Using Transaction Semantics to Increase Performance*

Arthur J. Bernstein[†] David S. Gerstl Philip M. Lewis Shiyong Lu

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400 USA
{art, gerstl, pml, shiyong}@cs.sunysb.edu

1 Introduction

Serializability is the most stringent level of isolation used in transaction processing systems and has been widely accepted as the correctness criterion for concurrently executing transactions. It generally uses a strict two-phase locking protocol, in which locks are held until transactions commit. Unfortunately, this implies that locks might be held for long periods of time, causing performance to suffer, particularly in applications having long running transactions and/or data hotspots. In this paper, we describe an alternative definition of correctness, *semantic correctness*, which can be used to significantly improve the performance of transaction processing systems while maintaining correctness. We present our results in three different contexts.

1. Some authors have proposed decomposing programmed transactions into atomic and isolated pieces called *steps*. Steps release locks when they complete, and thus the steps of concurrently executing transactions can be interleaved. Since locks are held for shorter periods of time, performance can be improved. Furthermore, by choosing the step boundaries, the programmer can control the points at which locks are released. However, correctness is a problem since the interleaved schedules of steps might not be serializable and might not be correct. Semantic correctness can be used to provide a specification of correct step interleavings and as the basis of a concurrency control that will schedule the execution of steps based on that specification. We give experimental results showing the increased performance that can be provided with such a concurrency control.

2. One technique for improving the performance of an application is to use an isolation level that is less stringent than SERIALIZABLE. With READ COMMITTED, for example, performance gains are achieved by releasing read locks early. Unfortunately, only the SERIALIZABLE level guarantees correct execution for *all* applications. Lower isolation levels are generally chosen in an *ad hoc* fashion, with no guarantee that the application will run correctly.

*This paper is based upon work supported by NSF grant CCR-9402415. The authors would like to express their gratitude to Computer Associates International^{tm} for the donation of the copy of CA-Open Ingres^{tm} used in these experiments.

[†] Contact Author:(516)632-8457/8334 (fax)

Semantic correctness can be used to provide such a guarantee — thus achieving the higher performance allowed when lower isolation levels are used, without sacrificing correctness.

3. A multidatabase system is a collection of logically interrelated databases distributed over a network. A global transaction over a multidatabase is a transaction that invokes the subtransactions exported by the individual databases. If all sites in a multidatabase system use strict two-phase locking and participate in a two-phase commit protocol, the transactions are globally serializable and hence correct. However, in many applications some of the individual sites are unwilling (perhaps because of performance considerations) or unable (perhaps because they are legacy systems) to participate in a two-phase commit protocol and hence the transactions might not be globally serializable and might not be correct. Since the subtransactions in a multidatabase system are similar in many ways to the steps in a step-decomposed transactions, semantic correctness can be used to determine correct executions and to build a global concurrency control that will schedule the subtransactions to obtain correct executions — thus achieving the higher performance allowed when two-phase commit is not used, without sacrificing correctness.

2 Semantic Correctness

We assume that the desired effect of each transaction is described by a postcondition derived from its specification, and that the allowable states of the database are described by a consistency constraint. We have developed a formal definition of semantic correctness based on these postconditions. Informally, a schedule of transactions is *semantically correct* if the initial and final states of the database are consistent and the final state reflects the combined results of all committed transactions as specified in their postconditions. For example, the combined results of a schedule of bank deposit and withdrawal transactions is that the final value of the account-balance is the initial value plus the sum of the deposits minus the sum of the withdrawals. We believe this is the weakest sufficient condition for the correct execution of an arbitrary application. It is weaker than serializability since any schedule that is serializable is semantically correct, but it allows schedules that produce states that cannot be reached in any serial schedule. Other published "correctness criteria," such as strong correctness, two-level serializability, and the lower isolation levels, do not guarantee all parts of our definition for all applications.

3 A Concurrency Control for a Step-Decomposed Transactions

When transactions are decomposed into steps, the programmer must specify how the steps of transactions can be interleaved. This specification can be done either in an *ad hoc* fashion (analogous to the choice of a lower isolation level), or by using formal methods. In either case, allowable interleavings can be described by a table which indicates whether a step, S, of transaction T_1 can be allowed to execute between two steps, S_i and S_{i+1}, of transaction T_2. We have built a concurrency control, called an *assertional concurrency control* (ACC), within the CA-Open Ingres^{tm} database management system that uses such a table to permit only allowable interleavings.

An *ad hoc* specification might be appropriate when performance is degraded by a few serious points of lock contention. Performance might then be significantly improved by decomposing a few transactions into a small number of steps and determining allowable interleavings using informal reasoning. In a formal approach, a transaction's postcondition is used to determine the precondition of each step. We identify (at design time) the steps that can potentially invalidate each such assertion. The ACC produces semantically correct schedules by ensuring that a step's precondition is true when it is executed and a transaction's postcondition is not invalidated by transactions that have been interleaved with it.

Step interleaving is controlled using a new lock mode, called *assertional lock mode*, incorporated into a conventional locking system. Assertional locks are weaker than conventional read/write locks. Conflicts between assertional locks are detected at run time by a simple table look-up. While a step releases all conventional locks when it completes, assertional locks are held between steps to control interleaving. Viewed formally, assertional locks are used to lock assertions. An assertion is locked when each item referenced by the assertion has been assertionally locked.

The results produced by a step become visible to concurrent transactions when the step completes. Hence, the only way that the effects of a transaction can be reversed is through compensation. Compensating steps must be provided by the application programmer and are maintained by the database server as stored procedures. When a step completes, it writes a step-commit log record that includes a reference to the corresponding compensating step. These records need not be forced until the transaction as a whole commits. Thus the server can autonomously initiate compensation by reading the log record and invoking the appropriate stored procedure. Furthermore the transaction saves some of its work area in a database table so that compensation can be initiated if the transaction is rolled back due to a crash.

The ACC was tested using a load based on TPC-C^{tm} Benchmark Transactions. The major departures from benchmark code was that in some of the experiments we substituted a read/write *restock* transaction for the read/only *stock_level* transaction and in some experiments we introduced 3 seconds of simulated compute time into these transactions. The purpose of these modifications was to measure the impact of long-running transactions. Results are shown in Figure 1. Each curve plots the ratio of the average response time using strict two-phase locking to the average response time using the ACC as a function of the level of concurrency (measured by the number of active terminals). The figure demonstrates an improvement of up to 80% when lock contention is high, when long running transactions are a part of the mix, and/or when sufficient system resources are present to support the additional concurrency that the new control makes possible.

4 Correctness of Non-Serializable Isolation Levels

The ANSI/ISO standard defines three isolation levels lower than SERIALIZABLE: READ UNCOMMITTED, READ COMMITTED, and REPEATABLE READ. At least one major database vendor also provides SNAPSHOT isolation, implemented through a combination of locking and multiversion techniques. In addition some vendors implement a variant of READ COMMITTED, called optimistic READ COMMITTED or READ COMMITTED with first-committer wins.

Databases frequently use locking protocols to implement these levels. These protocols can be interpreted as decomposing transactions into steps. Thus semantic correctness can be used to determine the correctness of the resulting schedules. For each isolation level we prove a condition having the property that, if each transaction satisfies the condition corresponding to its level, all schedules will be semantically correct. We assume that different transactions can be executing at different levels, but that each transaction is executing at least at READ UNCOMMITTED. Thus the performance benefit resulting from the use of lower isolation levels does not come at the expense

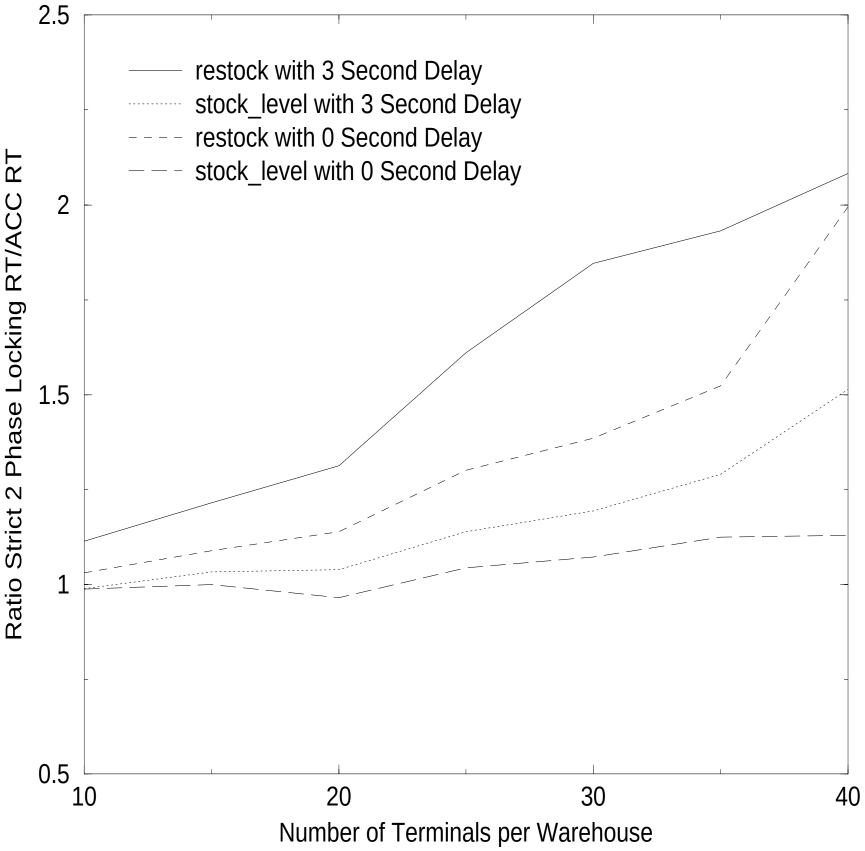

Figure 1: Ratio NACC/ACC Response Time in Four Configurations

of correctness. Examples of these results are:

> A transaction, T_i, executed at the READ COMMITTED level, executes semantically correctly if all the writes of each other transaction, considered as an atomic unit, do not interfere with the postcondition of every READ statement in T_i and with the postcondition of T_i.
>
> A transaction, T_i, executed at the READ COMMITTED with first-committer-wins level executes semantically correctly if all the writes of each other transaction, considered as an atomic unit, do not interfere with the postconditions of those READ statements in T_i that do not have a following WRITE statement (somewhere in the transaction) on the same item, and do not interfere with the postcondition of T_i.
>
> Let T_i be a transaction executed at the REPEATABLE READ level, and $S_{i,j}$ be an arbitrary SELECT in T_i. T_i executes semantically correctly if all the write statements of each other transaction, considered as an atomic unit, do not interfere with the postcondition of T_i and either

(1) do not interfere with the postcondition of $S_{i,j}$, or
(2) include DELETE or UPDATE statements that attempt to delete or update some subset of the tuples returned by $S_{i,j}$.

A set of transactions executes semantically correctly under SNAPSHOT isolation if, given any two transactions T_i and T_j from the set, either:
(1) T_i or T_j is a read-only transaction or
(2) T_i's write set intersects T_j's write set or
(3) all the write statements of T_j considered as an atomic unit do not interfere with the precondition of the first WRITE statement in T_i and with the postcondition of T_i.

5 A Global Concurrency Control for Multidatabase Systems

In a common model of a multidatabase systems, a global transaction invokes subtransactions exported by local databases. When each subtransaction completes, it immediately commits. Thus two-phase commit is not used to support global atomicity. We assume that each site uses two-phase locking, so subtransactions at a site are serialized, but since two-phase commit is not used, the serialization order at different sites might be different. Hence there might be no global serialization order and execution might not be correct. As an added complication, local transactions might be initiated at a site and might affect the serialization order at that site.

In the case when there are no local transactions, the subtransactions of a particular global transaction can be viewed as the steps of a step-decomposed version of that transaction. Thus the notion of semantic correctness can be used to specify the allowable interleavings of these subtransactions to achieve semantic correctness. The concepts underlying the ACC can then be used to build a Global Concurrency Control to schedule subtransactions to achieve semantic correctness.

We are currently building such a system, called JAMB, (Java Accessing MultidaBases). JAMB is middleware that provides global transaction support in the context of Java. It ensures that concurrent schedules of distributed transactions are semantically correct. JAMB occupies the middle tier of a 3-tier architecture. Clients talk to JAMB using RMI (Remote Method Invocation), and JAMB utilizes JDBC to talk to the component databases.

An initial prototype of JAMB has been implemented.[1] We hope to expand it to deal with the situation in which a site can contain local transactions that do not talk to JAMB. We are investigating the relationship between semantic correctness and the concepts of two-level serializability and strong correctness, which have been applied to this situation.

6 Conclusion

The use of serializability as a correctness condition has severe implications for reducing the performance of a transaction processing system. Significant performance improvements can be obtained using a weaker correctness condition *semantic correctness*, which specifies only that the combined execution of all transactions reflects the desired result of each transaction. Semantic correctness is arguably the weakest correctness condition. We have investigated the use of semantic correctness in the contexts of step-decomposed transactions, execution at lower isolation levels, and multidatabase systems — in each case providing improved performance without sacrificing correctness.

[1] http://www.cs.sunysb.edu/~shiyong/projects/jamb/jamb.html