the workload consisted of 25% searches and 75% appends. The appends create extremely high contention for the few right-most nodes in the tree, and as a side-effect, ensure that these nodes are permanently in the buffer pool. The keys for the searches are randomly generated and searches therefore interfere only minimally with the appends. The results of this experiment are shown in Figures 5a and 5b. (The X protocol is not shown in these figures since its kill percentage was already close to 50% even at arrival rates where the other protocols had a 0 miss percentage. Also, similar to the previous MIC experiment, the TypeFairness metric is not meaningful here since less than 0.1 percent of the transactions are read-only).

In Figure 5a, we observe that the KillPercent of SIX is much worse than that of both OPT and B-link. This is again due to the root becoming a bottleneck, resulting in much higher average lock waiting times as compared to OPT and B-link.

The gap between the performance of OPT and B-link in Figure 5a is slightly larger than that in the MIC experiment discussed previously. The reason for the increased separation is that the probability of transactions restarting here is higher than in the MIC case since 75% of the index actions are directed towards the right end of the tree. Once again, just as in the MIC case, we experimented with having no preemption for index node locks and found that OPT without preemption had many more restarts than the prioritized OPT protocol. The reduction in the number of restarts for the prioritized OPT protocol makes the performance differences between OPT and B-link to be less significant here than they would otherwise be.

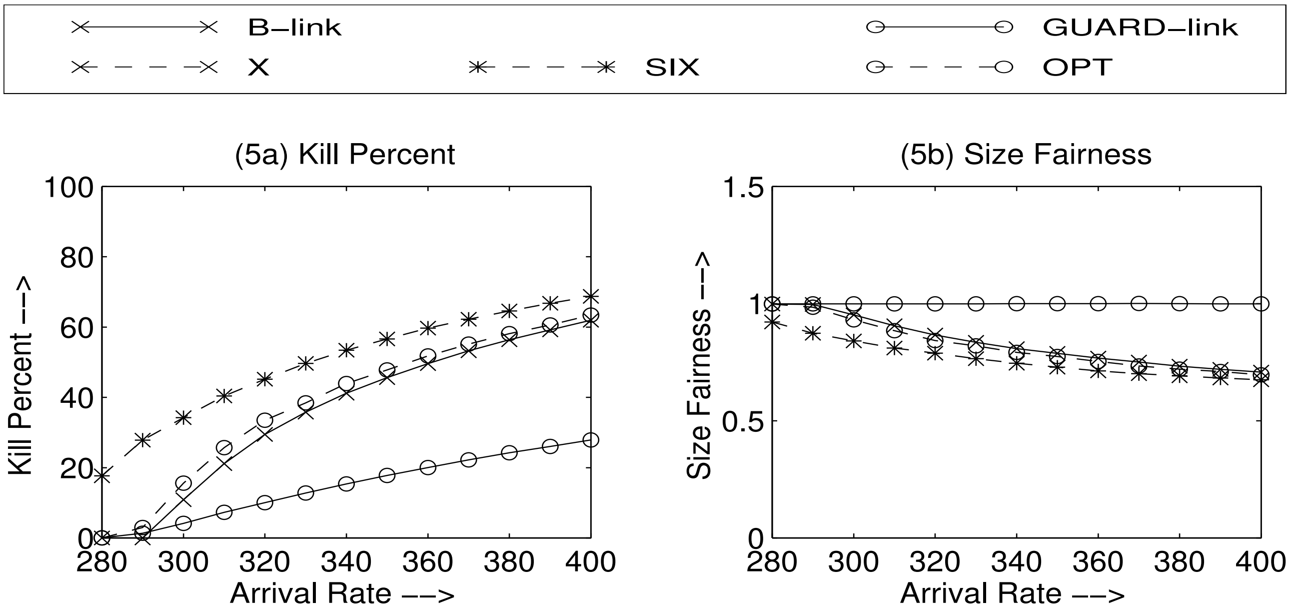

Fig. 5. High Index Contention

Finally, notice that in this experiment too GUARD-link provides the best KillPercent performance although the bottleneck resource is now the CPU, and not the disk as in earlier experiments. In addition, its SizeFairness is virtually ideal, unlike the other protocols which favor smaller-sized transactions.

A.4 No Index Contention (NIC)

In the previous experiments, transactions which missed their deadlines did so due to the *cumulative* effect of three factors: The conflict for the data, the conflict for the physical resources (processors, disks, memory), and the conflict for the index. We now move on to analyzing the specific extent to which each of these factors contributed to the overall kill percentage.

The data conflict due to key value locking was negligible since the number of restarts suffered by a transaction on average was never greater than *0.2 percent* in the above experiments. In order to study the relative extent to which index conflict and (physical) resource conflict contributed to the kill percentage, we conducted an experiment where there were no index conflicts. For this experiment, the workload and system configuration were identical to the LIC experiment except that *all* the index operations were searches. Figure 6 shows the performance of B-link and GUARD-link for this full-search workload and the corresponding figures for the mixed workload of the LIC experiment. Note that the performance of B-link for the full-search workload is significantly better than its performance with the mixed workload. The difference corresponds to the performance drop due to data conflicts at index nodes. This difference is quite substantial even at very high arrival rates and clearly demonstrates that index conflict is a significant performance-determining factor even for the B-link protocol, which is reputed to provide the maximum concurrency. A similar performance difference is observed for the GUARD-link protocol as well.

The above experiment, by demonstrating that index conflict has considerable impact on the real-time performance, clearly highlights the need for designing sophisticated real-time ICC protocols.

Another point to note is that even in the complete absence of index contention (full search), the RTDBS begins to miss transaction deadlines when the transaction arrival rate exceeds a certain level – this is due to the effect of resource contention becoming noticeable, and its impact increases at higher arrival rates, as should be expected.

A.5 No Resource Contention (NRC)

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

We conducted an experiment to evaluate the KillPercent performance that could be achieved for the transaction workload of the LIC experiment in the *absence* of resource contention. For this experiment, the entire index tree was resident in memory and the number of CPUs was infinite –

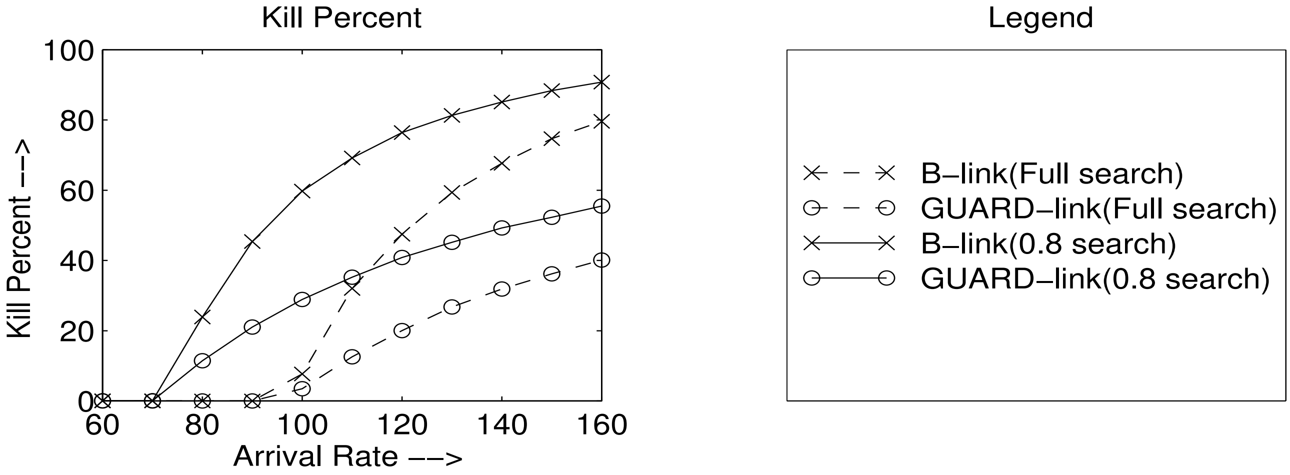

Fig. 6. No Index Contention

TABLE III
Kill Percent for No Resource Contention

ArrRate	SIX	OPT	B-link and GUARD-link
1000	0.02	0.02	0.02
2000	0.02	0.02	0.02
3000	25.74	0.03	0.03
4000	59.36	0.04	0.04
5000	70.21	0.05	0.05
10000	79.23	0.07	0.06
15000	79.70	0.09	0.09
20000	79.83	0.14	0.18

the kill percentage was therefore determined solely by index contention. Note that this means that the performance numbers observed in this experiment capture the *best* performance that each index concurrency control protocol can deliver for the chosen transaction workload.

The performance in this experiment is shown in Table III for SIX, OPT, B-link and GUARD-link (the values for GUARD-link are identical to those of B-link since the kill percentage is very low and therefore GUARD-link's admission control policy does not kick in). Note that OPT and B-link manage to complete almost all the submitted transactions, even at an arrival rate of 20,000 transactions/sec. In contrast, SIX misses close to 80% of transaction deadlines at higher arrival rates. However, the kill percentage of SIX does not degrade beyond 80%. This is yet another indication of the fact that SIX preferentially completes only the read-only transactions (recall that the percentage of read-only transactions is approximately 20% in the LIC workload).

B. Analysis of the GUARD-link Protocol

The results of the above set of experiments (LIC, MIC, HIC, NIC, NRC) indicate that the GUARD-link protocol is capable of simultaneously providing both the lowest kill percentage and close to ideal fairness. We now wish to verify whether the admission control policy of GUARD-link operated as designed for in maintaining the "right" multiprogramming level in the system, resulting in the improved performance. To this end, we measured the "hit ratio" for the ADMIT group – the hit ratio is the fraction of transactions that complete before their deadlines – in the above set of experiments. These results are shown in Figures 7a–d for the LIC, MIC, HIC and NIC environments, respectively. The graphs clearly indicate that the GUARD admission policy is uniformly successful in maintaining a hit ratio of 1.0 in the underload region and a hit ratio close to 0.95 in the overload region, thus meeting its design goals.

As discussed before, the GUARD-link protocol uses two algorithmic parameters, `ADMITbatch` and `ALLbatch`, in its feedback process. These parameters were both set to 20 in the above set of experiments (these settings are the same as those recommended for the AED priority assignment policy in [12]). To evaluate the performance sensitivity to these protocol parameters, we conducted two experiments wherein the `ADMITbatch` and `ALLbatch` parameters were individually varied from 16 to 24, respectively, keeping the other parameter fixed at 20.

The results of these experiments for the LIC environment are shown in Figures 8a and 8b for `ADMITbatch` and `ALLbatch`, respectively. Note that GUARD-link behaves almost indistinguishably for all the `ADMITbatch` and `ALLbatch` range of values that were considered. This robustness of the GUARD policy to the choice made for the feedback parameters was also observed for the AED policy in [12]. In summary, these experiments indicate that the performance of GUARD-link is reasonably stable and does not depend critically on the choice of algorithmic parameters.

C. Effect of Undos

We now move on to considering the performance impact of modeling the *undos* of the index actions of aborted transactions. It is tempting to surmise that there is really no need to explicitly evaluate this feature based on the following argument, which was advanced in [6]: The

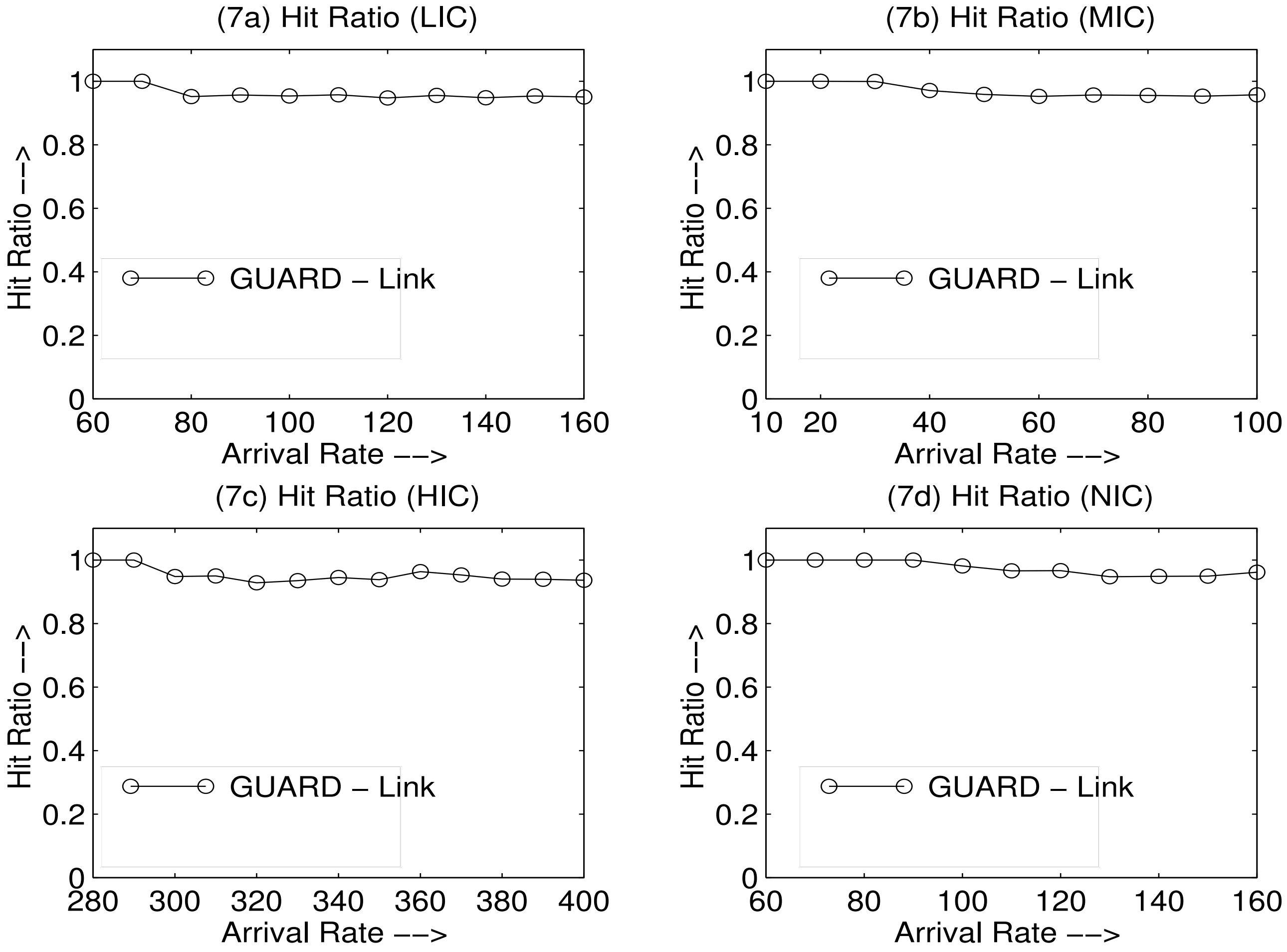

Fig. 7. Hit Ratio for ADMIT group (GUARD-link)

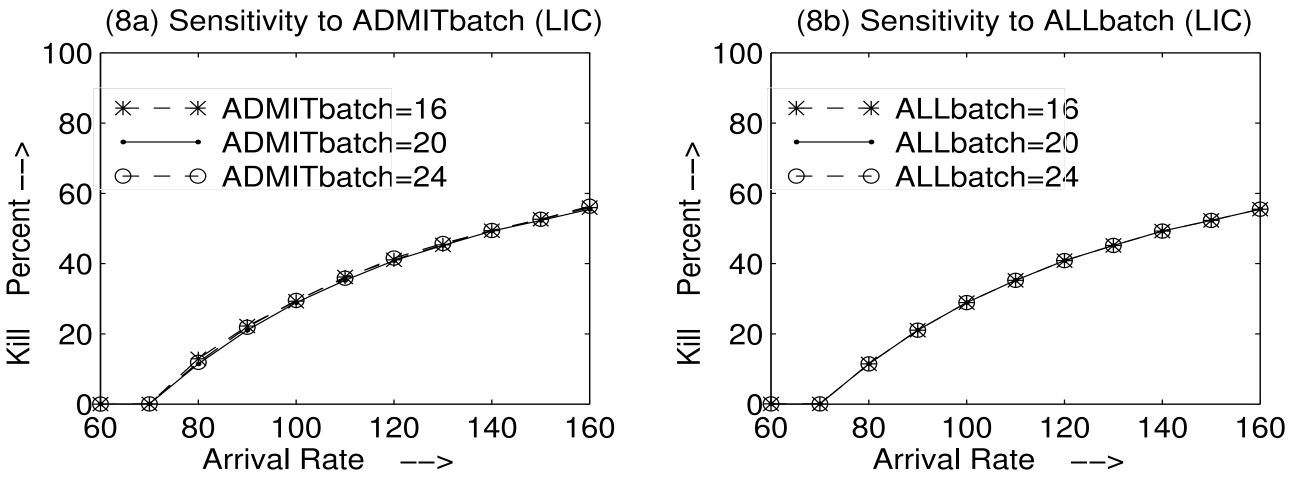

Fig. 8. GUARD-link Sensitivity Analysis

number of undos can be expected to be directly related to the transaction kill percentage, that is, a higher kill percentage leads to more undos. Given this, implementing undos in the model would only result in *further increasing* the difference in performance of the protocols seen in the above experiments. This is because the poorly performing algorithms would have *more* clean-up work to do than the better protocols and therefore miss even more transaction deadlines. In summary, there is a direct positive feedback between the undo overhead and the kill percentage. Due to this relationship, incorporating undo actions will correspondingly increase, but not qualitatively alter, the observed performance differences between the various protocols.

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

With the inclusion of undos, Figures 9a–b, 9c–d and 9e–f present the results for the LIC, MIC and HIC environments, respectively. For graph clarity, we separately show the performance of the B-link-based protocols and the other protocols. To aid in understanding the effects of the undos, we have also included the corresponding results for the "no-undo" model.

We first see in these figures that the effect of undos is significant only in the HIC environment (Figures 9e–f) but in the LIC and MIC environments (Figures 9a–b, 9c–d), 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 kill percentage 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 9b). These results, which suggest that the presence of overheads results in *improved* performance may appear at first glance to be illogical. A careful investigation showed, however, that there are subtle effects that come into play in the undo model, which do indeed cause this apparently strange behavior. We explain these reasons below.

First, the primary reason for little adverse performance impact in the LIC and MIC environments is that the *disk* is the bottleneck in these experiments. Undo operations, however, typically only need the CPU since the index pages they require will usually be *already resident in the buffer pool* as they have been accessed recently. This was quantitatively confirmed by measuring the buffer hit ratio of undo operations – it was typically about *90 percent*, as against a hit ratio of only around 50 percent for the normal operations. 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 GUARD-link protocol specifically, an additional factor is at work: Its use of admission control and completion of most of the transactions that it admits 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 GUARD-link protocol. This is confirmed in Figure 9e, where we see that even in the HIC environment, where undos were found to have a significant effect on the classical protocols, the GUARD-link protocol is 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. 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. Updaters, on the other hand, have to wait for each other since SIX locks are mutually incompatible. Due to the lock-coupling nature of SIX, this may result in a *convoy* phenomenon, wherein there is a chain of updaters from the leaf to the root, each holding a latch that the next one wants. Since index trees are usually of small height, such convoys can occur very easily. Moreover, the convoy may persist for quite a while since the updater on the leaf node typically has to wait for the leaf node to be brought from the disk before it can release its latch on the parent node (and that too, only if the leaf is safe).

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

D. 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 index operation mix similar to that of the LIC experiment, which was composed of 80 percent searches, 10 percent inserts and 10 percent deletes. The only difference here is that each search now is a *range search* which retrieves *RangeKeys* key values, set equal to 10 for this experiment. The results for this experiment are presented in Figures 10a–e.

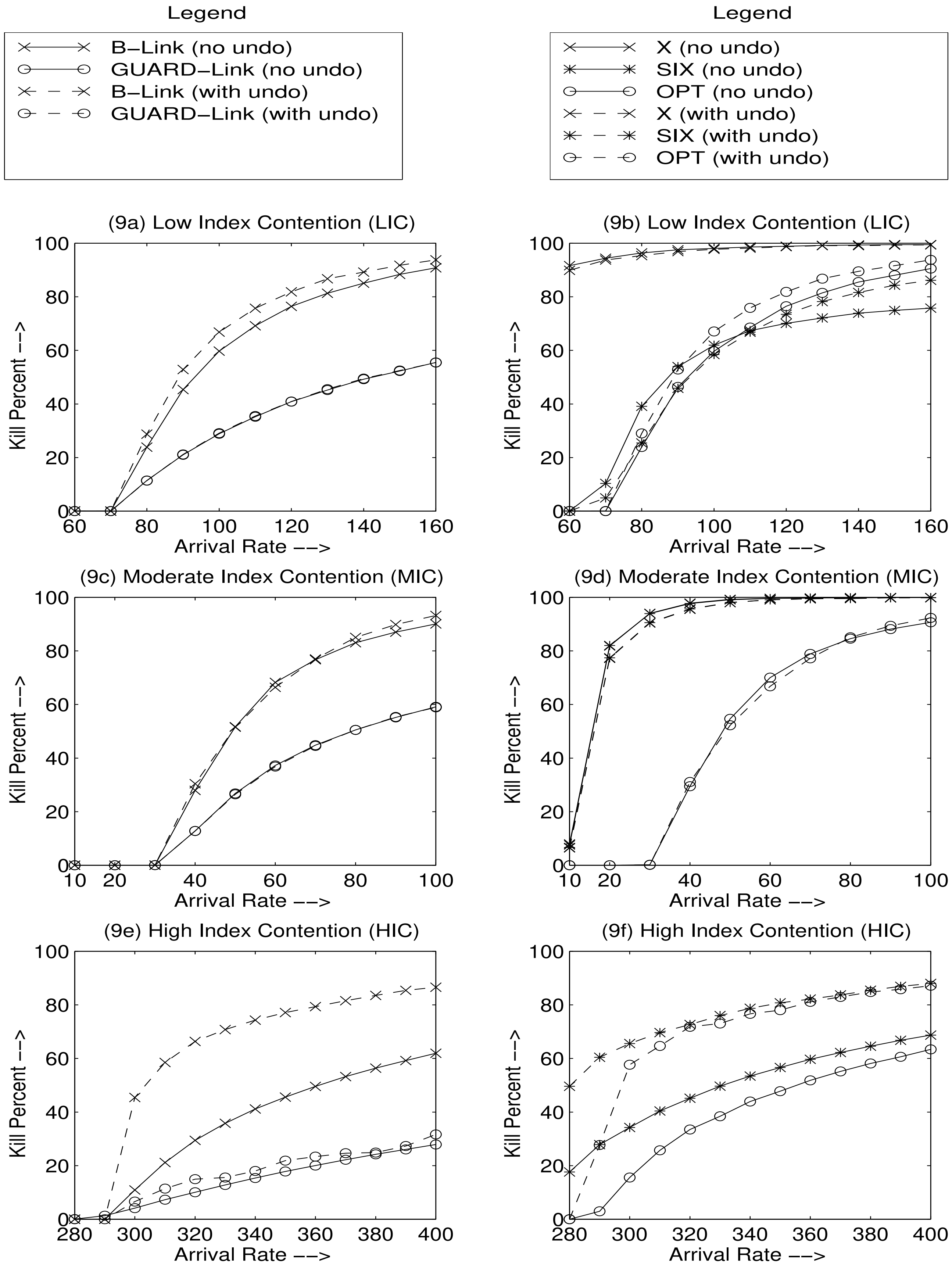

Fig. 9. Effect of Undos

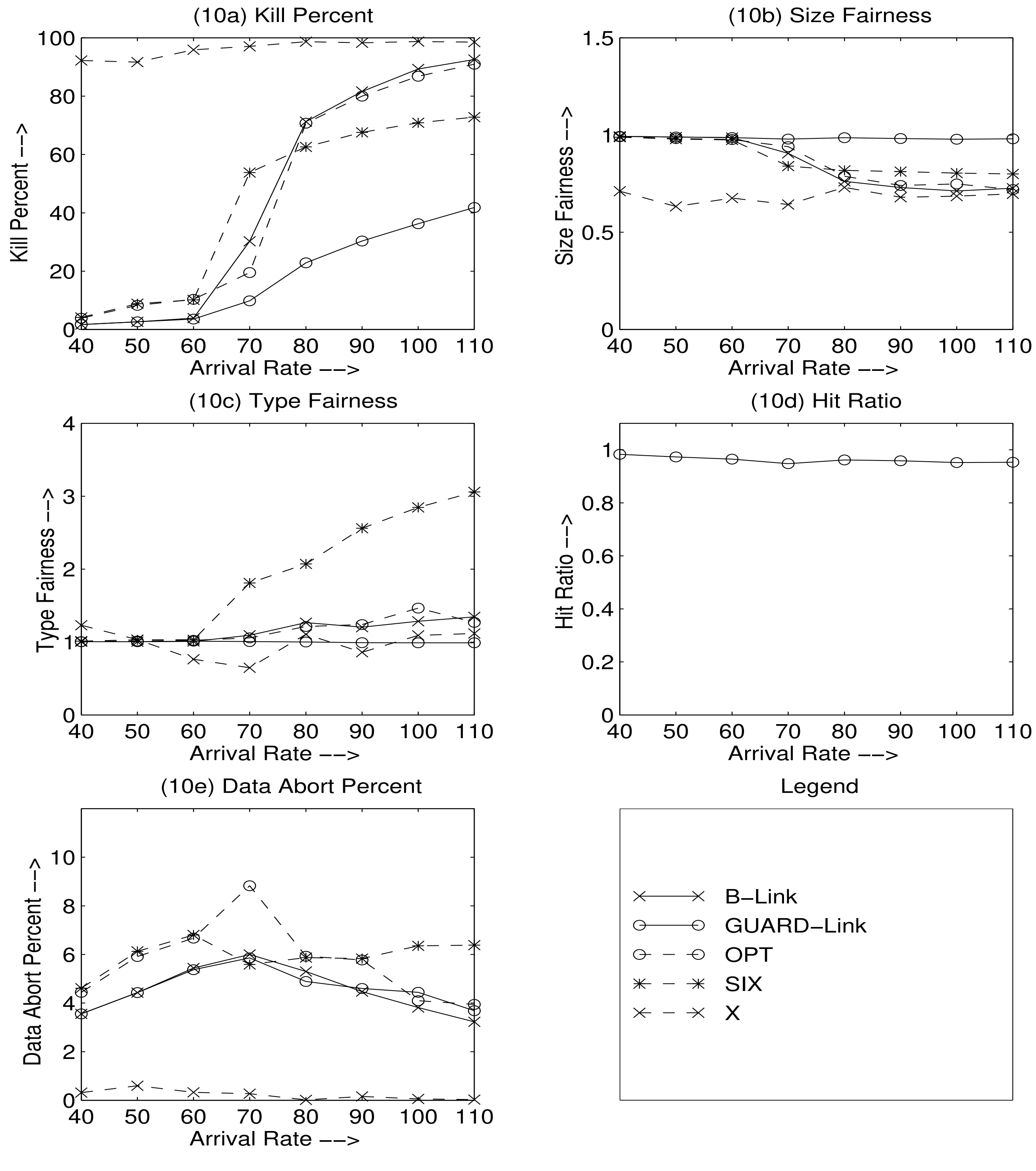

Fig. 10. Range Queries

In Figure 10a, which shows the KillPercent performance of the ICC protocols, we first observe that the results are qualitatively similar to those seen for the original LIC experiment with point queries (Figure 3a). Quantitatively, however, the performance differences between the protocols generally *increase*. For example, the difference between B-link and GUARD-link is *significantly* larger than that seen in Figure 3a. The reason for this increase is that range searches significantly increase the level of *data contention* in the system since they hold locks on several key values simultaneously. This is verified in Figure 10e, which captures the percentage of transactions that were restarted due to key-value contention – the figure shows a relatively high value of data contention in the system as compared to the equivalent point query workload.

We further observe in Figures 10a–c that the GUARD-link protocol provides the best performance for all metrics in this experiment also by maintaining, as desired, a 0.95

hit ratio in the overload region (Figure 10d).

E. Other Experiments

We conducted several other experiments to explore various regions of the workload space. In particular, we evaluated the sensitivity of the results to the transaction deadline slack factor, the transaction size, the database size, the feedback parameters, etc. The relative performance of the protocols in these additional experiments remained qualitatively similar to those seen in the experiments described here and we therefore do not discuss them further in this paper.

VII. Conclusions

In this paper, we investigated, for the first time in the RTDBS literature, the problem of index concurrency control for real-time applications with firm deadlines. Using a detailed simulation model, we studied the real-time performance of three different classes of ICC protocols: Bayer-Schkolnick, Top-Down and B-link, under a range of workloads and operating conditions. We also proposed and evaluated a new ICC protocol called GUARD-link, which augmented the classical B-link protocol with an admission control mechanism. The performance metrics in our experiments were the percentage of killed transactions and the fairness with respect to transaction size and type. We also included the modeling of undos of index actions of aborted transactions in the experimental framework and evaluated the performance effects of range queries.

Our experimental results showed that two factors characteristic of the (firm) real-time domain: Addition of priority and discarding of late transactions, significantly affected the performance of the ICC protocols. In particular, B-link missed many more deadlines at high loads as compared to lock-coupling protocols. This was in contrast to conventional DBMS where B-link always exhibited the best throughput performance. In fact, the very reason for its good performance in conventional DBMS (full resource utilization) turned out to be a liability here. Secondly, the optimistic protocol, OPT, performed almost as well as B-link even under high index contention conditions (in contrast to conventional DBMS). This was because prioritization of transactions caused a marked decrease in the number of index operation restarts. In short, our experimental results show that in moving from the conventional DBMS domain to the RTDBS domain, there are new performance-related forces that come into effect and that these factors can cause index performance behaviors that were valid in a conventional DBMS setting to be significantly altered in the corresponding RTDBS setting.[9]

The new GUARD-link protocol, by virtue of its admission control policy, which successfully limited transaction admissions to a sustainable number, significantly reduced the kill percentage of B-link in the overload region and thereby provided the best performance over the entire loading range. This clearly demonstrates the need for admission control in index management for real-time database systems. Interestingly, such need for load control has also been identified in other modules of real-time database systems (e.g. [12], [27]).

Apart from its good KillPercent performance, GUARD-link also provided close to *ideal fairness*, not discriminating either based on transaction size or on transaction type. In contrast, all the classical protocols showed bias in favor of smaller-sized transactions and in favor of read-only transactions with increasing loading levels. Finally, we showed that GUARD-link is relatively *robust* to the settings chosen for its algorithmic parameters.

Performing the undos of index actions of aborted transactions was expected 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 GUARD-link protocol, due to its admission control policy, undos have virtually no effect under any contention level. In addition, undos can have an unanticipated beneficial impact of (a) reducing the number of index node splits and merges, and (b) for the SIX protocols, preventing extended formation of updater convoys. Finally, our experiments involving range queries showed GUARD-link to perform far better than B-link.

Although not explicitly discussed in this paper, we have also found the design and performance of GUARD-link to be significantly superior in several ways to that of LAB-link (the protocol that we had proposed in an earlier effort [6] based on a different admission policy) – the details of the improvements are available in [26].

In summary, we suggest that designers of real-time database systems may find the GUARD-link protocol proposed in this paper to be a good choice for real-time index concurrency control.

A. Future Work

We have not considered in our study a high-performance ICC protocol called ARIES/IM that was proposed in [23]. In [31], several reasons were advanced as to why ARIES/IM might be expected to exhibit performance behavior similar to that of the B-link protocol in the conventional database domain. While we expect a similar outcome to also arise in the real-time environment, confirmation of this expectation requires actual implementation and evaluation of the ARIES/IM protocol — we intend to do this in our future research work.

Another interesting issue is to extend our analysis to *multi-dimensional* index structures such as, for example, $R-trees$ [8]. We expect that these structures would be utilized in real-time applications such as mobile databases that include spatial data in their repositories.

Acknowledgments

Preliminary and partial versions of the work reported here were presented earlier in [6], [13]. Brajesh Goyal, S. R. Narayanan and V. Srinivasan made significant contributions to the research effort. Krithi Ramamritham provided expert feedback on early drafts of this paper.

[9] A similar, though unrelated, phenomenon has been observed for *data* concurrency control also [10].

The work of J. R. Haritsa was supported in part by research grants from the Dept. of Science and Technology and the Dept. of Bio-technology, Govt. of India.

References

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

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

[3] A. Biliris, "A Comparative Study of Concurrency Control Methods in B-trees," in *Proc. of the Aegean Workshop on Computing*, July 1986.

[4] D. Comer, "The Ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, no. 4, 1979.

[5] A. Bestavros (editor), "Special Issue on Real-Time Database Systems," *SIGMOD Record*, vol. 25, no. 1, March 1996.

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

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

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

[9] T. Haerder, "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, vol. 9, no. 2, 1984.

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

[11] J. Haritsa, M. Carey, and M. Livny, "Value-based Scheduling in Real-Time Database Systems," *Intl. Journal on Very Large Data Bases*, vol. 2, no. 2, April 1993.

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

[13] J. Haritsa and S. Seshadri, "Real-Time Index Concurrency Control," *SIGMOD Record*, vol. 25, no. 1, March 1996.

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

[15] T. Johnson and D. Shasha, "A Framework for the Performance Analysis of Concurrent B-tree Algorithms," in *Proc. of ACM Symp. on Principles of Database Systems*, April 1990.

[16] Y. Kwong and D. Wood, "A New Method for Concurrency in B-trees," *IEEE Trans. on Software Engineering*, vol. 8, no. 3, May 1982.

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

[18] P. Lehman and S. Yao, "Efficient Locking for Concurrent Operations on B-trees," *ACM Trans. on Database Systems*, vol. 6, no. 4, 1981.

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

[20] D. Lomet and B. Salzberg, "Access Method Concurrency with Recovery," in *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1992.

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

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

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

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

[25] S. Narayanan, "Index Concurrency Control in Firm Real-Time Data Bases," M.S. thesis, Dept. of Computer Science and Automation, Indian Institute of Science, January 1997.

[26] S. Narayanan, B. Goyal, J. Haritsa, and S. Seshadri, "Robust Real-Time Index Concurrency Control," Tech. Rep. TR-97-03, DSL/SERC, Indian Institute of Science, 1997.

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

[28] Y. Sagiv, "Concurrent Operations on B*-trees with Overtaking," *Journal of Computer and System Sciences*, vol. 33, no. 2, 1986.

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

[30] D. Shasha and N. Goodman, "Concurrent Search Structure Algorithms," *ACM Trans. on Database Systems*, vol. 13, no. 1, March 1988.

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

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

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

[34] O. Ulusoy, "Research Issues in Real-Time Database Systems," Tech. Rep. BU-CEIS-94-32, Dept. of Computer Engg. and Information Science, Bilkent University, Turkey, 1994.

Jayant R. Haritsa received the BTech degree in Electronics and Communications Engineering from Indian Institute of Technology (Madras), and the MS and PhD degrees in Computer Science from Univ. of Wisconsin (Madison). Currently, he is on the faculty of the Supercomputer Education & Research Centre and the Dept. of Computer Science & Automation at the Indian Institute of Science, Bangalore. His research interests are in database systems and real-time systems. He is a member of IEEE and ACM, and an associate editor of the Intl. Journal of Real-Time Systems.

S. Seshadri received his BTech degree in Computer Science from Indian Institute of Technology (Madras) and then the MS and PhD degrees from Univ. of Wisconsin (Madison). Currently, he is a Member of Technical Staff at Lucent Bell Laboratories, Murray Hill, NJ and also on the faculty of the Indian Institute of Technology (Bombay). His current research interests are in database systems and the world-wide web.