Quantifying Isolation Anomalies

Alan Fekete
School of Information Technologies
University of Sydney, Australia
fekete@it.usyd.edu.au

Shirley N. Goldrei
Didco Systems
shirley@dynalogics.com.au

Jorge Pérez Asenjo
University of Sydney, Australia
esyorcho@yahoo.es

ABSTRACT

Choosing a weak isolation level such as Read Committed is understood as a trade-off, where less isolation means that higher performance is gained but there is an increased possibility that data integrity will be lost. Previously, one side of this trade-off has been carefully studied quantitatively – there are well-known metrics for performance such as transactions per minute, standardized benchmarks that measure these in a controlled way, and analytic models that can predict how performance is influenced by system parameters like multiprogramming level. This paper contributes to quantifying the other aspect of the trade-off. We define a novel microbenchmark that measures how rapidly integrity violations are produced at different isolation levels, for a simple set of transactions. We explore how this rate is impacted by configuration factors such as multiprogramming level, or contention frequency. For the isolation levels in multi-version platforms (Snapshot Isolation and the multi-version variant of Read Committed), we offer a simple probabilistic model that predicts the rate of integrity violations in our microbenchmark from configuration parameters. We validate the predictive model against measurements from the microbenchmark. The model identifies a region of the configuration space where a surprising inversion occurs: for these parameter settings, more integrity violations happen with Snapshot Isolation than with multi-version Read Committed, even though the latter is considered a lower isolation level.

1. INTRODUCTION

Concurrency control is an essential component in any DBMS platform, and many different algorithms are known [24]. The "gold standard" for concurrency control is to provide serializability, which has the valuable feature that any integrity constraint will continue to hold in the database, provided it is true initially, and each transaction is coded to preserve the constraint when run alone. Unfortunately, the algorithms implemented in most platforms that ensure serializable execution can have detrimental impact on performance in cases of contention. These algorithms involve strict two-phase locking (2PL), with locks, in shared or exclusive (or other) modes being taken at various granularities covering data items as well as on indices, and these locks are held till the end of the transaction.

From the earliest implementations like System R, it was realized that many developers would desire better throughput under contention, and so systems offer the option of running a program at weaker isolation levels, where some of the locks, needed for serializability, were released early, or even not taken at all. For example, the Read Committed isolation level is provided by releasing any shared lock once the transaction has finished reading the item concerned. This prevents a transaction reading dirty data, but it does allow a transaction that reads several items to see information from inconsistent states.

More recently, several prominent platforms including Oracle and PostgreSQL have used multiversion concurrency control algorithms without any locks for reading. The Snapshot Isolation (SI) mechanism [3] makes each read see the version of the data that had committed most recently before the transaction started (that is, a read skips over versions that committed after the reading transaction started); any transaction's writes are not visible to concurrent transactions. As part of SI, the First Committer Wins rule prevents lost updates, by forcing a transaction to abort if a concurrent transaction has already committed, where both transactions wrote to the same item. In multiversion systems, one can also provide isolation in which each transaction sees only committed data, but inconsistent states may be observed. We call this algorithm RC_MV or multiversion Read Committed[1]; here each read sees the version that was committed most recently *at the time of the read*, by skipping over any as yet-uncommitted versions. A write will be blocked till after the commit of the previous version of the item[2].

In commercial practice, weaker isolation levels such as Read Committed are frequently used; most platforms (whether they are based on locking or multiversion approaches) have this as the default for concurrency control. It is widely described as allowing the application developer to take advantage of domain knowledge which means that their particular code will work correctly with low isolation; however, there

VLDB '09, August 24-28, 2009, Lyon, France

[1]Some Oracle documentation and [3] use the term Oracle Read Consistency for the RC_MV algorithm.
[2]Unlike what happens with SI, under RC_MV, a write does not force the transaction to abort in case a concurrent transaction has already committed changes to the same item.

is no set of principles or guidelines to determine whether this is so for given code. [17] reported examples of deployed applications using SI which produced non-serializable executions. Thus in practice, what most developers actually see is a trade-off between performance and correctness. Running with strong isolation such as serializability will protect data integrity but limit throughput under contention; conversely, weak isolation offers better performance but with the risk that data might be corrupted by undesirable interleavings between concurrent activities.

A long-term goal of our research agenda is to allow developers to make this tradeoff *quantitatively*: they should know how much data corruption will be the consequence of a given improvement in throughput. The field has many ways to quantify performance; there are benchmarks to measure it, simulations to study it, even analytical models that can predict it. This paper proposes to give a number for the extent to which data corruption occurs. We define a microbenchmark that provides a single figure-of-merit, capturing the extent to which data integrity is violated in executions at various isolation levels. Some previous work [21] has measured aspects of the trade-off, using the percentage of queries that return incorrect answers, rather than the amount of corruption in the database state.

In Figure 1 we show an example of the use of our microbenchmark. We have measured the rate of violations of the integrity constraint at different isolation levels, on a particular platform, and for a range of configurations which vary only in the MPL (the number of concurrent clients that are submitting transactions). We have not shown on the graph any data from Serializable isolation (based on strict two-phase locking), because there will never be any violations in serializable executions[3]; similarly the particular transactions we use never select rows by a predicate (except for selection by primary key), nor are rows inserted or deleted during execution, and so Repeatable Read isolation also can not cause any violations. We clearly see confirmation of our expectation that the locking and multiversion forms of Read Committed produce many more violations than the rather strong Snapshot Isolation level. Also we observe a trend of approximately linear increasing violation rate as MPL increases (as a sanity check, we realize that MPL=1 has no concurrency in the server, and thus no violations, no matter what concurrency control is used). Note that the number of transactions run in a given time also increases approximately linearly with MPL, at least for low MPL, so the number of violations in the data after a given time will increase approximately quadratically with MPL.

The ANSI SQL standard provides for 4 isolation levels which the application developer can request. These levels are arranged in a hierarchy, with Serializable at the top, then Repeatable Read, Read Committed and finally Read Uncommitted. Each level is "defined" to rule out some particular anomalous behaviors which the lower levels allow. The standard does not however mandate particular implementation approaches, and there are some well-known flaws in the way the behaviors of each level are specified[3]. If we assume that the well-known locking-based concurrency control algorithms are used, then the hierarchy is valid. Each algorithm holds all the locks of the lower levels, and also some extra locks (or it holds the same locks for longer). Thus each locking algorithm permits a strict subset of the interleavings permitted by the locking algorithm at the level below, and so the higher levels will produce fewer violations of integrity than the lower levels.

[3]As a check on our code, we did measure some runs with 2PL for concurrency control; they reported no violations at all.

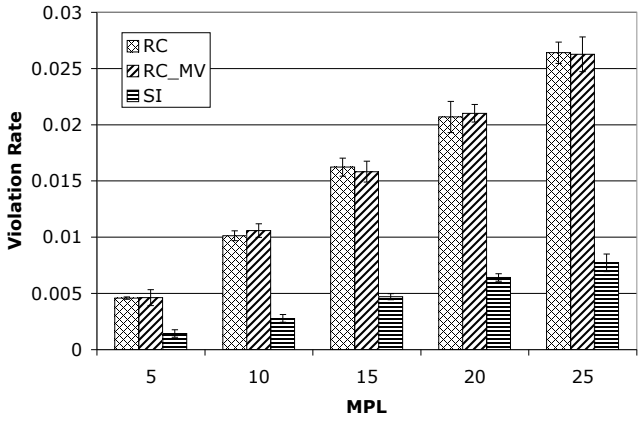

Figure 1: Example chart of measured Violation rate vs MPL.

However, as we saw, many platforms implement multiversion algorithms for concurrency control. Snapshot Isolation is treated as a strong algorithm (in particular, it prevents all the anomalies described in the ANSI standard as shown by [3], although it does allow an anomaly called Write-Skew). It is generally assumed that the multiversion Read Committed algorithm is weaker than SI, because it does allow anomalies like Inconsistent Reads and Phantoms. Indeed, several platforms use SI when an application is declared to be Serializable, and they use the multiversion form of Read Committed when the application requests the lower isolation level. Despite this, [3] shows that SI is not strictly less permissive than RC_MV. Consider the interleaving shown in Eq (1), where $r_X(A)$ denotes transaction X reading item A, c_Y is the commit of transaction Y, etc.

$$r_X(A)r_Y(A)r_Y(B)w_Y(B)c_Y r_X(B)w_X(A)c_X \qquad (1)$$

Under SI, the interleaving in Eq (1) is allowed, and when X reads B, it does not see the value produced by Y; instead, X sees the version of B as it was when X started. This is non-serializable, an example of Write-Skew[3]. However, under RC_MV, the same interleaving is serializable: when X reads B, it sees the version which was committed by B before the read occurred, and the execution is view-equivalent to running Y then running X. This suggests that SI may sometimes produce more violations than RC_MV; of course, there are also many interleavings where RC_MV is non-serializable, but SI is serializable. The question is: can we tune the configuration so that cases like the one in Eq (1) happen more often than the cases where RC_MV produces a violation while SI does not? If this happens we say there is an "inversion"; more violations occur at what is regarded as a higher isolation level, than at a lower isolation level.

The key contributions of this paper are:

- We have designed a measurement approach so we can determine, for a particular system configuration and isolation level, a single figure of merit which reports the rate at which transactions introduce violations of data

integrity into the database. We call our measurement a microbenchmark to stress that it is not intended as a realistic application for making purchase decisions, but rather is to be used to explore how different features of the configuration impact on the rate of violation. Section 2 gives details of the data and programs, the way measurements are done, and the main parameters of the configuration space.

- For the widely-implemented multiversion concurrency control algorithms, Section 3 introduces a mathematical model based on simple probability calculations, that predicts the rate of violations in our microbenchmark for any system configuration.
- We compare the predictions with measurements, across a range of parts of the total configuration space. In large part the results (presented in Section 4) validate the usefulness of our model, showing that our formulae mostly predict within 15% of the measured value. As well, these graphs show many trends that provide guidance in how different parameters of the configuration impact on the rate of violations.
- Based on the predictive model, we find that there are some (rare) combinations of configuration parameters, for which the rate of violation introduction is actually higher at the strong SI level, than at the weaker RC_MV level. We have used our microbenchmark to confirm the existence of this inversion, in Section 5.

These contributions combine to begin the *quantitative* study of how weak isolation actually damages the integrity of data. Finally, in Section 6 we give references to the most relevant prior work, and in Section 7 we conclude the paper and suggest directions for future work.

2. THE MICROBENCHMARK

The anomaly-measuring microbenchmark we designed follows closely the format of traditional performance benchmarks from Debit-Credit [2] onwards. There is a set of tables, and a set of transaction types that operate on those tables. The experiment begins by creating the appropriate tables, and loading data into them, to achieve a particular database size and a given distribution of contents. A number of client computations run, each of which repeatedly chooses a random transaction to run on the database engine (the choice is made according to some distribution on the transaction types, and also on a distribution on the parameters passed to that transaction type, which is set so that some part of the tables is a hotspot with high contention between transactions). The clients are run for a warm-up period, and then the measurement period starts, lasting a predetermined duration. After the measurement period has finished, the clients cease sending operations to the database engine and statistics are collected and written to a file. The whole run (with warmup then measurement) is repeated multiple times so that confidence intervals can be determined for the results. The rest of this section is devoted to describing the details of the microbenchmark, and also we discuss the issues that influenced our design.

We are designing a microbenchmark, whose purpose is to elucidate the impact of various factors on the rate of isolation anomalies (in contrast to a benchmark whose purpose is to evaluate an overall system on performance and cost). Thus, we do not need to have a database whose contents or transactions are in any way realistic or meaningful; rather they should be very simple so that different factors can be explored separately. The schema of our database has only two tables. TableA has 3 columns: id (an integer, the primary key), valueA (an integer) and description (a string, of variable length up to 100 characters). Similarly we have TableB(id, valueB, description). An undeclared integrity constraint is that for any given id, the sum of the corresponding values across the two tables must be between 0 and 99 inclusive. This is enforced as the data is loaded, by choosing a random sum in the allowed range, choosing a random valueA, and then setting valueB so that valueA+valueB is the particular sum.

```
BEGIN TRANSACTION
     SELECT @oldA = valueA
     FROM TableA
     WHERE id = @id

     delay for sleeptimeAB millisecs

     SELECT @oldB = valueB
     FROM TableB
     WHERE id = @id;

     delay for sleeptimeBU millisecs

     IF ((( @oldA + @oldB ) < 0 ) OR
               (( @oldA + @oldB ) >= 100))
          SET @delta = 0;
     ELSE
          IF (( @oldA + @oldB ) < 50 )
               SET @delta = (50 * @deltaMultiplier);
          ELSE
               SET @delta = (-50 * @deltaMultiplier);

     UPDATE TableA
     SET valueA = ( valueA + @delta )
     WHERE id = @id;

COMMIT TRANSACTION
```

Figure 2: Pseudocode for changeA

There are three transaction types, changeA, changeB and changeAB. Each transaction type has a parameter which is an id and controls which row of each table is touched; there are some other parameters as well, which control the waits between steps, and there is a parameter deltaMultiplier which is set to zero during the warmup period, so that data is changed only during the measurement interval. Each transaction type is given as a stored procedure in the database engine. The pseudocode for changeA is in Figure 2. ChangeB is similar except that the update is applied to TableB.valueB, while in changeAB half the calculated delta is added to TableA.valueA and half to TableB.valueB.

Note that when run alone, each of these transactions preserves the integrity constraint. If for the given id, the sum TableA.valueA+TableB.valueB is between 0 and 49, then delta is 50, so the sum ends up between 50 and 99; while if the sum is initially between 50 and 99 inclusive, it ends

between 0 and 49 when 50 is subtracted.

After the end of the measurement interval, we scan the tables and calculate the number of violations, that is, how many rows there are for which the integrity constraint does not hold. We also report the number of transactions that were submitted by all the clients during the measurement interval, and the number of transactions that committed. The main metric we are concerned with in this paper is the rate of introduction of violations, defined as the number of violations found at the end of the run, divided by the number of transactions that committed during the measurement interval. Note that the transactions are coded (with deltaMultiplier) so that no violations are ever introduced during warmup, so it makes sense to normalize to the transactions which committed during measurement interval. We do not intend to report on overall throughput or other performance measures; that aspect of the isolation tradeoff is very well covered by existing benchmarks.

A central design decision for our transactions is that we do not want to risk removing a violation once it has been produced. If a violation could be removed (by a transaction running alone, or by some interleaving of transactions), then counting the violations at the end would show merely the difference between rate of production of violations and the rate of their removal, and it would be hard to understand the impact of any factors that affect both rates in similar ways. Thus, once an id has sum of valueA and valueB which falls outside the permitted range, we want to keep the sum there; we achieve this by having all transactions leave the data unchanged once they observe a violation[4]. We make the transaction leave data unchanged through an update that adds zero, rather than by skipping the update, thus keeping the CPU and I/O demands steady during the execution.

This feature of our transaction design has an unfortunate side-effect on the meaningfulness of our measurements. Once integrity has failed for a given id, subsequent non-serializable interleavings of transactions on that same pair of rows will not introduce another violation. Thus, over time, as more violations build up in the data, the rate of new violations reduces; in the limit, once all rows of the hotspot have a violation, almost no further violations will be added, and the calculated rate will drop towards zero, as more and more transactions are committed. Thus we need to ensure that we calculate our rate in runs that are not too long. We discuss this point further, below.

The main choices that the client makes, each time a transaction is to be submitted, are the transaction type, the id (determining the row to be examined in each table, and perhaps altered) and the delays we introduce as simulated busytimes, one between read(valueA) and read(valueB), the other between read(valueB) and the updates. These choices are all randomized according to distributions parameterized by aspects of the system configuration.

- The transaction type is chosen from a discrete distribution: a fraction f_A of transaction instances have type changeA, and similarly fractions f_B are changeB, and f_{AB} are changeAB. Of course these fractions are dependent: $f_A + f_B + f_{AB} = 1$.
- The id of the affected rows is chosen from a piecewise uniform distribution. Two parameters of the configuration define the distribution on ids. H denotes how many rows of each table make up the hotspot, and F denotes the proportion of transaction instances that operate on the hotspot. Thus, for a fraction F of the instances, the client chooses an id uniformly within the hotspot; for the remaining 1-F fraction of transactions, the id is chosen uniformly outside the hotspot. The hotspot rows are equally spaced out, among the whole table. In future work, we would like to allow the hotspot rows to shift over time through the run, but so far we just make rows 1, 1+H, 1+2H etc as being in the hotspot.
- Each delay is chosen from a normal[5] distribution, whose mean and standard deviation are configurable.

To perform our measurements, we use a commercial database platform with stored procedures encoding the three transaction types. To submit requests we have a Java implementation, where a single "client" machine (different from the database server) hosts concurrent threads that represent the clients, repeatedly choosing a transaction instance to run, and invoking one stored procedure with appropriate parameters. At the start of a run, the Java program cleans up the database tables and ensures that the data has no violations; it then launches the threads, establishes a connection from each thread to the DBMS server, sets the appropriate isolation level for the connections; then after a warmup period[6], the main program sets a shared variable that tells each thread to begin counting transactions (also to set deltaMultiplier to 1, so that updates will begin to change the data in the tables). After the measurement period is over, the threads that represent the clients are stopped, and data is collected from each (eg how many transactions were submitted and how many committed; we also learn how many aborts were due to different causes). As well, the program runs a query that counts the rows where the sum is outside the allowed range. All the statistics are written to files after the measurement period is finished; another file keeps the configuration information that controlled the execution.

We discuss two aspects of the configuration space that were quite problematic because of practical considerations. The first concerns the length of the measurement interval.

As we noted above, the longer the measurement interval, the more the hotspot will fill with violations, and the more non-serializable interleavings that occur but are missed in our metric, because the conflicting transactions detect the violation and add zero to the values, instead of altering them again. We call this "under-reporting"; a simple estimate of its presence is to calculate the number of violations as a percentage of H, the size of the hotspot, and average this changing quantity over the duration of the measurement interval. Since the curve of violation count has decreasing slope, the time average is at least half the final count. Thus

[4]Of course, in principle, a transaction at low isolation may observe a violation even when there is none, through inconsistent reading of the tables.

[5]To be precise, since a delay can not be negative, we truncate the distribution at zero; to keep it symmetric, we also truncate the distribution so the maximum sleep time is twice the mean.

[6]We have a warmup period in order that during measurement, we can be sure that all threads are running evenly; typically thread start is staggered, so that effective concurrency is not as high as expected, for a short period at the start of a run.

we see that our measurements will generally give a measure of the rate of introducing violations that is low by about half the final percentage of violations in the hotspot. In our experiments we aim to keep the measurement interval small enough, that this discrepency is below 5% of the true amount; that is, we try to ensure that the final count of violations is less than 0.1 of the size of the hotspot. We reach this goal in all experiments reported in this paper, except for H=100 with RC_MV isolation, where the under-reporting can be estimated at about 13% of the true value.

However, there is a problem of precision with short measurement intervals. Usually, experimental work measures the same quantity in many runs, and then one averages the figure from the different runs to find a more precise determination of the true quantity; one uses the standard deviation of the measurements to report a 95% confidence interval about the calculated mean. Readers can then see differences between quantities as significant if each falls outside the confidence interval surrounding the other. We wish to follow this approach, but the usual way to obtain more precision (take many runs) does not work well for our microbenchmark. The number of violations in a run follows close to a binomial distribution with small probability[7], and this is approximately a Poisson distribution. The standard deviation of a large sample from this distribution stays constant as the size of the sample increases; in many of our experiments, we would see a confidence interval whose width is half the quantity being measured, no matter how many runs we took! Instead, to get a sufficiently precise estimate, we want to make each run involve many transactions. The well-known results for Poisson distribution show that to obtain a confidence interval of width 0.002 around a value near 0.01 (as most violation rates are, in our part of the configuration space) we would need to have each run involve about 10000 transactions. But so many transactions leads to placing violations on many rows (about 100), and contradicts our policy of controlling the amount of under-reporting.

We escape this dilemma by taking many short runs, and grouping them into a super-run. Rather than calculate the violation rate in each run, we take the sum of all the anomalies produced in all the runs of the super-run; we divide this by the total number of transactions committed among all the runs of the super-run. This ratio is the rate of violation introduction during the super-run, and constitutes one measurement of the quantity we seek[8]. We repeat the whole super-run a few times, and we report the mean of these super-run measurements; also we give a 95% confidence interval using the standard deviation among the super-run measurements. In our experiments, we have done 5 super-runs; mostly each super-run has 50 runs, and about 25000 transactions; the least transactions per super-run is 12500 for MPL=25. Each point on a graph in this paper is produced from a total of up to 3 real-time hours of execution.

Another pragmatic concern has led to our choices of values for the delays between steps, which we implement using the DBMS wait command to sleep for a duration. The values we used lead to transactions that are unrealistically long (600 ms in most of our experiments, from two sleeps of 300 ms each). Unfortunately, the database platform on which we are running is quite inaccurate in delivering the exact sleep time which is requested. Trials from the database console have shown that each sleep request falls short of the requested time, by rounding down to a multiple of a quantum which is about 15ms. Thus in our experiments, we have a standard deviation for the requested sleep time of at least 30ms, so there is reasonable variation among the actual sleep time (we want to avoid any chance of convoy effects which could happen if all transactions run for exactly the same time). This also forces the average sleep time to be significantly above 60ms, so truncation of the distribution at zero is not a serious worry. We can thus estimate the true average sleep time as about 8ms less than the mean we request.

In Figure 3, we show a summary of the configuration parameters. We also give here the range of values for each that we used in the experiments we report in Figure 1 and in Secs 4 and 5; but we note that our microbenchmark design is not limited to these particular ranges.

Above, we described the microbenchmark as we ran it. Many OLTP systems run with frequent cross-network communication within a transaction, and the client makes a separate call (eg through JDBC) for each SQL statement, rather than a single-shot call of a stored procedure. An advantage for us of the stored-procedure approach is that it makes behavior more predictable for the probability model in Section 3, since each extra message delay adds to the variability. An advantage of multiple JDBC calls would be that we could set delays that are more realistic as simulated busy time; we would not need to choose values that are large compared to the 15ms quantum in the implementation of the DBMS sleep command. We have done some measurements with a variant of our microbenchmark, with a separate JDBC call for each select or update statement, and interstatement delays smaller by one order of magnitude than those reported in this paper; we saw the same trends, including the existence of the inversion.

3. PREDICTIONS FOR MULTIVERSION ISOLATION LEVELS

In this section, we show a simple probability model that predicts the rate of violation introduction for a given configuration, in terms of the parameters that define the configuration. Our model deals with both the multiversion isolation levels: SI and RC_MV. We were inspired by Gray et al's predictions of deadlock rates [12, 11], and by the Elnikety et al model predicting performance of replication algorithms [7]. We illustrate our approach on a particular configuration (where MPL=10 and with SI) from the experiment of Figure 1, and then we present the general formula for isolation level SI. Afterwards, we deal with multiversion Read Committed isolation.

Throughout this section we will use greek symbols for quantities that cannot be set directly in the configuration; for example δ_{SC} will denote the average duration of a transaction, that is, the period from its start (at the server) till its completion (at the server). We break this up at the significant events within the transaction, into δ_{SA}, δ_{AB}, δ_{BU}, δ_{UC} as shown in Figure 4; for example, δ_{SA} is the average time from the transaction's start till the read of valueA, and δ_{BU}

[7]This is because each transaction has a particular small probability to introduce a violation.

[8]Another way to think about this, is to consider a super-run as an activity in which the measuring is interrupted periodically, and the anomalies so far are calculated, then the database is reset to a clean state, and we resume warmup and then measurement.

Symbol	Explanation	Range of values we used
Iso	The isolation level set for all transaction instances	2PL, SI, RC_MV, RC
MPL	Multiprogramming level; How many concurrent client threads are running	1 to 25
F	The fraction of transactions that access a row in the hotspot	0.9
H	How many rows in each table make up the hotspot	100 to 500
f_A	Fraction of transaction instances that are changeA	0 to 0.66
f_B	Fraction of transaction instances that are changeB	0 to 1
f_{AB}	Fraction of transaction instances that are changeAB	0 or 0.33
sleeptimeAB	Mean of distribution used to request delay between read(valueA) and read(valueB)	100ms to 900ms
sdevsleepAB	Standard deviation of distribution used to request delay between read(valueA) and read(valueB)	30ms to 120ms
sleeptimeBU	Mean of distribution used to request delay between read(valueB) and updates	100ms to 500ms
sdevsleepBU	Standard deviation of distribution used to request delay between read(valueB) and updates	30ms to 90ms
warmup	Time to run client threads before measurement interval	1s
MI	Measurement interval	30s or 50s

Figure 3: Configuration Parameters

is the average time from reading valueB till the update(s). Similarly, δ_{CS} will denote the average time from one transaction completing at the server, until the next transaction starts from the same client (this is the message time from server to client, plus the delay in the client while the result is recorded and then the next transaction is prepared, plus the message time back to the server). We will discuss below how we estimate the value for expressions that involve these indirect quantities, and which appear in our final formulae.

3.1 Isolation level = SI

When we consider the SI isolation level, we first think about an arbitrary pair of transaction instances X and Y, and we examine the way Y impacts on the outcome for X. We can give a prediction for the probability that X will produce a violation in the database, as the product of three quantities.

(i) The number of possible choices for Y; this is the number of transactions on other threads during the measurement interval, in our case there are 9 other threads, and each thread has one transaction at intervals of $\delta_{CS} + \delta_{SA} + \delta_{AB} + \delta_{BU} + \delta_{UC} = \delta_{CS} + \delta_{SC}$. Overall there will be $9 * MI/(\delta_{CS} + \delta_{SC})$ choices of Y.

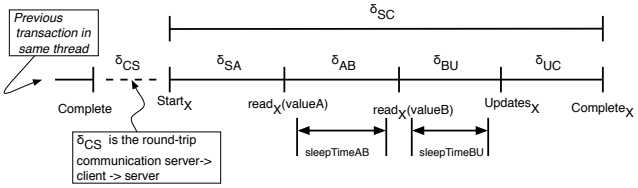

Figure 4: Diagram of the times between events of a transaction

(ii) The probability that X and Y collide (that is, that they act on the same row). This is approximately $(0.9)^2/500 = 0.00162$, since a collision occurs when both X and Y choose within the hotspot (each chooses the hotspot with probability 0.9), and they choose the same one out of 500 possible rows in the hotspot. We neglect here the much smaller chance of collision on a row outside the hotspot.

(iii) Now, when X and Y have a collision, we look at the possible outcomes for X, classified according to the transaction types for X and Y; each outcome may also depend on the timing of the completion of Y, compared against the events of X. The analysis is given in the Table of Figure 5. We refer to the condition during(Y,X) which means that commit(Y) occurs between start(X) and completion(X). Lines 1, 3, and 5-9 of the table in Figure 5 derive from the fact that when two concurrent transactions write to the same row, the later one aborts (the "First-Committer-Wins" rule of SI). However, when changeA and changeB are concurrent, and collision occurs, we have a non-serializable write-skew [3]; furthermore, in this case, each will see the same state of the rows concerned, and so each will choose the same delta. If before these two transactions, the sum of TableA.valueA and TableB.valueB for this row is between 0 and 49, each transaction will add 50 to its column, and thus the sum will increase by 100 (falling outside the permitted range); similarly if the sum each sees is between 50 and 99, the effect will be to decrease each column by 50, and the sum will fall by 100, so it becomes below 0. Thus when a collision occurs between concurrent changeA and changeB, the one to commit later will inevitably create a violation. This is illustrated in Figure 6. Thus we have all together a chance of 2/9 that X and Y will consist of one changeA and one changeB, and a chance of $(\delta_{SA} + \delta_{AB} + \delta_{BU} + \delta_{UC})/MI = \delta_{SC}/MI$ that Y will commit during the period when a violation will eventuate, if the appropriate transactions collide. Overall the probability of a violation by X, given a collision between X and Y, is $\frac{2\delta_{SC}}{9MI}$.

Combining the calculations above, we see that the probability of a violation caused by X at SI is $\frac{9*MI}{\delta_{CS}+\delta_{SC}} \frac{(0.9)^2}{500} \frac{2\delta_{SC}}{9MI}$. We define a new parameter $\alpha = \delta_{SC}/(\delta_{CS} + \delta_{SC})$ so that we

Type(X)	Type(Y)	Outcome for X in case of collision
changeA	changeA	abort if during(Y,X)
changeA	changeB	violation if during(Y,X)
changeA	changeAB	abort if during(Y,X)
changeB	changeA	violation if during(Y,X)
changeB	changeB	abort if during(Y,X)
changeB	changeAB	abort if during(Y,X)
changeAB	changeA	abort if during(Y,X)
changeAB	changeB	abort if during(Y,X)
changeAB	changeAB	abort if during(Y,X)

Figure 5: Outcomes for colliding X and Y at SI

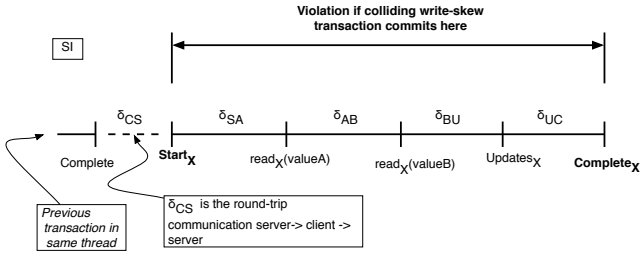

Figure 6: Diagram of the relationships in time for violation at SI

can express the probability of a given instance X producing a violation as $9 * \frac{(0.9)^2}{H} * \frac{2}{9}\alpha = 0.00324\alpha$.

In the same way we argue that the probability that a transaction instance aborts due to First Committer Wins rule is $9\frac{(0.9)^2}{500}\frac{7}{9}\alpha = 0.0113\alpha$. That is because in 7/9 of the cases where a collision occurs, the transactions write to the same item and so the later one will abort if they are concurrent. Thus the rate of violations per committed transaction is predicted as $0.00324\alpha/(1-0.0113\alpha)$. To make this useful, we need to estimate α. For the configuration we have, we know that δ_{SC} is greater than the sum of the sleep times, which are requested to be 300ms each on average ; as we noted above, the platform systematically sleeps less than requested, by on average about 8ms per sleep; thus we know that δ_{SC} will exceed (292+292)=584 ms. Measurements, taken with a single client thread and very fast transactions (sleeptimeAB and sleeptimeBU set to zero), show that the throughput available on our platform is over 1500 transactions per second, so the time needed for client-server round trip communication is well below 1ms, that is δ_{CS} is below 1ms. Thus in our configuration, α is indistinguishable from 1, and so we predict a violation rate of 0.00328.

Generalizing this calculation, we consider a general system configuration, with given MPL, hotspot of H rows accessed by fraction F of the transaction instances, and transaction mix with fraction f_A of changeA, f_B of changeB, and f_{AB} of changeAB. For this configuration, and isolation level of SI, we predict a violation rate which we call SI* and which is shown in Figure 7. Here (MPL-1) is the number of other threads to consider, (F^2)/H is the probability of two instances colliding on a row of the hotspot, $2f_Af_B$ is the probability of the instances forming a pair with one changeA and one changeB (so write skew can occur). In the denominator we correct for the number of aborts; here $[f_A^2+2f_Af_{AB}+f_B^2+2f_Bf_{AB}+f_{AB}^2]$ is the probability of the instances forming a pair which write to the same item (and so where the later one aborts), given that both a collision occurs, and the transactions are concurrent.

$$\frac{\frac{(\text{MPL}-1)*(F^2)}{\text{H}}[2f_Af_B]\alpha}{1-\frac{(\text{MPL}-1)*(F^2)}{\text{H}}[f_A^2+2f_Af_{AB}+f_B^2+2f_Bf_{AB}+f_{AB}^2]\alpha}$$

Figure 7: SI*: Predicted violation rate for SI

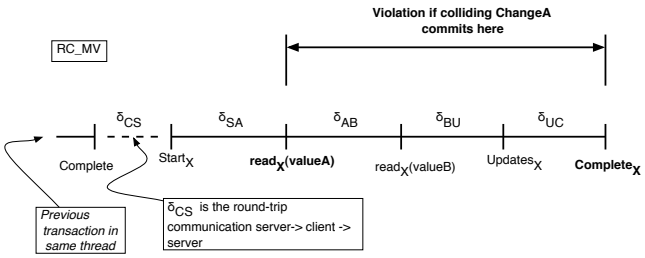

Figure 8: Diagram for violation at RCMV when Y is changeA

3.2 Isolation level = RC_MV

The calculation for multiversion Read Committed isolation is more complicated, because the behavior of a transaction instance X depends on how the commit of Y interleaves among the operations of X, and it also sometimes depends on the values in the database before X and Y begin. On the other hand, the formula is simpler in one respect, because there are no FCW aborts at Read Committed, and so we do not need anything corresponding to the denominator in Figure 7. As for SI above, the probability that X and Y collide on a row is F^2/H.

The table showing the outcome of X is in Figure 11, whose rows are derived from the illustrations in Figure 8, 9 and 10. Throughout this discussion we assume that X and Y collide on the same row. For example, Figure 8 shows that when Y is changeA, if the commit of Y occurs after X reads valueA, none of the operations in X will see the effect of Y, and so both transactions will act on the same information[9], namely whatever was in valueA and valueB before X and Y began). Thus, in this overlap, either both transactions will see a sum from 0 to 49 (and both will add 50 to the sum, causing a violation) or else both will see a sum between 50 and 99, both will subtract 50, and again a violation will occur. That is, a violation occurs when (there is a collision and) Y commits between X's read(valueA) and X's completion. However, if Y is changeA and it commits before X reads valueA, then X sees the impact of Y, and the violation is not produced (the interaction is serializable, as if Y ran entirely before X). Thus the probability of a violation being introduced by a given X because of a given Y that is changeA and that collides with X, is $(\delta_{AB}+\delta_{BU}+\delta_{UC})/MI$.

A similar argument applies when Y is changeB (and it collides with X). As shown in Figure 9, the probability of X introducing a violation because of Y is $(\delta_{BU}+\delta_{UC})/MI$ since when Y commits after X reads valueB but before X commits, the transactions sees the same sum, and so both add or both subtract from the sum, giving a violation.

[9]We neglect here the much smaller chance of a three-way collision.

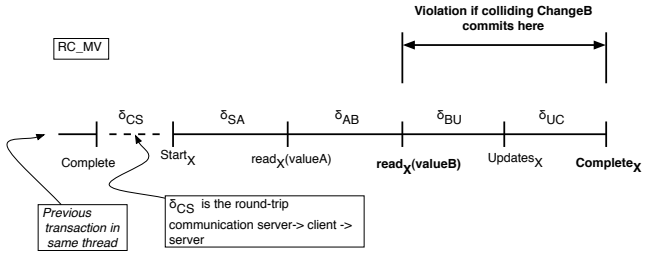

Figure 9: Diagram for violation at RCMV when Y is changeB

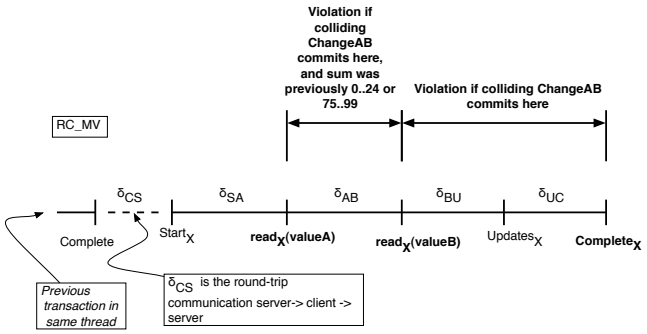

Figure 10: Diagram for violation at RCMV when Y is changeAB

The case when Y is changeAB (and it collides with X) is more complicated, and is illustrated in Figure 10. This gives rows 3, 6 and 9 of Figure 11. If Y commits after X reads valueB, but before X commits, then the transactions see the same sum and a violation occurs. If Y commits before X reads A, then X sees the sum as produced by Y, and the transactions behave like Y followed by X (so no violation). But when Y commits between the two reads by X, we have an inconsistent read in X. To deal with this case, we need to think about the different values the sum might have, before X and Y run. If the sum is form 0 to 24, then Y will add 25 to valueA (not seen by X) and it will add 25 to valueB (seen by X). Thus X will see a sum that is 25 higher than what Y saw; X will see a sum between 25 and 49, so X will choose delta of 50 too. The combined effect of X and Y will be that each added 50 to the sum, and so a violation is certain. However, if the sum before X and Y ran was between 25 and 49, then Y will choose delta of 50, X will see a sum that is increased by 25 (the part of Y's changes that it sees), and so X will calculate that the sum is between 50 and 74, and X will take delta of -50. The overall impact of X and Y will thus be to leave the sum as it was before, without a violation. A similar argument shows that when the sum is previously between 50 and 74, the combined effect is to leave the sum unchanged, while when the sum was previously above 74, both transactions subtract 50 from the sum and a violation is produced. Thus we can see that when Y is changeAB and commits between X's reads, a violation occurs in half the cases (when the sum is from 0 to 24 or from 75 to 99). That is, the probability that a particular X produces a violation because of this particular Y is $((\frac{1}{2}\delta_{AB}) + \delta_{BU} + \delta_{UC})/MI$.

Based on the argument above, we find the prediction in Figure 12. We have an overall prediction based on three

Type(X)	Type(Y)	Condition for X to violate integrity, supposing that a collision occurs
changeA	changeA	commit(Y) occurs between read$_X$(valueA) and commit(X)
changeA	changeB	commit(Y) occurs between read$_X$(valueB) and commit(X)
changeA	changeAB	commit(Y) occurs between read$_X$(valueB) and commit(X), or else both (the initial sum is not from 25 to 74) and (commit(Y) occurs between read$_X$(valueA) and read$_X$(valueB))
changeB	changeA	commit(Y) occurs between read$_X$(valueA) and commit(X)
changeB	changeB	commit(Y) occurs between read$_X$(valueB) and commit(X)
changeB	changeAB	commit(Y) occurs between read$_X$(valueB) and commit(X), or else both (the initial sum is not from 25 to 74) and (commit(Y) occurs between read$_X$(valueA) and read$_X$(valueB))
changeAB	changeA	commit(Y) occurs between read$_X$(valueA) and commit(X)
changeAB	changeB	commit(Y) occurs between read$_X$(valueB) and commit(X)
changeAB	changeAB	commit(Y) occurs between read$_X$(valueB) and commit(X), or else both (the initial sum is not from 25 to 74) and (commit(Y) occurs between read$_X$(valueA) and read$_X$(valueB))

Figure 11: Outcomes for colliding X and Y at RCMV

factors. One is the number of possible choices of Y (as for SI above this is $\frac{(\text{MPL}-1)*\text{MI}}{(\delta_{CS}+\delta_{SC})}$). Another factor is the chance of collision, which is $\frac{\text{F}^2}{H}$. Finally we have a weighted average of the probability of violation for particular X and Y, where the weights come from the frequency of transaction types. The quantity MI cancels from the formula, and we see that the impact of the different durations on the expression is all based on indirect parameters $\beta = \frac{\delta_{AB}+\delta_{BU}+\delta_{UC}}{\delta_{CS}+\delta SC}$ and $\gamma = \frac{\delta_{BU}+\delta_{UC}}{\delta_{CS}+\delta SC}$.

How can we estimate β and γ? For all the experiments we do in this paper, we have that δ_{CS}, δ_{SA} and δ_{UC} are each very short; their sum is less than the average spacing of transactions without sleeps, which we saw is below 1ms. On the other hand, δ_{AB} is at least the delay we explicitly sleep between the read(valueA) and the read(valueB); we request that this is on average the system parameter sleeptimeAB, and so (given the weirdness of the platform's quantum) it is estimated to be (sleeptimeAB - 8ms); similarly we estimate δ_{BU} to be sleeptimeBU - 8ms. Thus we estimate $\delta_{CS} + \delta_{SC}$ to be approximately (sleeptimeAB - 8)+(sleeptimeBU - 8), and so β is less than 1/(sleeptimeAB + sleeptimeBU - 16),

$$\frac{(\text{MPL}-1)*(F^2)}{\text{H}}\Psi$$

$$\begin{aligned}\text{where } \Psi &= [(1-\beta)f_A^2 + (2-\beta-\gamma)f_A f_B \\ &+(2-\frac{3\beta}{2}-\frac{\gamma}{2})f_A f_{AB} + (1-\gamma)f_B^2 \\ &+(2-\frac{\beta}{2}-\frac{3\gamma}{2})f_B f_{AB} + (1-\frac{\beta}{2}-\frac{\gamma}{2})f_{AB}^2]\end{aligned}$$

Figure 12: RCMV*: Predicted violation rate for multiversion RC

or essentially 0. Similarly, γ is very close to (sleeptimeBU - 8)/(sleeptimeAB + sleeptimeBU - 16).

For the example of the configuration with MPL=10 from Figure 1, we have F=0.9, H=500, $f_A = f_B = f_{AB} = 1/3$. SleeptimeAB = sleeptimeBU = 300ms, so β is indistinguishable from zero, while γ is almost 0.5. Thus we predict a violation rate of 0.0109.

4. VALIDATING THE PREDICTIONS

To validate the predictions, we have run measurements on a range of different configurations, exploring in turn the impact of changes to one of the parameters of the configuration. For these experiments we only consider the isolation levels of Snaphot Isolation and multiversion Read Committed, since those are the ones for which we have predictions. In each case we present the measured violation rates, and also predicted rates using the formulae of Figure 7 and Figure 12. We denote the measured rates by SI and RC_MV, and the predictions by SI* and RC_MV*. Note that measured rates are shown with error bars indicating the 95% confidence interval, but predictions have no error bars. All the graphs shown here include the particular configuration with MPL=10 from Figure 1. We did not repeat the experiment, but reused data that produced Figure 1. The other configurations presented were produced by varying one parameter systematically, and running new experiments with 250 runs, each measuring for 30s after 1s warmup. The runs were grouped into 5 sets of 50, with each set treated as a single super-run as described in Section 2.

Figure 13 shows experiments with varying MPL. There is a hotspot of 500 rows per table in a database whose tables have 5000 rows. The transactions are chosen equally from changeA, changeB and changeAB. The sleep time requested between read(valueA) and read(valueB) was chosen from a normal distribution Normal(300, 60), with mean 300ms and standard deviation 60ms (the distribution was truncated by removing values below 0 or above 600ms). The same distribution was used (independently) for each sleep between read(valueB) and the update in the transaction.

We note that the predictions show the correct trend, linear in MPL-1. They are also fairly accurate numerically, though there is a tendancy to predict above even the top of the confidence interval. The prediction is never more than 18% above the measured value (the worst accuracy happens for SI at MPL=10). This seems more extreme than just due to the under-reporting effect described in Section 2.

Next in Figure 14 we consider the impact of different amounts of contention as we change the size of the hotspot. All these experiments have MPL=10, equal amounts of the three transaction types, and sleep times each chosen from a

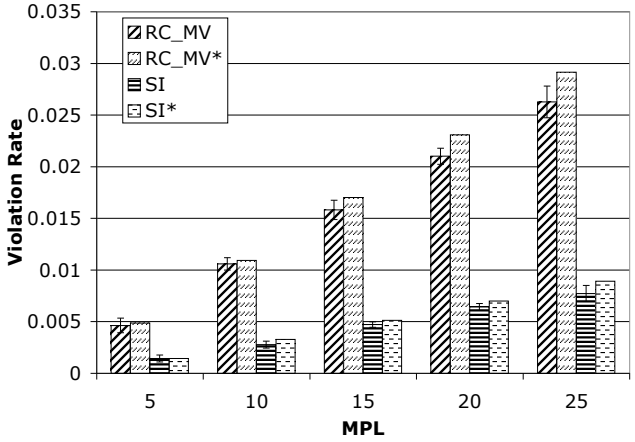

Figure 13: Measured and predicted Violation rate vs MPL.

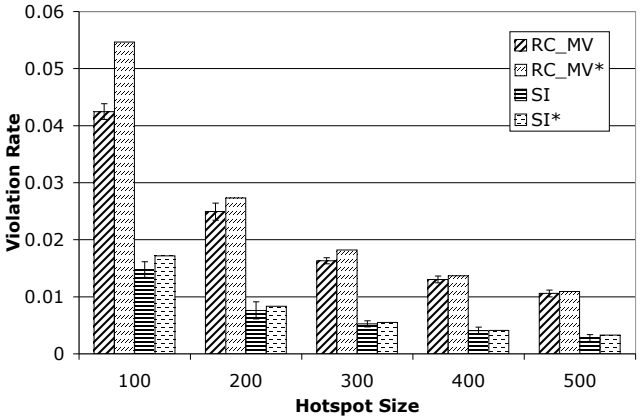

Figure 14: Measured and predicted Violation rate vs Size of Hotspot.

normal distribution of mean 300ms. We did 250 runs each measuring for 30s, and grouped them into 5 super-runs.

These predictions show the correct trend, decreasing as H^{-1}. They exceed the measurements by 28% in the case of RC_MV at Hotspot 100 (where we earlier showed we expect 13% under-reporting); in all the other cases the prediction is within 20% of the actual measurement. The worst accuracy (excepting the expected case) is for SI at H=500. This does not seem to be explained by under-reporting.

For Figure 15 we vary the transaction mix. In each configuration shown, changeAB occurs as 2/6 of the transactions; the frequency of changeA varies from 0 to 4/6, and the frequency of changeB goes correspondingly from 4/6 to 0. Zero violations are found with SI when no changeA is present. This is expected, since when only changeB and changeAB occur, any conflicts involve transactions that write a common item, and so SI's First Committer Wins rule prevents any non-serializable execution (a proof technique that implies this special case is given in [9]). The same is seen when all transactions are changeA or changeAB. Again the shape of the curve fits the prediction, and the values are close; except for the case of SI with 2:2:2, the discrepancy is always within 10% (though the confidence interval about the measurement does not always contain the prediction).

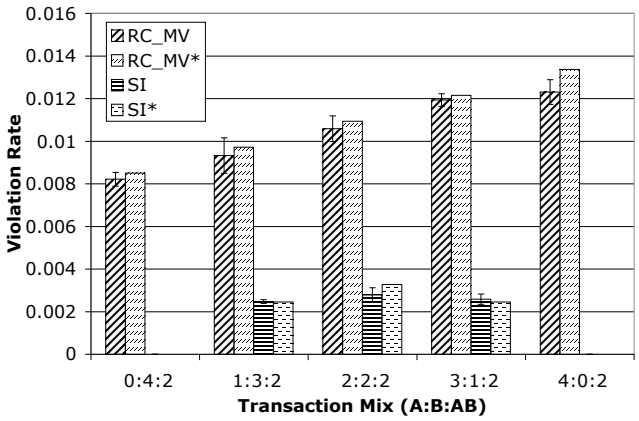

Figure 15: Measured and predicted Violation rate vs Transaction Mix.

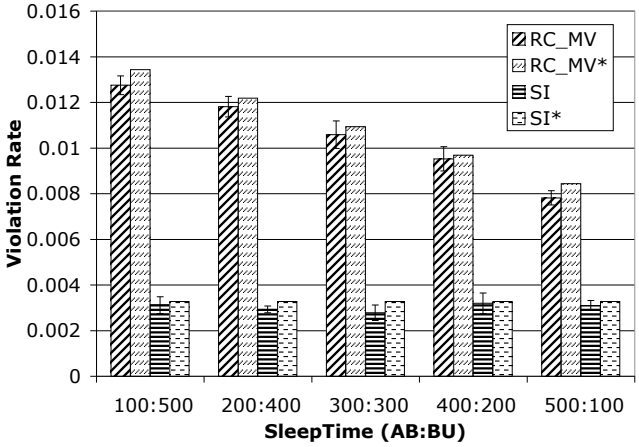

Figure 16: Measured and predicted Violation rate vs Sleep times.

Finally, we consider varying the balance between the sleep times. At the left of Figure 16 is the configuration where the sleep between read(valueA) and read(valueB) has an average of only 100ms (and standard deviation of 30ms), while the sleep from read(valueB) till the update has mean 500ms (and standard deviation 90s). At the righthand side we reverse the distributions used to choose the sleep times. Notice that as predicted, the relative placement of the read(valueB) within the transaction is not important for the behavior under SI; however at multiversion Read Committed, we see more violations when the gap between read(valueA) and read(valueB) is relatively more of the overall duration of the transaction. Aside from the case of SI at 300:300 (which comes from Figure 1 rather than being measured anew), the discrepancies are all below 12%.

5. AN INVERSION FOR VIOLATION RATE

The multiversion Read Committed concurrency control mechanism is not strictly more permissive than the Snapshot Isolation mechanism, even though the latter is usually seen as a stronger isolation level. That is, there are some executions allowed by the RC_MV algorithm that are not allowed by SI. This raises the question of whether any system

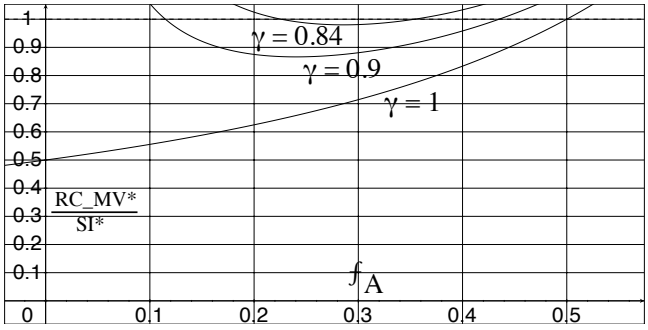

Figure 17: Predicting an Inversion.

configuration could be tuned so that an application would create more anomalies when run with the stronger SI algorithm than when run with weaker RC_MV concurrency control. Our predictive model guides us to find a part of the configuration space, where this "inversion" actually occurs.

To make it easier to search through the multidimensional space of configuration parameters, we consider each configuration parameter in turn. Consider the trends shown in the measurements we have reported, and also predicted by our model. Changes in MPL, F and H each impact of the violation rate of each isolation level in the same way: the violation rate increases linearly with MPL, quadratically with F, and inversely with H. Thus choice of MPL, F or H will not be crucial in finding an inversion. Changing the relative duration of sleepAB to sleepBU has no effect on the violations under SI, but there are more violations under RC_MV when sleepAB is large. That is, γ should be as close to 1 as possible, in order to see an inversion. The impact of transaction mix is more complicated. We note first that any pair of transactions, one of which is changeAB, never produces anomalies at SI, but may do so when running under RC_MV. Thus we have the best hope of seeing an inversion if we do not run any changeAB instances, that is, we will take $f_{AB} = 0$. Now the trend shows clearly that the rate of violations under SI is highest when changeA and changeB are evenly balanced; and the rate is lower when the rates f_A and f_B are far apart. Under RC_MV, the rate is higher as f_B increases and f_A decreases. To find an inversion we will need to carefully choose the balance between changeA and changeB; the rate f_A should be low but not too low.

We illustrate this is Figure 17. This shows on the y-axis the ratio of RC_MV* to SI* (to be precise, we use a simplified SI* which neglects the correction from transactions that abort, which is the denominator of the formula in Figure 7). The x-axis shows different values for the fraction f_A of changeA; throughout this graph we assume $f_{AB} = 0$, so $f_B = 1 - f_A$. We also assume $\alpha = 1$ and $\beta = 0$, which is very close to true for the systems we can experiment on. The remaining configuration that matters is γ. We have drawn the curves for γ being 0.84 (corresponding to sleep times of 500:100), for $\gamma = 0.9$, and for $\gamma = 1$ (the limiting case, where read(valueB) occurs right at the end of the transaction). When $\gamma < 0.83$ the curve lies entirely above the ratio of 1, and so inversion is not possible; even at $\gamma = 0.84$ the ratio is never far below 1; so detecting this situation reliably will be difficult. We measured this case, with sleeptimes 500:100 and $f_A = 0.25$ or 0.3; we found the violation rate for RC_MV below that for SI, but not significantly so. Thus

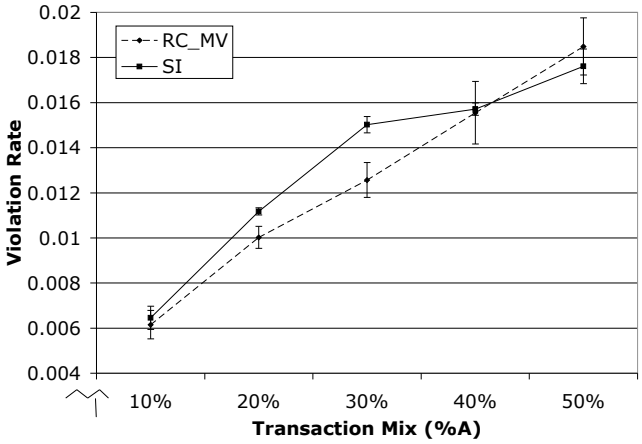

Figure 18: An Inversion.

we decided to look at cases with higher γ: we choose to measure when $\gamma = 0.9$, for which an inversion ought to be found for f_A values between 0.11 and 0.42, with the clearest difference occurring between 0.2 and 0.3.

In Figure 18 we show measurements we took at the configurations where this prediction suggests an inversion is most likely to be detectable. The points where f_A is 0.2 or 0.3 clearly demonstrate the existence of an inversion: the confidence interval around the violation rate for RC_MV is entirely below the confidence interval around the violation rate for SI, for these points. These quantities are all based on sleeptimeAB with mean 900 and standard deviation 120, and sleeptime BU with mean 100 and standard deviation 30. These sleeptimes mean that γ is close to 0.9, as we intended. Other configuration parameters in the system that we measured are MPL=10, H=500, warmup = 1s, MI=30s. In this graph we have $f_{AB} = 0$, and f_A varies as shown on the x-axis; of course f_B is $1 - f_A$. Note that the y-axis does not extend down to 0.

We are not claiming that inversion will occur at realistic configurations or for realistic application code; indeed most of the graphs in this paper show the expected situation with more violations at the weaker RC_MV level than at SI. It is surprising that any situation could be chosen to reliably have more violations at the stronger level. Finding this also demonstrates the usefulness of our predictive model.

6. RELATED WORK

The proposal to provide application developers with a choice of isolation levels comes from [14]. In this paper the proposal gives specific locking algorithms that provide each level; the levels are named "Degree 0", "Degree 1", etc. Degree 2 is what was later called Read Committed in the SQL standard. The phantom problem, and the distinction between Repeatable Read and Serializable is described in [8].

The state-of-practice concurrency control for DBMS engines is presented very clearly in [15]. For locking approaches, the widely-used techniques include an efficient lock manager [13], with phantoms prevented by some form of next-key or index lock [19, 20, 18].

The alternative multiversion approach to concurrency control usually provides Snapshot Isolation; indeed several products run with SI even if the application declares its isolation level to be Serializable[16]. The fact that SI does not ensure serializable execution was shown in [3]. A graph-based condition can determine whether particular application programs give rise to non-serializable execution under SI [9], and checks for this condition can be automated [17]. A variant of the SI algorithm provides true serializability [4]; however, this is not yet available in commercial platforms.

There are several different ways in which the database community has quantified the performance of different system designs (in particular, investigating the impact of different concurrency control algorithms). One approach has focused on measuring the performance under controlled conditions[10]. Jim Gray was responsible for much of [2] which introduced a small banking set of programs as a benchmark, to allow fair comparison of performance, especially peak sustainable throughput. A consortium (TPC) has standardized many benchmarks, fixing precisely what is to be measured, and under what conditions [23]. [21] reports measurements of both throughput, and the number of erroneous answers to queries, among transactions at different isolation levels.

Another way to investigate performance is by detailed simulations, where a computer program representation of a whole system is evolved through many steps. An early paper was [6]. [1] compares the throughput obtained at different MPL, under different assumptions about the level of disk and CPU resources. A similar simulation study has focused on multiversion concurrency control algorithms [5].

Instead of measuring performance in a real system or a computer model, another approach tries to find a formula which relates performance to configuration parameters. In the analytical style, the formula is based on queueing theory and the focus is on the impact of different distributions for the random choices, such as the variance in the work needed for different requests, or in the inter-arrival spacing [22]. A different style aims for much simpler formulas, usually with grossly simplified assumptions and estimates of the probability of various situations arising. The seminal example is the prediction of waiting and deadlock probabilities in [12]; later examples include [11, 7].

7. CONCLUSIONS AND FUTURE WORK

We have described a way to quantify the extent to which different, weak, isolation levels allow data corruption. Our microbenchmark allows us to explore carefully the impact of different system and application features on the rate of serialization errors for each isolation level. While the microbenchmark uses a few simple transaction types, the trends we observe should apply more widely. For example, the rate of violation introduction is proportional to MPL $-$ 1, and inversely proportional to the size of the hotspot.

We have provided a novel predictive model for the rate of violations introduced when our microbenchmark is run using the multiversion concurrency control algorithms. Our model gives appropriate predictions of the trends, and the actual values are also quite accurate, within 20% of the measurement in almost all cases, and frequently within 10%.

Using our predictive model, we found a carefully tuned configuration where an unexpected inversion occurs: more violations are produced with the fairly strong SI isolation level (which is even used for serializability by some platforms) than with the multiversion Read Committed mechanism which is considered a weaker level.

In future work, we hope to invent a microbenchmark that

includes some predicate-based reads, and insert and delete statements, so that we can measure the violations allowed by Repeatable Read. We would hope to examine some industry installations, and see how frequently isolation errors are in data used by realistic application code. Another direction is to extend our predictive model from the particular transactions in the microbenchmark, to deal with arbitrary mixes of application code, in particular so it can be applied to more realistic programs. This will be based on a static dependency graph [9] which shows which pairs of transactions deal with the same item, but it will also need careful analysis of which interleavings actually cause violation of integrity constraints.

8. ACKNOWLEDGMENTS

Research supported by Australian Research Council grant DP0987900. Early versions of the microbenchmark were coded with help from Mohammad Alomari and Michael Cahill. The University of Sydney Database Research Group provided feedback on the paper.

9. REFERENCES

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[2] Anon, et al. A measure of transaction processing power. *Datamation*, 1985.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, , and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 1–10. ACM Press, June 1995.

[4] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 729–738, New York, NY, USA, 2008. ACM.

[5] M. J. Carey and W. A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, 1986.

[6] M. J. Carey and M. Stonebraker. The performance of concurrency control algorithms for database management systems. In U. Dayal, G. Schlageter, and L. H. Seng, editors, *Tenth International Conference on Very Large Data Bases (VLDB'84)*, pages 107–118, 1984.

[7] S. Elnikety, S. Dropsho, E. Cecchet, and W. Zwaenepoel. Predicting replicated database performance from standalone database profiling. In *Proc EuroSys'09*, pages 303–316, 2009.

[8] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[9] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[10] J. Gray. *The Benchmark Handbook (2nd ed)*. Morgan Kaufmann, 1993.

[11] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 173–182. ACM Press, 1996.

[12] J. Gray, P. Homan, R. Obermarck, and H. Korth. A strawman analysis of the probability of waiting and deadlock in a database system. Technical Report RJ3066, IBM San Jose Research Laboratory, February 1981.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[14] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems*, pages 365–394. North Holland Publishing Company, 1976.

[15] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[16] K. Jacobs, R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, and B. Quigley. Concurrency control, transaction isolation and serializability in SQL92 and Oracle7. Oracle White Paper, Part No A33745, 1995.

[17] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB '07: Proceedings of the 33rd international conference on Very Large Data Bases*, pages 1263–1274. VLDB Endowment, 2007.

[18] D. B. Lomet. Key range locking strategies for improved concurrency. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 655–664, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[19] C. Mohan. ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *Proceedings of international conference on Very Large Databases (VLDB'90)*, pages 392–405, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[20] C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of ACM SIGMOD international conference on Management of Data (SIGMOD'92)*, pages 371–380, New York, NY, USA, 1992. ACM.

[21] D. Shasha and P. Bonnet. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[22] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, 1998.

[23] Transaction Processing Performance Council. TPC-E Benchmark Specification, 2007.

[24] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, San Francisco, California, 2002.