Performance Characteristics of Epsilon Serializability with Hierarchical Inconsistency Bounds*

Mohan Kamath and *Krithi Ramamritham*
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
{*kamath@freya.cs.umass.edu*, *krithi@nirvan.cs.umass.edu*}

Abstract

Epsilon serializability (ESR) is a weaker form of correctness designed to provide more concurrency than classic serializability (SR) by allowing, for example, query transactions to view inconsistent data in a controlled fashion, by limiting the inconsistency within the specified bounds. In this paper, we introduce the notion of hierarchical inconsistency bounds that allows inconsistency to be specified at different granularities — with transactions being at the top of the hierarchy, objects at the bottom, and groups in between. We discuss mechanisms needed to control the inconsistency so that it lies within the specified bounds. This paper also reports on an evaluation of the performance improvement due to ESR. We make two important observations. First, the thrashing point shifts to a higher multiprogramming level when transaction inconsistency bounds are increased. Further, for a particular multiprogramming level and a particular transaction inconsistency bound, the throughput does not increase with increasing object inconsistency bounds but peaks at some intermediate value.

1 Introduction

Epsilon Serializability (ESR) is a weaker form of correctness that allows transactions to view inconsistent data in a limited fashion. ESR does this by permitting some non-SR execution schedules, thereby enhancing concurrency. Potential benefits from ESR include increased system availability and greater autonomy [5,8]. In typical scenarios, lengthy epsilon transactions (ETs) that just perform queries may execute in spite of ongoing concurrent update transactions. In such a situation, query ETs may view inconsistent data. Thus an update ET may *export* some inconsistency when it updates a data item while query ETs are in progress; a query ET may *import* some inconsistency when it reads possibly inconsistent data items while update ETs are in progress. The correctness notion in ESR is based on limiting the amount of imported and exported inconsistency within the specified bounds. If the bounds are set to zero, ESR reduces to SR. We review some key aspects of ESR further in section 2.

Let us consider an example. A large bank may wish to know how many dollars, in millions, are there in the corporate accounts of its clients. If this query is executed directly on the accounts during banking hours, there would be substantial interference from the ongoing updates. Most of the interference is irrelevant, however, since typical updates refer to small amounts compared to the query's results, which is in millions of dollars. This can be exploited using ESR, to execute the query during banking hours with the desired accuracy. Specifically, under ESR, the result would be guaranteed to be within \$ 1,000,000 of a consistent value (produced by some serial execution of the same transaction). For example, if the ET returns the value \$ 450,500,000 then at least one of the serial transaction executions would have yielded a serializable query result in the \$ 450,500,000 $\pm$ \$ 1,000,000 interval. Though update ETs can view inconsistent data the same way query ETs do, in this paper we focus our attention on the situation where query ETs run concurrently with *consistent* update ETs in a centralized system.

This paper introduce the notion of *hierarchical inconsistency bounds*, which allows inconsistency specification at different levels. From a data organization point of view, data objects can be grouped hierarchically based on some common features and interrelationships. Depending upon the relative importance of the groups, there could be a limit on the inconsistency resulting from update to each of the groups. This is the motivation for the concept of hierarchical inconsistency bounds. Bounds on transactions are specified at the top of the hierarchy, while bounds on the objects are specified at the bottom, and groups in between. Section 3 introduces hierarchical inconsistency bounds using a practical example. It also discusses the two-level system used in our implementation — the transaction level and the object level.

In this paper, we discuss the various control aspects of ESR based on timestamp ordering as it forms the basis of concurrency control in our experiments. Section 4 gives further details of timestamp based ESR where we identify the situations where inconsistent operations will be allowed to execute, provided they do not violate the inconsistency bounds. We present detailed mechanisms for controlling the inconsistency seen by the queries within the specified bounds at different levels of a hierarchy in section 5. Wu et. al [8] suggest some general ways by which the amount of inconsistency can be controlled in ETs. However our notion of hierarchical inconsistency control attempts to provide a finer-grained approach to controlling inconsistencies in ETs.

ESR is relatively a new concept and its performance potential has been predicted in the literature [5,8] but never evaluated. So another contribution of our work is the evaluation of the quantitative performance improvement resulting from ESR by changing the various settings of the inconsistency bounds. Previous performance studies of concurrency control ([1], for example) have used simulation models in evaluating the performance, whereas we have actually built a prototype system, to do a more realistic appraisal of ESR. Our prototype transaction processing system is based on the client-server model. Details of our prototype system are presented in Section 6. Section 7 discusses the objectives of the performance tests. We look at some of the performance metrics and the various settings for the tests. Results of the performance tests are presented in Section 8. Here, we also discuss the behavior of the system and the effects of the various inconsistency bounds on system performance. Section 9 concludes with a summary of the paper.

2 Review of Epsilon Serializability

In this section we touch upon some of the key aspects of ESR that are necessary for our discussion. While [8] gives the details of ESR, a more precise and formal description can be found in [7].

To an application designer and transaction programmer, an ET is a classic transaction with the addition of inconsistency limits. A query ET has an *import-limit*, which specifies the maximum amount of inconsistency that can be imported by it. Similarly, an update ET has an *export-limit* that specifies the maximum amount of inconsistency that can be exported by it. Henceforth

*This material is based upon work supported by the National Science Foundation under grant IRI - 9109210.

these will be referred to as TIL (*transaction-import-limit*) and TEL (*transaction-export-limit*) respectively. For a query ET Q_i and an update ET U_j they are denoted as TIL_{Q_i} and TEL_{U_j} respectively. In addition to the overall inconsistency experienced by a transaction, bounds could be placed, for example on inconsistency due to each object. These are called OIL (*object-import-limit*) and OEL (*object-export-limit*). Given a query Q, for an object x they are denoted as OIL_x^Q and OEL_x^Q respectively. OIL_x^Q specifies the maximum amount of inconsistency Q can view with respect to object x and similarly OEL_x^T specifies the maximum amount of inconsistency an update transaction T can export with respect to object x. OIL_x^Q is represented as $import\_limit_x^Q$ in [7]. In the rest of this paper OIL_x^Q will be assumed to be the same for all Q and is denoted by OIL_x. Similarly, OEL_x^T will be assumed to be the same for all T and will be denoted by OEL_x. A more detailed discussion on the bounds appears in sections 3 and 5. An application designer specifies the limits for each ET and the system ensures that these limits are not exceeded during execution.

We define the *proper* value[1] of an object that is being read by a query Q as the value that would have been seen by the reads of Q if no concurrent update ETs were allowed. $Inconsistency\_from\_Read$ is defined as the difference between the *present* value of the object and the *proper* value of the object. We define the *new* value of an object as the value that would be written by a write if it is allowed to execute. $Inconsistency\_from\_Write$ is defined as the difference between the *new* value of the object and the *proper* value of the object. Our notion of this is slightly different from the one mentioned in [8] and we discuss this further in section 5.

A database state is a set of data values. A database state space is a set of all database states. ESR is applicable to a database state space S_{DB} if it is a metric space. Eventhough this limits the applicability of ESR to a certain extent, many database state spaces have such a regular geometry, for example, dollar amount of bank accounts and airplane seats in airline reservation systems.

3 Specifying Inconsistency Bounds

Inconsistency in a data item x with respect to a query q is defined as the difference between the current value of x and the value of x if no updates on x were allowed to execute concurrently. We will denote this by $Inconsistency_x$. When an inconsistency bound, $Limit_x$, is specified for a data item x with respect to query Q, $Inconsistency_x$ should be less than or equal to $Limit_x$. In the previous literature on ESR, inconsistencies have been specified with respect to transactions [8] or with respect to objects [7]. Here we first present the concept of hierarchical inconsistency bounds and then focus on specifying inconsistency in a two level system - *i.e.* at the root and the leaf levels (as we have done in our tests using the prototype system).

3.1 Hierarchical Inconsistency Bounds

In a banking system or an airline reservation system, data can be grouped hierarchically based on some commonality. Hence while executing a transaction by reading objects, apart from specifying a bound on the overall inconsistency experienced by the transaction, bounds can also be specified on the inconsistency arising from groups and data items.

To motivate the need for hierarchical inconsistency bounds we look at a practical situation. *Figure 1* pertains to a banking system. For accounting purposes, the bank categorizes the accounts broadly into some classes such as *company, preferred customer, personal accounts* etc.. Let us consider the case when the bank needs to estimate the *overall* amount held by the bank. Apart from specifying the inconsistency bound on the *overall* estimate ($Limit_{overall}$), it can specify inconsistency bounds for each of the categories ($Limit_{company}$,$Limit_{preferred}$,$Limit_{personal}$ etc..). Further, each category could be subclassified and can be assigned an inconsistency bound ($Limit_{com1}$,$Limit_{com2}$ etc..). This will proceed till the inconsistency is specified at the leaf level ($Limit_{div1}$,$Limit_{div2}$, etc..). It should be noted that all these bounds are with respect to an individual transaction.

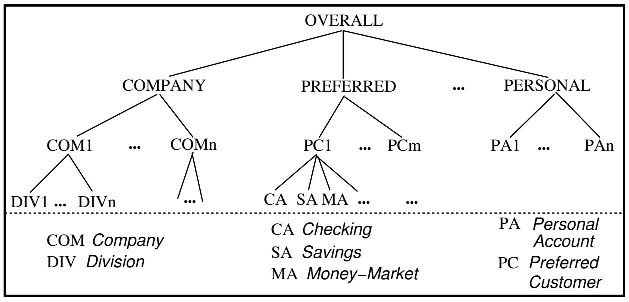

Figure 1: Hierarchy of objects in banking systems

Essentially, the constraints are hierarchically ordered, the inconsistency limits being associated with the *nodes* of the tree. It should be noted that, the data items (objects) are present only at the leaf level and the intermediate nodes just represent groups. Using the transitive relationship between a node and its children, it can be seen that, if a node m has a inconsistency limit $\mathcal{L}$, the sum of the inconsistencies viewed by all the leaves (data items) of the subtree rooted at node m is less than or equal to $\mathcal{L}$. The inconsistency limit on a node at level i places a bound on the inconsistencies viewed by nodes at level $i+1$. Also, the inconsistency limit on a node places bounds on the inconsistency tolerable by its siblings. The information flow is top-down during the inconsistency specification stage and bottom-up during the control stage *i.e.* at transaction execution time. During the control stage if the inconsistency bounds are violated at any level of the tree, then the transaction has to be aborted.

The inconsistency limits specified in the above example imply the following constraints:

$\{Inconsistency_{company} + Inconsistency_{preferred} + \ldots + Inconsistency_{personal}\} \leq Limit_{overall}$,

$$\sum_{i=1}^{n} Inconsistency_{com_i} \leq Limit_{company} ,$$

$$\sum_{j=1}^{n} Inconsistency_{div_j} \leq Limit_{company_k} \ etc...$$

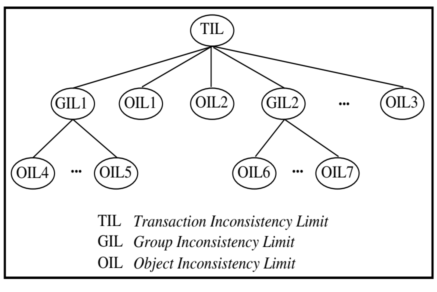

Figure 2: Specifying inconsistency at multiple levels.

[1] In [7] this *proper* value is denoted by $x^Q_{initial}$.

There are many different ways of specifying inconsistency limits at multiple levels. Inconsistency bounds could also be specified using relative weights for the nodes in the tree and thus the inconsistency viewed at a level will be the weighted sum of the inconsistency of the nodes at that level. Referring to *Figure 2*, a transaction accesses objects, some of which may be independent and some could be part of a group. GIL_i specifies the total inconsistency that can be contributed by operations on objects within a particular group i. Groups in turn can contain subgroups. Thus the total inconsistency that is viewed by the transaction is the sum of the inconsistencies from the operations that access the independent objects and objects that are parts of groups. Here is a query specifying hierarchical inconsistency bounds[2] (the syntax of the operations are explained later).

```
BEGIN Query TIL 10000
   LIMIT company 4000
   LIMIT preferred 3000
   LIMIT personal 3000
   LIMIT com1 200
   ...
   t1=Read com2745
   t2=Read com4639
   ...
END
```

Thus each transaction has an inconsistency specification part at the beginning before the actual operations are submitted. The inconsistency could be specified as an absolute value or as a percentage of the object values. It should be mentioned that hierarchical specification and control does not come free of charge and a (small) price is to be paid for it as we discuss in section 5.

3.2 Two Level Specification used in our System

3.2.1 Transaction Level

Throughout our discussion, for simplicity, we assume that an object is read or written once within a transaction and that the transaction calculates the sum of the read values. To allow multiple operations on the same object in a transaction, a simple extension to the control mechanism should suffice. In section 5, we discuss the effect of inconsistency on transactions that compute results other than the sum.

Imported Inconsistency:

Since TIL denotes the total inconsistency that can be tolerated for a query,

$$\sum_{i=1}^{k} Inconsistency\_from\_Read_i \leq TIL_{Q_i}$$

where the left hand side gives the total inconsistency imported, assuming that the query Q_i has k read operations. In this section, wherever appropriate, we have included examples of the query/update ETs used in the evaluation. The following is a query ET. The Read operation takes the *id* of the object whose value is to be read. The BEGIN operation takes the *type* of the transaction and the corresponding inconsistency bound as the arguments.

```
BEGIN Query TIL = 100000
   t1 = Read 1863
   t2 = Read 1427
   t3 = Read 1912
   t4 = Read 1543
   output('Sum is :  ',t1+t2+t3+t4)
COMMIT
```

Exported Inconsistency:

TEL denotes the total exported inconsistency that can be tolerated for an update ET. Since TEL denotes the total inconsistency that can be tolerated for a update,

$$\sum_{j=1}^{k} Inconsistency\_from\_Write_j \leq TEL_{U_j}$$

where the left hand side gives the total inconsistency exported, assuming that the U_j has k write operations. An update ET submitted to the system for processing is shown below. Note that in this, the value of the writes are dependent upon the reads and hence reads from update ETs must be consistent. The argument for the Write operations are the *id* of the object and the *value* to be written.

```
BEGIN Update TEL = 10000
   t1 = Read 1923
   t2 = Read 1644
   Write 1078 , t2+3000
   t3 = Read 1066
   t4 = Read 1213
   Write 1727 , t3-t4+4230
   Write 1501 , t1+t4+7935
COMMIT
```

3.2.2 Object Level

We use a client-server model for our implementation and usually transaction limits are specified at the client side and object limits at the server side, but this could be overridden by explicitly specifying the object limits in the specification stage at the beginning of the transaction.

Imported Inconsistency:

OIL_x is defined as the maximum amount of inconsistency that can be viewed (imported) by read operations of a query ET on an object x.

$$Inconsistency\_from\_Read(x) \leq OIL_x$$

Exported Inconsistency:

OEL_x is defined as the maximum amount of inconsistency that can be exported by the write operations from an update ET on an object x.

$$Inconsistency\_from\_Write(x) \leq OEL_x$$

A system that implements ESR should ensure that the various inconsistencies are within the bounds. In our system which implements a two-level hierarchy, for an operation to succeed,

at the object level:

$object\_import\_inconsistency \leq$ $object\_import\_limit$ (Read from Query)

and $object\_export\_inconsistency \leq$ $object\_export\_limit$ (Write from Update)

at the transaction level:

$transaction\_import\_inconsistency \leq$ $transaction\_import\_limit$ (Read from Query)

and $transaction\_export\_inconsistency \leq$ $transaction\_export\_limit$ (Write from Update)

[2] For explanation purposes we provide examples of *simple* Query and Update transactions. However more sophisticated examples could be considered.

4 Timestamp based ESR

Our goal was to build a simple prototype system to measure the performance of ESR — especially its impact on concurrency. We chose timestamp ordering for concurrency control. To simplify recovery, we enforce *strict* ordering [2]. Though deadlocks are rare, we detect them using the RPC timeout mechanism. Out of order operations which cannot be executed are aborted and the transaction is resubmitted after some delay. Hence we don't have the need to maintain detailed histories/logs since recovery is simple and rollbacks are not necessary. Recall that timestamps are assigned when transactions begin.

We do not go into the details of timestamp based SR, but all standard terminologies and techniques are applicable here. The algorithm used here is an enhanced version of the timestamp ordering algorithm used to implement SR. The enhancements come in places where the algorithm would normally abort the transaction because the operation is late or some other conflicting operation is running concurrently that prevents the current operation from being executed. The detailed algorithm for implementing ESR using time stamp ordering is shown in *Figure 3.* The enhancements that have been incorporated to implement ESR have been "boxed". In the ESR case, the operations are allowed to take place, provided the inconsistency is within the inconsistency (epsilon) bounds. In Section 5 we discuss in detail how to check whether the inconsistency viewed by an ET has crossed the inconsistency bounds. Referring to *Figure 3*, it can be seen that there are three cases where operations that will normally be rejected in SR will be given a chance to perform under ESR.

```
if (transaction_type == Query) {
        if (object.committed == True) {  /* committed value */
                if (transaction_t–stamp >= object_last_write_t–stamp) {
                        Allow Read to be performed
                }
                else {    /* tardy read from query transaction */
                  if (epsilon bounds are not violated)
                          Allow Read to be performed
                  else
                          ABORT transaction
        }
        else {    /* will see uncommitted value ; concurrent update */
          if (epsilon bounds are not violated)
                  Allow Read to be performed
          else
                  WAIT (Delay operation till value is committed)
        }
}
else {   /* it is an update transaction */
        if (operation_type == Read) {
           if (object.committed == True) {  /* committed value */
                  if (transaction_t–stamp >= object_last_write_t–stamp) {
                        Allow Read to be performed
                  }
                  else  /* tardy read from update transaction */
                        ABORT transaction
           }
           else {    /* will see uncommitted value ; concurrent update */
                  WAIT (Delay operation till value is committed)
           }
        }
        else {   /* it is a Write operation */
            if (object.committed == True) {  /* committed value */
                  if (transaction_t–stamp >= object_last_read_t–stamp) {
                        if (transaction_t–stamp >= object_last_write_t–stamp) {
                                Allow Write to be performed
                        }
                        else {   /* tardy write with respect to last write */
                              Neglect this Write and return success
                              /* Thomas's Write Rule */
                        }
                  }
                  else {     /* tardy write with respect to last read */
                        if (object.last_read–type == Query) {
                          if (epsilon bounds are not violated)
                              Allow Write to be performed
                        }
                        else {   /* last read was from an update */
                              ABORT transaction
                        }
                  }
            }
            else {    /* will see uncommitted value ; concurrent update */
                  WAIT (Delay operation till value is committed)
        }
}
```

Figure 3: Algorithm for Time Stamp based ESR.

The first two cases correspond to reads from query ETs and the last one corresponds to a write from update ETs, as mentioned below:

1. A query read views committed data but the query ET's timestamp is older than the object's last-write timestamp.
2. A query read views uncommitted data from a concurrent update ET.
3. A write from an update ET arrives with a timestamp that is older than the object's last-read timestamp and the last read was from a query ET.

Case 1 can also be considered to occur due to a concurrent update because if the read operation has an older time-stamp and it read a committed value with a newer timestamp then the two transactions must have been concurrent at some point of time. A similar argument applies to case 3. (The last read is required to be from a query ET because we allow only consistent update transactions and hence, reads from one update transaction will conflict with writes from another update transaction.)

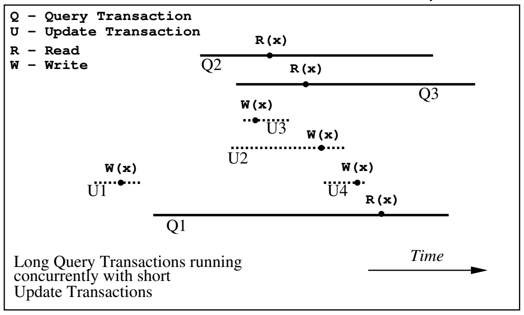

Figure 4: Time Stamp ordering for ESR — ETs with conflicts on object x

Figure 4 gives some examples of the three cases of concurrent updates discussed above. The first occurs when $R_{Q1}(x)$ (read from query $Q1$) comes after the value written by $W_{U4}(x)$ has committed, the second when $R_{Q2}(x)$ views uncommitted data written by $W_{U3}(x)$ and the third when $W_{U2}(x)$ needs to write when $R_{Q3}(x)$ has already read it. In all these cases the reads from query ETs view inconsistent data states because of out of order timestamps. The magnitude of this inconsistency must be controlled within the specifications. In the previous section we discussed ways to specify the allowable inconsistency. The following section deals with the mechanisms to meet these specifications.

5 Controlling Inconsistency

Throughout this discussion we have seen that in the ESR case, for an operation to succeed, the inconsistency should be controlled within the specified bounds. In this section we describe the timestamp based mechanisms we have used to control the inconsistencies. Some methods suggested in the literature [8], where the total inconsistency of a query ET is determined by multiplying the bound of an operation and the number of operations in the ET, may result in the overestimation of the accumulated errors. We present detailed mechanisms for controlling the inconsistency at fine grain levels.

Recall that to keep the prototype system simple and to perform controlled experiments, we have a two level specification and we look at only one type of query, those that compute the *sum*. At the end of the section, we address the issues related to controlling inconsistency when there is a hierarchical specification and when there are queries of other types.

5.1 Imported Inconsistency

Let us consider the scenario shown in *Figure 5.* We will focus on the read operation $R_{Q1}(x)$. For $R_{Q1}(x)$, the *proper* value is the value written by $W_{U1}(x)$ as that was the last write on x before $Q1$ started (which we will call $\mathcal{P}1$). $U2$, $U3$ and $U4$ are the update ETs that run concurrently with $Q1$. If $R_{Q1}(x)$ is successful it will read the value written by $W_{U4}(x)$ which is the *present* value (which we will call $\mathcal{N}4$). In the figure, d ($=$

$\mathcal{N}4 - \mathcal{P}1$) denotes the difference[3] between the *present* and the *proper* values and hence is the *magnitude* of the inconsistency in the data that is being read by $R_{Q1}(x)$ (*Inconsistency_from_read*).

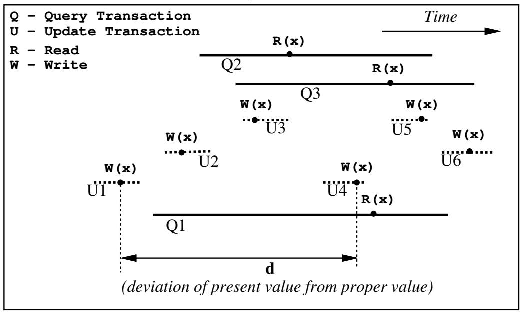

Figure 5: ESR: Check for Read (from Query ET) to go through

If the magnitude of the total inconsistency imported by $Q1$ before $R_{Q1}(x)$ is denoted by $\mathcal{I}$ then for $R_{Q1}(x)$ to succeed, the following must be satisfied in our two level system:

$$d \leq OIL_x \quad \text{(object-level)}$$
$$\mathcal{I} + d \leq TIL_{Q1} \quad \text{(transaction-level)}$$

If $R_{Q1}(x)$ is successful, $\mathcal{I}$ is incremented by the value d and Q1 proceeds, else it fails and $Q1$ has to be aborted.

If changes made by W_{U4} were negative, and $U4$ aborts then we could have a problem. The *current* value would then correspond to the one written by $W_{U3}(x)$ which could violate the inconsistency bound. One solution to this would be to always add the maximum change by an update transaction [7] while determining the inconsistency seen by a read.

In our implementation we store the values of the last 20 writes on each object with the corresponding time stamps. The *proper* value of an object is found by indexing backwards through this list until a older timestamp (than the query) is found. 20 is an empirical figure derived by dividing the measured values of the average duration of query ETs by that of update ETs.

It should be noted that this is not the same as multiversion timestamp ordering (MVTO). In the MVTO case, timestamped versions are maintained so that if a read operation arrives late, based on the versions, the value written by the last write with a timestamp lesser than this read is returned. However in our case, the value read is the value of the current instance of the object which is the *present* value. The value written by the last write with a timestamp lesser than this read, the *proper* value, is only used in determining the amount of inconsistency viewed by the read operation. We designed it this way so that the query ETs return a value that is meaningful in the current context (rather than returning an outdated value). We return to this issue in the concluding section.

5.2 Exported Inconsistency

We now move to *Figure 6*, focusing our attention on $W_{U5}(x)$. $Q1$, $Q2$ and $Q3$ are three query ETs that are running concurrently with update ET $U5$. $W_{U5}(x)$ exports some amount of inconsistency to those concurrent query ETs that have accessed x. The *proper* values of x for $R_{Q1}(x)$ is the value written by $W_{U1}(x)$; for $R_{Q2}(x)$ and $R_{Q3}(x)$ it is the value written by $W_{U2}(x)$. For each object x, we maintain a list of uncommitted query ETs which have read its value, along with the respective *proper* values (in practical environments, a better mechanism could be used). Let us denote these values as $\mathcal{P}1$, $\mathcal{P}2$ and $\mathcal{P}3$ for object x corresponding to $Q1$, $Q2$ and $Q3$ respectively. Let $\mathcal{N}5$ denote the *new* value that will be written by $W_{U5}(x)$ if it is allowed to execute. It should be noted that in this particular example, $\mathcal{P}1$ and $\mathcal{P}2$ happen to be the same. Since there are many concurrent query ETs, $W_{U5}(x)$ would export some inconsistency to each of these. Hence the inconsistency that will be exported by $W_{U5}(x)$ denoted by d is chosen as the maximum of the inconsistencies that will be exported to the concurrent query ETs (*Inconsistency_from_write*). In this case, it is the higher of $|d1|$ and $|d2|$ whose values are ($\mathcal{N}5 - \mathcal{P}1$) and ($\mathcal{N}5 - \mathcal{P}2$) respectively. Our notion of *Inconsistency_from_write* differs from [8] at this point. In [8] d is determined by adding the inconsistencies exported to all the concurrent query ETs.

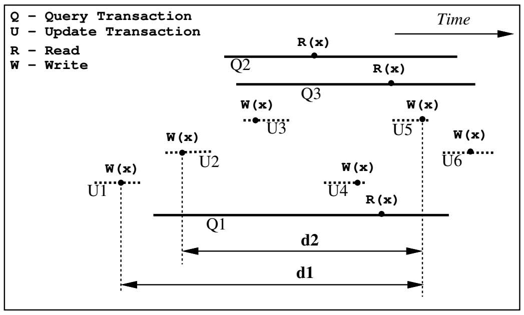

Figure 6: ESR: Check for Write to go through

If the magnitude of the total inconsistency exported by $U5$ before $W_{U5}(x)$ is denoted by $\mathcal{E}$ then for $W_{U5}(x)$ to succeed, the following must be satisfied in our two level system:

$$d \leq OEL_x \quad \text{(object-level)}$$
$$\mathcal{E} + d \leq TEL_{U5} \quad \text{(transaction-level)}$$

If these conditions are satisfied then $W_{U5}(x)$ is executed and $\mathcal{E}$ is incremented by the value d and $U5$ proceeds, else it fails and $U5$ is aborted.

5.3 Other Issues

5.3.1 Controlling Inconsistency in a Multi-level Hierarchy

Additional checks are required to determine if an operation can execute without violating the hierarchical inconsistency bounds. In a multi-level hierarchy, during the control stage the information flows bottom-up *i.e.* the inconsistency viewed by an object at the leaf level percolates to the root level through intermediate levels. As it percolates, the inconsistency is checked at the various nodes it passes through. Revisiting the banking system shown in *Figure 1*, the following set of checks are required to maintain the inconsistency within the bounds (if the inconsistency in the value read for a division k of company j is d).

$$d \leq Limit_{div_k} \quad \text{(leaf level)}$$
$$Inconsistency_{com_j} + d \leq Limit_{com_j} \quad \text{(level 3)}$$
$$Inconsistency_{company} + d \leq Limit_{company} \quad \text{(level 2)}$$
$$Inconsistency_{overall} + d \leq Limit_{overall} \quad \text{(root level)}$$

If there is no violation at any of the levels, then the total inconsistency accumulated at each of the levels is incremented (by making another pass through the hierarchy or some other means) by d. If the bounds are violated at any stage, the operation is unsuccessful and the transaction has to be aborted.

5.3.2 Controlling Inconsistency in Other Types of Queries

So far the mechanism we have explained is sufficient if all the queries involve *sum*. Though it is not easy to

[3]In [7] d is defined as $distance(x_{current}, x^{Q}_{initial})$.

specify general mechanisms for all kinds of queries we show how this can be done for *average*. In this discussion we will be concerned only with query inconsistency limits and not with object limits as the criterion for object inconsistency is going to remain unchanged. We also assume that an object could be read any number of times during a transaction. We have not implemented this in our system as we were working only with the *sum* operation and hence the performance results hold.

The mechanism would work as follows. For every transaction, we maintain the maximum and minimum values viewed, for each object it accesses. When an aggregate operation is encountered, the actual inconsistency is calculated based on the object's maximum and minimum values viewed, *i.e.*, if the operation is $avg(o_1,..,o_n)$ then the *avg_min_result* is determined by summing up the minimum values of $o_1..o_n$ and dividing it by n and the *avg_max_result* is obtained similarly. Then the *result_inconsistency*[4] is half the difference between *avg_max_result* and *avg_min_result*. The *result_inconsistency* is compared with the TIL. It is at this point a decision is made whether to reject or execute the operation as opposed to dynamically doing it when objects are read, as was done in the previous discussion. This is a viable solution to this problem as predeclaration of objects to be accessed or number of operations in a query may not be practicable.

As we saw above, there are some overheads in maintaining the various data structures and moving information around for hierarchical inconsistency control. However, by paying a small cost, the user gets some flexibility in deciding the precision of the query results.

6 Details of the Prototype System

We used the client server model for our implementation. Multiple clients submit transactions to a central transaction server. The server and each of the clients run on different DECstation 5000's running Unix, all connected in the same LAN. Since the environment supported threads and RPC library routines, the implementation of the prototype system was much simplified. There were 10 workstations in the LAN and because of the synchronous nature of our RPC, the maximum multiprogramming level is limited to 10. The prototype system processes about 50 to 60 transactions[5] per second, with each transaction having an average of 10 operations.

The system supports the five basic operations *Read*, *Write*, *Begin*, *Commit* and *Abort*. The recovery part of the system is very simple and is just sufficient enough to recover from aborted transactions. This is so because (i) we have strict ordering of operations and hence do not have any commit/abort dependencies (ii) we use the shadow paging technique while updating the value of the data items, so that if a transaction aborts, instead of rollbacks, all the data items are restored to their previous values and the transaction is started with a new timestamp.

The clients are supplied with data files consisting of a number of transactions that are randomly generated, to serve as the load of transactions. The clients read transactions from the file and submit operations to the server successively until all transactions have been processed. If a transaction is aborted the client keeps resubmitting it with a new timestamp, until it is successfully completed. The server primarily consists of a scheduler, a transaction manager and a data manager. The scheduler, which acts as the front-end, receives transaction requests from the clients and schedules the operations based on timestamp ordering by submitting it to the transaction manager. The transaction manager passes the operation to the data manager and based on the return value the transaction manager takes the appropriate action. The transaction manager uses many other data structures, like counters that are required for maintaining inconsistency accumulated by transactions etc. The data manager deals with the maintenance of object inconsistency. Objects are defined in a simple way, each has an *id*, a *value* associated with it, and the respective OIL and OEL. The database is maintained in the main memory on the server side and hence writing an object is simulated by changing its value in memory. This has been done to avoid overheads and complications in recovery if data is stored in files. When the server is invoked, it initializes all the objects by reading the start-up data file. The object limits are actually defined at the server side. These values can be set by the application designer/programmer. All the data required for evaluating the performance are available at the server.

7 Objectives of the Performance Tests

The main focus of the first set of tests was on measuring the actual increase in concurrency, resulting from the use of ESR, over SR in terms of the throughput for various MPL (multiprogramming levels) and different ranges of inconsistency bounds. We also studied some related metrics like the number of retries(aborts), the total number of operations performed (number of reads and writes) and the number of inconsistent operations that succeed (even after viewing some inconsistency) at various MPL and inconsistency bounds. The tests were performed for *high*, *medium*, *low* and *zero* values for the bounds (both TEL and TIL). During these tests we kept OIL and OEL constant at high values so that they do not affect the results. As the bounds on inconsistency increase, we would expect the throughput to increase.

The thrashing point (the MPL where the throughput begins to drop) is highly dependent on the conflict ratio. For example, for the conflict ratios considered in [1], thrashing occurs in most cases when the MPL is around 30 or 40. However in our case we had to use a higher conflict ratio so that we could observe thrashing at a lower MPL (within 10 to be precise). This has a side effect of producing reduced overall throughputs. Though we had about 1000 objects defined in our database, most of our transactions accessed only about 20 objects thus creating a high conflict ratio. The following is the mean magnitude of the various bounds on inconsistency we used for the first set of tests.

Level	TIL	TEL
high-epsilon	100,000	10,000
medium-epsilon	50,000	5,000
low-epsilon	10,000	1,000

Typical query ETs have about 20 operations per transaction while update ETs have around 6 operations per transaction. Hence the TEL values are lower than the TIL values.

In the second set of tests, we studied the effect of different TIL values on system performance, while maintaining the TEL at various constant levels. We also studied the effect of OIL on the throughput of the system.

8 Results and Discussion

The tests were repeated a few times to eliminate any disturbances caused by the workstation load and network traffic in the LAN. Though we do not include any information about confidence intervals in our graphs, the 90 percent confidence intervals lie within $\pm 3\%$ percentage points of the mean value of the performance metrics shown in the various graphs. In the graphs, *epsilon* refers to the inconsistency bounds. Due to space limitations we

[4]This is explained in more detail in [7].

[5]The number of conflicting operations during these tests were small.

show only some of the results here. Others can be found in [4]. In *Figure 7* we see that at higher epsilon values, the throughput with ESR is much higher than with SR. As the bounds decrease in value, ESR starts approaching SR. Another key observation that we can make is the shift in thrashing point to higher MPL with increase in epsilon. Thrashing occurs at a MPL of 3 in the case of lower bounds and it shifts to a MPL of 5 at higher bounds.

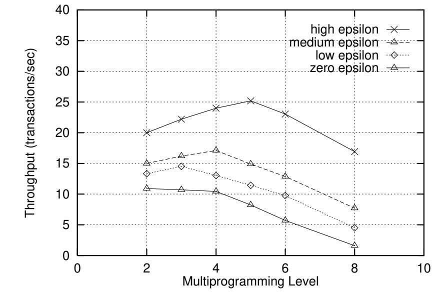

Figure 7: Throughput vs MPL

This significant shift of the thrashing point can be explained by studying some of the other performance metrics. These give us a picture of the dynamics that take place internally when the transactions are processed. The results (not included here) showed that the number of inconsistent operations that are successful increases with an increase in epsilon and the MPL.

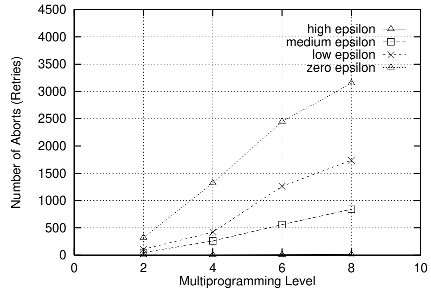

Figure 8: Number of Aborts vs MPL

Also, examining the number of aborts (retries) at various levels (*Figure 8*), we note that the number of aborts at high inconsistency bounds is almost *zero*. We see that at lower bounds the number of aborts shoots up rapidly and for the case of *zero epsilon* (SR) the number is very high. Now let us look at the total number of operations performed, which is the sum of the read and write operations. This is very useful because it gives an indication of the effort that is wasted by the system, when transactions abort, thereby affecting the throughput.

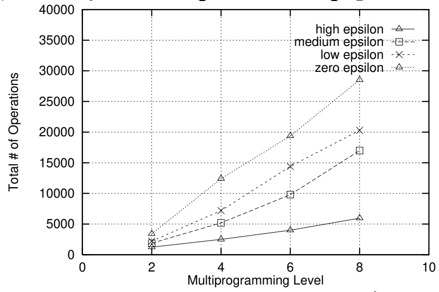

Figure 9: Number of Operations (R+W) vs MPL

Figure 9 gives the total number of operations executed at various MPL values. Since we noted in the previous graph that the number of aborts is practically *zero* for the case of high inconsistency bounds, the total number of operations performed in this case is the same as the actual number of operations required by the transactions. Anything above this gives a measure of the number of useless operations executed by the system. All these graphs give us some insight into the quantitative performance improvement that can result from ESR.

Now we come to our next set of tests where we study the effects of the imported inconsistency bounds *i.e.* TIL and OIL. All these tests are for a MPL of 4. *Figure 10* shows that when TEL is held at a constant level, the throughput increases with an increase in TIL.

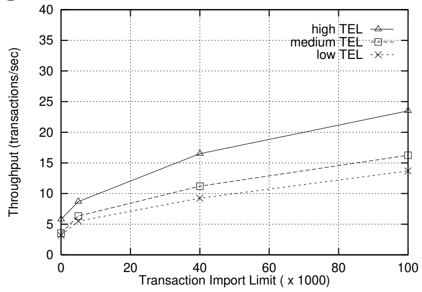

Figure 10: Throughput vs TIL

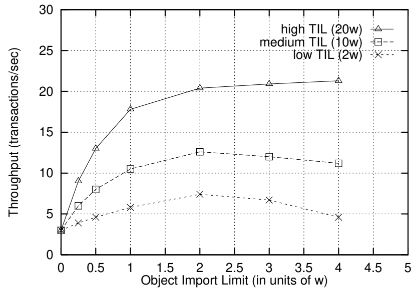

Figure 11: Throughput vs OIL

We made several interesting observations in our study of the affect of OIL on the throughput. Apart from object and transaction inconsistency limits, the average change in value due to a write also affects the throughput. Let the average change in value due to a write be denoted by w. In these graphs, instead of mentioning the absolute value of the bounds, we parameterize it, and specify it in terms of w. *Figure 11* shows the variation of throughput with respect to OIL. The interesting observation here is that in the case of low to medium TIL, the throughput is low at both low and high values of OIL but higher at medium values of OIL. At low OIL, very little inconsistency is tolerated and the throughput is low. At *zero* OIL, no inconsistency is tolerated on any operation and hence the throughput corresponds to the SR case.

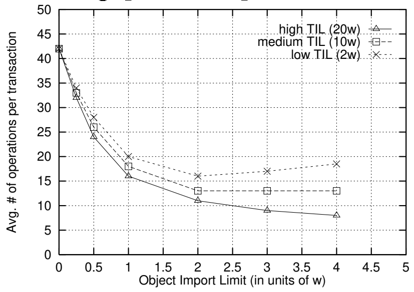

Figure 12: Average number of operations per transaction (TIL varies)

Figure 12 goes into the details that help explain this observation. It gives the average number of operations that are executed for a transaction to complete and this includes the number of operations executed in aborted transactions (which is essentially a waste). Here OELs are maintained at constant high values. For a moment if we ignore TIL, as OIL increases, the number of operations per transaction should keep decreasing and that is what we see in the case of high TIL. However if we consider TIL, we see its effect slowly creeping in as OIL increases. In fact, for the low TIL case after a certain value of OIL the number of operations per transaction increases. The drop in the throughput at low TIL when the OIL is high can be attributed to the fact that more number of operations are executed by an aborted transaction in this case before it is aborted. The reason is as follows. Some operations that cause high inconsistency are not allowed by medium OIL. As a result, the operation fails and the transaction is aborted and resubmitted again. However in the case of high OIL these operations which view high inconsistency are allowed to pass through. In most cases, due to such high inconsistency operations, the total inconsistency viewed by the transaction drastically increases and because of the low TIL the transaction inconsistency bound is violated and the transaction has to be aborted. Thus the throughput is lower because of the wasted efforts in executing more operations per aborted transaction. Even though the curves indicate that this happens only at low values of TIL, we conjecture that for high TIL also the same would have been observed had we tested with even higher values of OIL. In practice, the object inconsistency limits have to be selected by the application designer/programmer based on the range of values the objects can take and the desired TILs.

In summary, we see from the various performance curves that the performance of ESR improves with increase in the transaction inconsistency bounds. Most performance tests yielded the expected results. Interesting observations include the shift in the thrashing point and affects of object inconsistency bounds on performance at different TILs. The behavior of a system with more levels in the hierarchy will certainly be more complex than what we have seen. Though we expect a similar type of interaction between the bounds, an experimental verification would strengthen the claim. It should also be mentioned that our overheads for version management was low as we used a main memory database. In disk-based databases, the throughput could drop by some amount but the performance of ESR should still be much better than SR. The actual quantitative performance improvement in an application environment would depend upon the nature of the applications, the typical conflict ratio in those environments, etc. In practice, there has to be a compromise between the desired accuracy of the results and the throughput.

9 Conclusion

Previous literature on ESR specified inconsistency bounds with respect to transactions [8] or objects [7]. We have introduced the notion of hierarchical inconsistency bounds that allows inconsistency to be specified at different levels. Using a practical example we demonstrated that with this, a user could gain more flexibility in specifying the inconsistency viewed by ETs. We have also presented detailed mechanisms to control the inconsistency seen by ETs within the specified bounds at different levels of a hierarchy. Our notion of hierarchical inconsistency control attempts to provide a finer-grained approach to controlling inconsistencies in ETs than before [8].

Several notions of correctness weaker than SR have been proposed previously but very few have been evaluated [6]. Since no work has been done to determine the performance gains from ESR, we have presented an evaluation of the quantitative performance improvements resulting from ESR, obtained through a series of tests on our prototype system. For simplicity, we used (i) a two-level hierarchy for inconsistency specification and control, and (ii) queries that calculate the *sum* of the values read. Apart from studying the throughput, we studied many other performance metrics (like number of aborts and number of successful inconsistent operations) that gave us a picture of the dynamics that take place internally when the transactions are processed. While studying the performance of ESR, we made a couple of important observations. First, the thrashing point shifts to a higher multiprogramming level when inconsistency bounds are raised. Second, at a particular multiprogramming level, for a given transaction inconsistency bound, the peak throughput occurs at some intermediate value of the object inconsistency bound rather than at low or high values. This shows that in practice, there has to be a compromise between the desired accuracy of the results with respect to individual objects and the throughput. We are currently extending our algorithm and the implementation to do multiversion timestamp ordering since we can then make further use of the version that are being maintained anyway. Another possible extension is to evaluate ESR in the case of a distributed system with data replication.

Acknowledgements

The Digital Equipment Corporation sponsored Project Pilgrim at the University of Massachusetts provided us the environment for conducting the performance tests. We thank the anonymous referees for their suggestions.

References

[1] R. Agrawal, M.J. Carey and M. Livny, "Concurrency control performance modeling: Alternatives and implications", *ACM Transactions on Database Systems*, 12(4): pp 605-654, December 1987.

[2] P.A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publishing Company, first edition, 1987.

[3] T. Haerder and A. Reuter, "Principles of transaction oriented database recovery", *ACM Computing Surveys*, 15(4); pp 287-317, December 1983.

[4] M. Kamath and K. Ramamritham, "Performance Characteristics of Epsilon Serializability with Hierarchical Inconsistency Bounds", *Technical Report 92-46*, Department of Computer Science, University of Massachusetts, June 1992.

[5] C. Pu and A. Leff, "Replica Control in distributed systems: An asynchronous approach", In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pp 377-386, Denver, May 1991.

[6] K. Ramamritham and P. K. Chrysanthis, "In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties", *Distributed Object Management*, edited by Özsu, Dayal, and Valduriez, Morgan Kaufmann, 1992.

[7] K. Ramamritham and C. Pu, "A Formal Characterization of Epsilon Serializability", *Technical Report 91-92*, Department of Computer Science, University of Massachusetts, Dec. 1991.

[8] K.L. Wu, P.S. Yu and C. Pu, "Divergence Control for Epsilon Serializability", In *Proceedings of Eighth International Conference on Data Engineering*, pp 506-515, Phoenix, February 1992.