Transaction Chopping: Algorithms and Performance Studies

DENNIS SHASHA
Courant Institute, New York University

FRANCOIS LLIRBAT, ERIC SIMON, and PATRICK VALDURIEZ
Projet Rodin, INRIA Rocquencourt

Chopping transactions into pieces is good for performance but may lead to nonserializable executions. Many researchers have reacted to this fact by either inventing new concurrency-control mechanisms, weakening serializability, or both. We adopt a different approach We assume a user who

—has access only to user-level tools such as (1) choosing among isolation degrees 1–4, (2) the ability to execute a portion of a transaction using multiversion read consistency, and (3) the ability to reorder the instructions in transaction programs; and

—knows the set of transactions that may run during a certain interval (users are likely to have such knowledge for on-line or real-time transactional applications)

Given this information, our algorithm finds the finest chopping of a set of transactions TranSet with the following property: *If the pieces of the chopping execute serializably, then TranSet executes serializably.* This permits users to obtain more concurrency while preserving correctness. Besides obtaining more intertransaction concurrency, chopping transactions in this way can enhance intratransaction parallelism.

The algorithm is inexpensive, running in $O(n \times (e + m))$ time, once conflicts are identified, using a naive implementation where n is the number of concurrent transactions in the interval, e is the number of edges in the conflict graph among the transactions, and m is the maximum number of accesses of any transaction. This makes it feasible to add as a tuning knob to real systems.

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*simulation*; H 2 4 [**Database Management**]: Information Systems—*concurrency*; *transaction processing*; I.6.8 [**Simulation and Modeling**]: Computing Methodologies—*discrete event*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Locking, multidatabase, serializability, tuning

A preliminary version of this paper appeared in the *Proceedings of the ACM SIGMOD International Conference*, held in San Diego, California, May 1992, under the title "Simple Rational Guidance for Chopping Up Transactions."
This work was supported by U.S. Office of Naval Research N00014-91-J-1472 and N00014-92-J-1719, and U.S. National Science Foundation grants IRI-89-01699 and CCR-9103953. Much of this work was done while D. Shasha visited INRIA during the academic year 1991–1992.

Authors' addresses: D. Shasha, Courant Institute, New York University, New York, NY 10012; F Llirbat, E. Simon, and P. Valduriez, Projet Rodin, INRIA Rocquencourt.

1. MOTIVATION

The database research literature has many excellent proposals describing new concurrency-control methods. The proposals (some of which are described in Section 8 and others that are cited in Krithi 1995) aim to help database-management system (DBMS) designers to build better concurrency-control methods into their systems. However, the fact remains that the vast majority of commercial database systems use two-phase locking to enforce serializability and less restrictive locking methods to enforce degree 2 isolation (i.e., write locks obey two-phase locking; read locks may be released immediately after the read of the locked resource completes—this is the normal implementation of degree 2 isolation). Some other systems offer multiversion read consistency. So, it is of significant practical interest to find ways to reduce concurrent contention given just those mechanisms.

Performance consultants and tuning guides suggest a simple way: "Shorten transactions or use less restrictive locking methods whenever you can. Serializability is an overly conservative constraint in any case." Database administrators and users follow this advice. The trouble is that problems can then crop up mysteriously into applications previously thought to be correct.

Example 1. *Inventory and Cash.* Suppose that an application program, named *purchase*, processes a purchase by adding the value of the item to inventory and subtracting the money paid from cash. The application specification requires that cash never be made negative, so the transaction will roll back (i.e., undo its effects) if subtracting the money from cash will cause the cash balance to become negative.

To improve performance, the application designers divide the two steps of *purchase*(i, p), that is, purchase item i for price p, into two transactions:

T1: if ($p >$ cash) then rollback else inventory[i] := inventory[i] + p;
T2: cash := cash − p;

They find that the cash field occasionally becomes negative. Consider the following scenario: There is \$100 in cash available when a first application program begins to execute. The item to be purchased costs \$75. So, the first transaction commits. Then some other execution of this application program causes \$50 to be removed from cash. When the first execution of the program commits its second transaction, cash will be in deficit by \$25.

So dividing the application into two transactions can result in an inconsistent database state. Once seen, the problem is obvious, though no amount of sequential testing would have revealed it. Most concurrent testing would not have revealed it either, since the problem occurs rarely. Still, this comes as no surprise to concurrency-control afficionados. A little surprising is how slightly the example must be changed to make everything work well.

Example 2. *Variant on Inventory and Cash.* Suppose that we rearrange the purchase application to check the cash level and decrement it in the first step if the decrement will not make it go negative. In the second step, we add

the value of the item to inventory. We make each step a transaction:

T1: if (p > cash) then rollback else cash := cash − p;
T2: inventory[i] := inventory[i] + p;

Using this scheme, cash will never become negative, and any execution of purchase applications will appear to execute as if each purchase transaction executed serially.

Remark Concerning System Failures and Minibatching. Suppose a system failure occurs after the first transaction but before the second transaction completes. The recovery subsystem will have no way to know that it must execute the second transaction. To solve this problem, the application must record its progress in each inventory–cash transaction and then complete the transaction from its recorded point upon recovery. This idea is reminiscent of the *minibatching* technique in industrial transaction processing [Gray and Reuter 1992]. For example, suppose that a batch transaction sequentially scans and updates 100 million records and executes in isolation. For performance and recoverability reasons, it may be a good idea to decompose that batch transaction into sequential "minibatch" transactions, each of which updates 1 million records and then writes the number of millions updated. On recovery, the application reads the last committed value of the number of millions updated and continues from there.

In the inventory–cash scenario, we do something analogous: Each transaction piece writes into a special table the item i, price p, and piece number. Recovery requires reexecuting the second pieces of inventory transactions whose first pieces have finished. In general, one must write enough information to complete the transaction in the case of failure. Minibatching will be required anytime a transaction chopped into k pieces has writes in one of the first $k - 1$ pieces. The alert reader may now wonder whether this special table will be a concurrency bottleneck. To avoid that problem, we can create several special tables, possibly one per user.[1]

Our goal is to help practitioners shorten lock times by chopping transactions into smaller pieces, all without sacrificing serializability. For the purposes of this paper, we do not propose a new concurrency-control algorithm, because we assume the user is tuning on top of a purchased DBMS (this assumption may be relaxed in future work). We do, however, assume that the user knows the set of transactions that may run during a certain interval of time. The user may also choose to release read locks early (e.g., the default in Sybase) or to use multiversion read consistency (e.g., the default in ORACLE). We consider this to be an important point about tuning research, as contrasted with classical research in database internals: "The tuner cannot

[1]Current research on this subject has identified efficient algorithms that use the log to save the context of applications instead of using separate tables [Salzberg and Tombroff 1994] This research entails changing the internal algorithms of the DBMS, so we consider it out of the purview of this paper.

change the system. But can use his or her knowledge to change the way an application runs on the system by rewriting it and by adjusting a few critical tuning knobs."

Surprisingly, the results are quite strong and compare favorably with some of the semantic concurrency-control methods proposed elsewhere. The algorithm is efficient. Given conflict information, the algorithm runs in $O(n \times (e + m))$ time using a naive implementation, where n is the number of concurrent transactions in the interval, e is the number of edges in the conflict graph among the transactions, and m is the maximum number of accesses of any transaction.

2. ASSUMPTIONS

To use this technique, the database user must have certain knowledge:

—*The database system user (here, that means an administrator or a sophisticated application developer) can characterize all of the transaction programs*[2] *that may run in some time interval.* The characterization may be parameterized. For example, the user may know that some transaction programs update account balances and branch balances, whereas others check account balances. However, the user need not know exactly which accounts or branches will be updated. The characterization may also include information such as that certain transaction programs always access a single record or that no two instances of some transaction program will ever execute concurrently. The more such information, the merrier.

—*The goal is to achieve the* guarantees *of serializability—without paying for it.* That is, the user would like either to use degree 2 isolation or to use multiversion read consistency even though the transaction has modification statements, or to chop transaction programs into smaller pieces. The guarantee should be that the resulting execution be equivalent to one in which each original transaction instance executes alone (i.e., serializably).

—*The user knows where rollback statements occur.* Suppose that the user chops up the code for a transaction program T into two pieces T_1 and T_2 where the T_1 part executes first. If the T_2 part executes a rollback statement after T_1 commits, then the modifications done by T_1 will still be reflected in the database. This is not equivalent to an execution in which T executes a rollback statement and undoes all of its modifications. Thus, the user should rearrange the code so rollbacks occur early. We will formalize this intuition later with the notion of rollback-safety.

—*If a failure occurs, it is possible to determine which transaction instances completed before the failure and which ones did not (by using some variant of the minibatching technique illustrated in the Remark of Example* 2).

[2]The word *transaction* can mean two things in the literature: a program text that states when transactions begin and end, and a running instance of that program text. We make the distinction between these two notions (calling them *transaction program* and *transaction instance*, resp.) where it is not clear from context.

Suppose there are n transaction instances $T_1, T_2, \ldots, T_n$ that can execute within some interval. Assume, for now, that each such transaction instance results from a distinct program. Chopping a transaction instance can then be done by modifying the unique program that only this transaction instance executes. We reexamine this assumption in Section 4.

Throughout this paper we consider transaction programs with simple control structures: sequences, loops, and **if-then-else** statements. As defined by Aho et al. [1986], a reducible flow graph can be associated with a transaction program. We shall say that a database access s "precedes" a database access s in a transaction instance if there is an edge from s to s' in the reducible flow graph of the transaction program. Intuitively, this means that if database accesses s and s' both execute, then s precedes s'. Database accesses s and s' may derive from the same instructions in the program text, but s' may be associated with a later iteration of a looping construct (e.g., a **while** or **for** loop).

A *chopping* partitions each T_i into *pieces* $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$. That is, every database access performed by T_i is in exactly one piece.

A chopping of a transaction program T is said to be *rollback-safe* if either T has no rollback statements or all of the rollback statements of T are in its first piece. Furthermore, all of the statements in the first piece must execute before any other statement of T. This prevents a chopped transaction instance from committing some of its modifications and then rolling back.[3] A chopping is said to be *rollback-safe* if the chopping of each of its transaction programs is rollback-safe.

Execution Rules (for the pieces of a chopping)

(1) When pieces execute, they obey the precedence relationship defined by the transaction program.[4]
(2) Each piece will execute according to some concurrency-control algorithm that ensures serializability and will commit its changes when it ends.[5]
(3) If a piece is aborted due to a lock conflict, then it will be resubmitted repeatedly until it commits.
(4) If a piece is aborted due to a system failure, it will be restarted.
(5) If a piece is aborted due to a rollback statement, then pieces for that transaction instance that have not begun will not execute.

[3]This definition of rollback-safety is slightly overly restrictive for the sake of presentation. It would be sufficient for all rollback statements to be in the first piece that has modification statements, as opposed to necessarily the first piece that has any database access. Such a definition would complicate the presentation, however.
[4]If piece s should precede piece s', then s should **complete** before s' **begins**.
[5]Extensions of this work to nonserializable environments or to multidatabase settings will find it necessary to relax this assumption. Normally, the result of the change will be to say that each piece should execute according to some correctness criterion of interest. e.g., epsilon serializability or multidatabase serializability. The challenge will be to show that each transaction obeys the correctness criterion.

3. WHEN IS A CHOPPING CORRECT?

We will characterize the correctness of a chopping with the aid of an undirected graph whose vertices are pieces and whose edges consist of the following two disjoint sets:

(1) *C-edges*—C stands for *conflict*. Two pieces p and p' from different original transaction instances conflict if reversing their order of execution will change either the resulting state or the return values. Formally, p and p' conflict if there is some state s such that executing pp' on s yields either a different resulting state or different return values when compared with executing $p'p$ on s. Thus, conflict is the same as *noncommutativity*. If we do not know the semantics of p and p', we will say that they conflict if there is some data item x that both access and at least one modifies. This is called a *syntactic* conflict. Knowing the semantics can, however, be helpful. For example, additions to inventory are commutative (i.e., do not conflict) with other additions to inventory, even though two such additions are in syntactic conflict. The more the user knows about his or her application, the fewer conflicts he or she will need to identify. If there is a conflict, draw an edge between p and p', and label the edge C.

(2) *S-edges*—S stands for *sibling*. Two pieces p and p' are siblings if they come from the same original transaction T. In this case, draw an edge between p and p', and label the edge S.

We call the resulting graph the *chopping graph*. (Note that no edge can have both an S and a C label.)

We say that a chopping graph has an *SC-cycle* if it contains a simple cycle that includes at least one S-edge and at least one C-edge.[6] We say that a chopping of $T_1, T_2, \ldots, T_n$ is *correct* if any execution of the chopping that obeys the execution rules is equivalent to some serial execution of the original transaction instances.

Equivalence is defined using the serialization graph formalism of Bernstein et al. [1991] as applied to pieces. Formally, a serialization graph is a directed graph whose nodes are transaction instances and whose directed edges represent ordered conflicts. That is, $T \rightarrow T'$ if some piece of T precedes and conflicts with some piece of T'. Following Bernstein et al. [1991], if the serialization graph resulting from an execution is acyclic, then the execution is equivalent to a serial one. Furthermore, Bernstein et al. proved the following fact:

Fact. (†) If all transaction instances use two-phase locking, then all those that commit produce an acyclic serialization graph.

THEOREM 1. *A chopping is correct if it is rollback-safe and its chopping graph contains no SC-cycle.*

---

[6] Recall that a "simple cycle" consists of (1) a sequence of nodes $n_1, n_2, \ldots, n_k$ such that no node is repeated; and (2) a collection of associated edges: There is an edge between n_i and n_{i+1} for $1 \leq i < k$ and an edge between n_k and n_1; no edge is included twice

PROOF. Call any execution of a chopping for which the chopping graph contains no SC-cycles an *SC-acyclic execution* of a chopping. We must show that

(1) any SC-acyclic execution yields an acyclic serialization graph on the given transaction instances $T_1, T_2, \ldots, T_n$ and, hence, is equivalent to a serial execution of committed transaction instances; and
(2) the transaction instances that roll back in the SC-acyclic execution would also roll back if properly placed in the equivalent serial execution.

For (1) we proceed by contradiction. Consider an SC-acyclic execution of a chopping of $T_1, T_2, \ldots, T_n$. Suppose there were a cycle in the serialization graph of $T_1, T_2, \ldots, T_n$ resulting from this execution. That is, $T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_i$. Identify the pieces of the chopping associated with each transaction instance that are involved in this cycle: $p \rightarrow p' \rightarrow \cdots \rightarrow p''$. Both p and p'' belong to transaction instance T_i. Pieces p and p'' cannot be the same, since each piece uses two-phase locking by the execution rules, and the serialization graph of a set of committed two-phase locked transaction instances is acyclic by fact (†). Since p and p'' are different pieces in the same transaction instance T_i, there is an S-edge between them in the chopping graph. Every directed edge in the serialization graph cycle corresponds to a C-edge in the chopping graph since it reflects a conflict. So, the cycle in the serialization graph implies the existence of an SC-cycle in the chopping graph, a contradiction.

For (2) notice that any transaction instance T whose first piece p rolls back in the SC-acyclic execution will have no effect on the database, since the chopping is rollback-safe. We want to show that T would also roll back if properly placed in the equivalent serial execution. Suppose that p conflicts with and follows pieces from the set of transaction instances $W_1, \ldots, W_k$. Then place T immediately after the last of those transaction instances in the equivalent serial execution. In that case, the first reads of T will be exactly those of the first reads of p. Since p rolls back, so will T. □

Theorem 1 shows that the goal of any chopping of a set of transactions should be to obtain a rollback-safe chopping without an SC-cycle. We now present a few examples of chopping to train the reader's intuition. The x's, y's, and z's here are specific and distinct data instances. Later we discuss the common case of programs parameterized by bind variables. For the purposes of these examples, we assume nothing about the application except that **R** stands for read and **W** for write. So our notion of conflict is entirely syntactic.

Chopping Graph, Example 1. Suppose there are three transaction instances that can abstractly be characterized as follows:

T1: **R(x) W(x) R(y) W(y)**
T2: **R(x) W(x)**
T3: **R(y) W(y)**

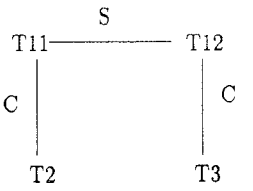

Fig 1. Graph without an SC-cycle for chopping example 1

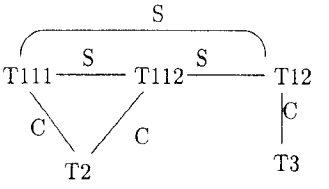

Fig 2. SC-cycle in chopping example 2

Breaking up **T1** into

T11: R(x) W(x)
T12:R(y) W(y)

will result in a graph without an SC-cycle (see Figure 1).

Chopping Graph, Example 2. With the same **T2** and **T3** as above, breaking up **T11** further into

T111: R(x)
T112: W(x)

will result in an SC-cycle (see Figure 2).

Chopping Graph, Example 3. Now, consider an example in which there are three types of transaction programs:

(1) a transaction program that updates a single depositor's account and the depositor's corresponding branch balance,
(2) a transaction program that reads a depositor's account balance, and
(3) a transaction program that compares the sum of the depositors' account balances with the sum of the branch balances.

For purposes of concreteness, consider the following transaction programs. Suppose that depositor accounts **D11**, **D12**, and **D13** all belong to branch **B1**; depositor accounts **D21** and **D22** both belong to **B2**. Here are the transaction texts (**RW** means a read followed by a write):

T1 (update account): RW(D11) RW(B1)
T2 (update account): RW(D13) RW(B1)
T3 (update account): RW(D21) RW(B2)
T4 (balance): R(D12)
T5 (balance): R(D21)
T6 (comparison): R(D11) R(D12) R(D13) R(B1) R(D21) R(D22) R(B2)

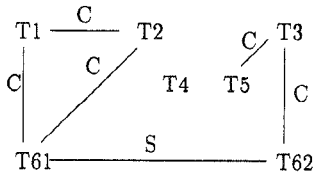

Fig. 3. Absence of an SC-cycle after chopping balance transaction.

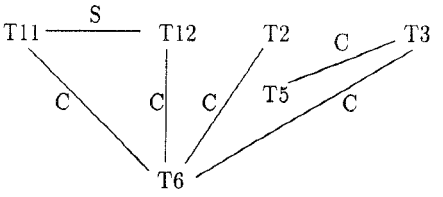

Fig. 4. Dividing **T1** into two transactions.

Let us see first whether the balance comparison transaction **T6** can be broken up into two transactions:

T61: **R(D11) R(D12) R(D13) R(B1)**
T62: **R(D21) R(D22) R(B2)**

The absence of an SC-cycle shows that this is possible (see Figure 3). Note that this could be generalized to u updates, b balance transactions, and 1 comparison. Each balance transaction would conflict with some update transaction. Each update transaction would conflict with exactly one piece of the branch-by-branch chopping of the comparison transaction. So, there would be no cycles.

Chopping Graph, Example 4. Taking the transaction population from the previous example, now consider dividing **T1** into two transactions giving the following transaction population (see Figure 4):

T11: **RW(D11)**
T12: **RW(B1)**
T2: **RW(D13) RW(B1)**
T3: **RW(D21) RW(B2)**
T4: **R(D12)**
T5: **R(D21)**
T6: **R(D11) R(D12) R(D13) R(B1) R(D21) R(D22) R(B2)**

This results in an SC-cycle.

Remark about Order-Preservation. The choppings we offer are serializable, but not necessarily order-preserving serializable. Consider the following example:

T1: **R(A) R(B)**
T2: **RW(A)**
T3: **RW(B)**

The chopping graph remains acyclic if we chop up **T1** into the transaction **R(A)** and the transaction **R(B)**. This would allow the following execution:

R(A) RW(A) RW(B) R(B)

This is equivalent to **T3, T1, T2** and so is serializable. It is not, however, order-preserving serializable, because **T2** completed before **T3** began yet appears to execute after **T3** in the only equivalent serial schedule.

4. FINDING THE FINEST CHOPPING

On the way to discovering an algorithm for finding the best possible chopping, we must answer two basic questions:

(1) Can chopping a piece into smaller pieces break an SC-cycle?
(2) If a chopping of transaction T alone does not create an SC-cycle and a chopping of transaction T' alone does not create an SC-cycle, can chopping both create one?

As it happens, the answer to both questions is negative under a natural notion of conflict. This will allow us to find an efficient optimization procedure. In the first subsections of this section, we assume that the execution of a chopped transaction follows the initial ordering of operations defined in the corresponding transaction program. In the last subsection, we explain how to accomplish this. This will lead us to a discussion of the problem of reorganizing transaction programs to make chopping more effective.

4.1 Separability Results about Choppings

For the purposes of the following two lemmas, please remember that we assume that there is a one-to-one correspondence between transaction instances and the transaction program. The results will allow us to relax this overly restrictive assumption later:

LEMMA 2. *If a set of chopped transaction instances contains an SC-cycle, then any further chopping of any of the transaction instances will not render it acyclic.*

PROOF. Let p be a piece of a transaction instance T to be further chopped, and let the result of the chopping be called pieces(p). If p is not in an SC-cycle, then chopping p will have no effect on the cycle. If p is in an SC-cycle, then all distinct subpieces q and q' of p will be linked to one another by S-edges, and if p is connected by an S-edge to p', then q and q' will both be linked to p'. There are now four cases:

(1) There are two C-edges touching p from the cycle, and both edges touch piece q in pieces(p). Then q takes the place of p in the SC-cycle that p was in before.
(2) There are two C-edges touching p from the cycle: One touches piece q, and the other touches piece q' in pieces(p), respectively. Then the SC-cycle contains q and q', and these two are linked with an S-edge.

(3) There is one C-edge and one S-edge touching p. Suppose the C-edge touches q in pieces(p). Then, q takes the place of p in the SC-cycle that p was in before.
(4) If there are two S-edges touching p, then these edges will touch each piece of pieces(p) so several SC-cycles are created where there was just one before. □

The *Enclosing Conflict Assumption* holds in some application, if whenever operation o conflicts with operation o' then o will conflict with any collection of operations containing o'. This assumption holds in the case where the only conflict information is syntactic (i.e., where a write on a data item conflicts with a read or a write on that item). In models in which one has value-independent semantic information, this assumption holds provided we never chop semantically nonconflicting operations into smaller ones. For example, if we know that increments commute with one another and with decrements, then we should not chop increments or decrements into their component reads or writes. (Chopping an increment i into a read and write would result in a situation where the read conflicts with another increment i' even though i does not conflict with i'.) We call these minimal operations with semantic commutativity properties *primitive accesses*.

LEMMA 3. *Suppose that, in some chopping* $chop_1$, *two pieces, say,* p *and* p', *of transaction instance* T *are in an SC-cycle and that the Enclosing Conflict Assumption holds. Then* p *and* p' *will also be in an SC-cycle in chopping* $chop_2$, *where* $chop_2$ *is identical to* $chop_1$ *with regard to transaction instance* T, *but in which no other transaction instance is chopped (i.e., all other transaction instances are represented by a single piece).*

PROOF. Since p and p' come from T, there is an S-edge between them in both $chop_1$ and $chop_2$. Since they are in an SC-cycle, there exists at least one piece p'' of some transaction instance T'' in that cycle. Merging all pieces of T'' into a single piece (i.e., T'') can only shorten the length of the cycle. The reason is that the C-edge that used to touch a piece of T'' will now touch T'' itself. If there had been S-edges in the cycles between pieces in T'', they would now be removed. (We need the Enclosing Conflict Assumption for this argument. Without the assumption, it might occur that a conflict with a piece of T'' would not necessarily induce a conflict with T'' itself.) The argument applies to every transaction instance other than T having pieces in the cycle. □

Figure 5 illustrates the graph collapsing suggested by this lemma. Putting the three pieces of T3 into one will leave T1 in a cycle and so will chopping T3 further.

Lemmas 2 and 3 lead directly to a systematic method for chopping transactions as finely as possible. Consider again the set of transaction instances $\{T_1, T_2, \ldots, T_n\}$. We will take each transaction instance T_i in turn. We call $\{c_1, c_2, \ldots, c_k\}$ a *private chopping* of T_i, denoted private(T_i), if

(1) $\{c_1, c_2, \ldots, c_k\}$ is a rollback-safe chopping of T_i; and

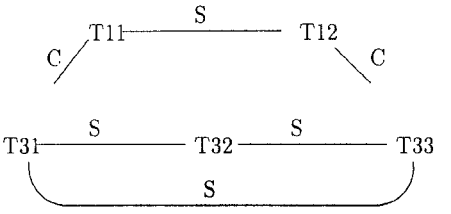

Fig. 5 Graph collapsing, illustrating Lemma 3.

(2) there is no SC-cycle in the graph whose nodes are $\{T_1, \ldots, T_{i-1}, c_1, c_2, \ldots, c_k, T_{i+1}, \ldots, T_n\}$, that is, the graph of all other transaction instances plus the chopping of T_i.

THEOREM 4. *Provided the Enclosing Conflict Assumption holds, the chopping consisting of* $\{private(T_1), private(T_2), \ldots, private(T_n)\}$ *is rollback-safe and has no SC-cycles.*

PROOF. *Rollback-safety*: The chopping is rollback-safe because all of its constituents are rollback-safe.

No SC-cycles: If there were an SC-cycle that involved two pieces of private(T_i), then Lemma 3 implies that the cycle is still present even if all other transaction instances are not chopped. But that contradicts the definition of private(T_i). □

4.2 Algorithm FineChop

Theorem 4 implies that, if we can discover a fine-granularity private(T_i) for each T_i, then we can just take their union. Formally, the *finest chopping* of T_i (whose existence we will prove) is

—a private chopping of T_i;

—if piece p is a member of this private chopping, then there is no other private chopping of T_i containing p_1 and p_2, where p_1 and p_2 partition p and neither is empty.

This suggests the following algorithm:

procedure chop $(T_1, \ldots, T_n)$
  for each T_i
    $Fine_i$:= finest chopping of T_i with respect to unchopped instances of the other transactions.
  end for;
  the finest chopping is
    $\{Fine_1, Fine_2, \ldots, Fine_n\}$

We now give an algorithm to find the finest private chopping of T. The basic idea of this algorithm is to chop T initially into pieces consisting of single primitive accesses. After this, the algorithm finds the connected components of these pieces according to C-edges with respect to the other unchopped transactions. Each connected component becomes a piece. Any finer chopping would result in SC-cycles. If you find the algorithm confusing, try reading the example that immediately follows it.

Algorithm FineChop

initialization:

if there are rollback statements then

p_1 := all database writes of T that may occur before or concurrently with any rollback statement in T

else

p_1 := set consisting of the first primitive database access

end

$P := \{\{x\} | x \text{ is a primitive database access not in } p_1\} \cup \{p_1\}$;

Merge pieces assuming $P = \{p_1, \ldots, p_r\}$ of transaction T:

Let the set $AllButT$ be the set of all transaction except T.

Consider the graph whose nodes consist of P and $AllButT$;

the edges are the conflict edges (C-edges) defined on the nodes.

Construct the connected components of the graph induced by the C-edges.

update P based on the following rule:

for each connected component $p_{e1}, p_{e2}, \ldots, p_{ek}$, if the k pieces of P have the property that $e_1 < e_2 < \cdots < e_k < r$, then put all accesses of $p_{e1}, p_{e2}, \ldots, p_{ek}$ into p_{e1} and then remove $p_{e2}, \ldots, p_{ek}$.

The net effect is that each connected component is replaced by a representative node.

call the resulting partition FineChop(T)

Figure 6 shows an example of a fine-chopping of transaction instance **T5** given a certain set of conflicts and assuming that statements can be reordered in any order. Since there are no rollback statements, each piece starts off being a single access. Assuming no rollback statements, **T5** can be "fine-chopped" into $\{\{a\}, \{b, d, f\}, \{c\}, \{e\}\}$. Indeed, it may seem surprising that three noncontiguous pieces $\{b, d, f\}$ can be merged; in fact, such merging sometimes entail further merging as we discuss in Section 4.3. The main point is that no finer chopping is possible. If $\{b, d, f\}$ were subdivided further, there would be an SC-cycle in the chopping graph.

Remark on Efficiency. The expensive part of the algorithm is finding the connected components of the graph induced by C on all transaction instances besides T and the pieces in P. We have assumed a naive implementation in which the connected components are recomputed for each transaction instance T at a cost of $O(e + m)$ time in the worst case, where e is the number of C-edges in the transaction graph and m is the size of P. Since there are n transactions, the total time is $O(n(e + m))$. In the syntactic conflict model, finding the conflicts can be done in time proportional to sorting all of the variables touched by the transaction programs.

Remark on Code Analysis. We now explain how to obtain the set P of primitive database accesses used in Algorithm FineChop from the analysis of a transaction program. This is straightforward for loop-free code: Construct a one-to-one correspondence between the database accesses of the transaction execution instance and the database operations in the transaction program. For loops, each iteration of the loop is a separate set of database accesses; so

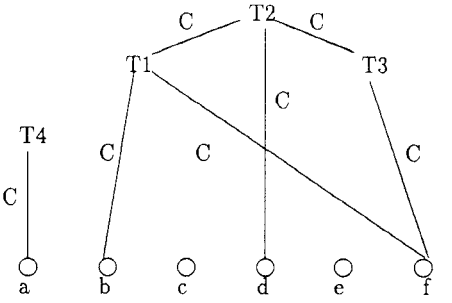

Fig. 6 Fine-chopping of transaction instance **T5**

two iterations may be in the same or different pieces. In most cases, either all of the iterations will be in one piece or all will be in different pieces.

Remark on Shared Code. Until now we have assumed that there is a one-to-one correspondence between transaction instances and transaction programs. The assumption made our discussion easier, but it is time to drop it. In the general case (one or more transaction instances for the same transaction program), we construct the input to Algorithm FineChop as follows: If we know that no two instances of a given transaction program will execute concurrently, we will represent that transaction program once in Algorithm FineChop; otherwise, we will represent that transaction program twice in the algorithm (see Figure 10 for an example). The reason twice is enough is that if there is a cycle in the SC-graph when multiple instances of a transaction program T are present, but not when there is only one such instance, then that cycle must touch two or more accesses of T. In that case, because of the symmetry of C-edges, two instances of the same program text are sufficient to reveal the SC-cycle.

Moreover, since Algorithm FineChop is also symmetric, the two copies of T will be chopped in the same way, so all instances of the transaction program will be chopped in the same way. In sum, we chop programs, and they result in chopped instances. We enclose two copies of the transaction program in Algorithm FineChop only if two or more instances of the transaction may execute concurrently.

Now consider the purchase transaction program of Example 1. Recall that this transaction is as follows:

(1) if (p > cash) {rollback};
(2) else inventory[i] := inventory[i] + p;
(3) cash := cash − p;

The initialization step of Algorithm FineChop yields the pieces:

p_1: Read cash (from line 1);
p_2: Write inventory (from line 2);
p_3: Write cash (from line 3).

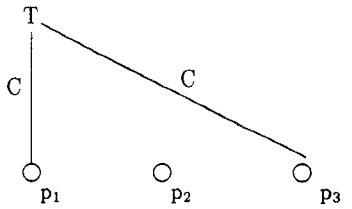

Fig. 7. Graph induced by C-edges.

Suppose we have another purchase transaction T. Then the graph induced by the C-edges is shown in Figure 7. Pieces p_1 and p_3 will be merged. Observe that this chopping does not depend on the number of concurrent purchase transactions considered while running Algorithm FineChop.

THEOREM 5. *FineChop(T) is the finest chopping of T.*

PROOF. We must prove two things: FineChop(T) is a private chopping of T, and it is the finest one.

—*FineChop(T) is a private chopping of T*

(1) *Rollback-safety*: by inspection of the algorithm. The initialization step creates a rollback-safe chopping. The merging step can only cause p_1 to become larger.

(2) *No SC-cycles*: Any such cycle would involve a path through the conflict graph between two distinct pieces from FineChop(T). The merging step would have merged any two such pieces into a single one.

—*No piece of FineChop(T) can be further chopped.* Suppose p is a piece in FineChop(T). Suppose there were a private chopping *TooFine* of T that partitions p into two nonempty subsets q and r.

(1) The accesses in q and r result from the merging step. In that case, there is a path from q to r consisting of C-edges through the other transaction instances. This implies the existence of an SC-cycle for chopping *TooFine*.

(2) Piece p is first piece p_1, and q and r each contain rollback statements of p_1 as constructed in the initialization step. So, one of q or r may commit before the other rolls back by construction of p_1. This would violate rollback-safety. □

4.3 The Dependency Graph for Ordering Pieces

Given the conflict edges, denoted *C-edges*, we apply Algorithm FineChop to obtain the pieces. The question then becomes, How should we execute the pieces of each transaction instance resulting from Algorithm FineChop? In particular, if a piece does not include consecutive database operations, the program may have to be rewritten, or pieces will have to be merged. Recall the purchase transaction program:

(1) if (p > cash) {rollback};
(2) else inventory[i] := inventory[i] + p;
(3) cash := cash − p;

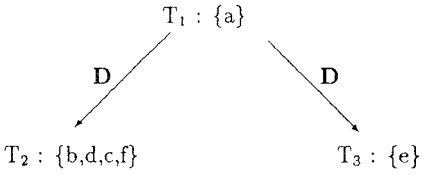

Fig. 8. Example of dependency graph.

In the previous section, we have shown that Algorithm FineChop divides the purchase transaction into two sets of database operations:

P1: {Read cash, Write cash};
P2: {Write inventory};

We now have to decide how to execute these two pieces. P1 must be executed first because it may execute a rollback statement. P2 can be executed after P1 because it is semantically valid to permute lines 2 and 3 in the original purchase transaction program. This yields the following rewriting of the purchase transaction program:

T1: begin T1; if (p > cash) {rollback} else cash := cash − p; commit;
T2: begin T2; inventory[i] := inventory[i] + p; commit;

Note that if lines 2 and 3 were not commutative we would have had to merge **P1** and **P2** into one "superpiece." This section explains how to decide when to perform such merges. Our algorithm entails only analysis of individual transaction programs. Thus, its complexity scales linearly with the number of transactions.

First, we establish data dependencies between pieces inside each transaction instance. These dependencies are represented by a directed graph, called the *dependency graph*, whose nodes are pieces and whose edges, denoted *D_edges*, are defined as follows: There is a *D_edge* from piece p to piece p' if

—piece p has a statement s that "precedes" a statement s' of piece p' (i.e., there exists a path in the precedence graph[7] from s to s'), and s and s' conflict; or

—p is the first piece and contains a rollback statement. (This part about the rollback statement is required for rollback-safety.)

Recall that intertransaction considerations are no longer relevant because Algorithm FineChop already took care of them; *D_edges* concern only intra-transaction dependencies.

The graph induced by *D_edges* may have a cycle. For example, consider Figure 6. Assume that every single piece $a, b, \ldots$ consists of a single SQL statement. If b precedes c and c precedes d in the original transaction program and if c conflicts with b and d, then there is a *D_edge* from $\{b, d, f\}$ to $\{c\}$, and vice versa. In this case, the two pieces cannot be executed in arbitrary concurrent order. We have to *merge* the two pieces into one consisting of $\{b, c, d, f\}$. The obtained dependency graph is shown in Figure 8

[7]The precedence graph represents the precedence relationship defined in Section 2.

(we have assumed that a contains a rollback statement). In general, all pieces that are in a directed cycle of D_edges must be merged into a single *superpiece*.

There is no cycle between two iterations of a same program loop. However, if a piece contains the first and the last iteration of a single loop, all of the iterations of the loop must be merged with this piece.

We summarize these considerations into the following algorithm:

Algorithm Preserve_Order(P, T).

(P is the finest chopping of a transaction instance T)
Construct the dependency graph (P, D_edges);
Reduce each cycle in the dependency graph to a new superpiece, and cause the superpiece to inherit the dependencies from its constituents
Superpieces will be executed according to their dependency graph;
The order of statements within a superpiece will be the original relative order of the statements in T.

The critical part of this algorithm is the detection of conflicting statements with precedence relationships. This can be done by using techniques combining data and control-flow analysis of the transaction program, such as those presented by Aho et al. [1986], Weiser [1984], and Agrawal [1994]. The more effective they are, the less numerous the D_edges will be.

The resulting dependency graph can be used to reorganize the transaction programs: One can execute the superpieces in any order consistent with the dependency edges. That is, if there is a dependency edge from superpiece p to superpiece p' then all of the statements of p must precede the statements of p' in the new program execution order. On the other hand, two superpieces of a transaction instance T can be executed in parallel or their order can be reversed, provided that there is no dependency edge between them.

A dynamically parallel approach consists of executing a superpiece s as soon as all superpieces have D_edge arcs pointing to s complete. Thus, chopping can enhance intratransaction parallelism, as well as reduce the time locks are held. In our example (see Figure 8), **T2** and **T3** can be executed in parallel.

Each superpiece will be managed as an independent transaction instance, committing its changes when it finishes. If a superpiece aborts due to a deadlock, then it is reexecuted by embedded SQL code that detects deadlock-induced error codes and retries the superpiece. If a superpiece aborts due to a system failure, then we use the "minibatch trick" illustrated in Example 2 of Section 1.

5. APPLYING THESE RESULTS TO TYPICAL DATABASE SYSTEMS

For us, a typical database system will be one running SQL. Our main problem is to figure out what conflicts with what. Because of the existence of bind variables, it will be unclear whether a transaction instance that updates the account of customer **:x** will access the same record as a transaction instance that reads the account of customer **:y**. So, we will have to be conservative.

We can use the tricks of typical predicate locking schemes, as pioneered in System R and then elaborated by Wong and Edelberg [1973] and by Dadam et al. [1983]. For example, if two statements on relation account are both conjunctive (only **AND**s in the qualification) and one has the predicate

AND name LIKE 'T%'

whereas the other has the predicate

AND name LIKE 'S%'

they clearly will not conflict at the logical data item level. (This is the only level that matters since it is the only level that affects the return value to the user.) Detecting the absence of conflicts between two qualifications is the province of compiler writers. We offer nothing new.

The new idea we have to offer is that we can make use of information in addition to simple conflict information. For example, if there is an update on the account table with a conjunctive qualification and one of the predicates is

AND acctnum = :x

then, if **acctnum** is a key, we know that the update will access at most one record. This implies that a concurrent reader of the form

SELECT...
FROM account
WHERE...

will conflict with the update on at most one record. So, the **SELECT** transaction can execute at degree 2 isolation. (Degree 2 isolation has the effect of chopping the **SELECT** transaction instance into pieces, each of which consists of a single data item access.) In fact, even if many updates of this form are concurrent with a single instance of the **SELECT** transaction, the reader can still execute at degree 2 isolation, because no SC-cycle will be possible.[8] On the other hand, if two instances of the **SELECT** transaction can execute concurrently, then they cannot execute in isolation because an SC-cycle is possible (see Figure 9).

Remark about Multiversion Read Consistency. Several DBMSs (e.g., ORACLE, GemStone, and RDB) offer a facility known as *multiversion read consistency*. For concreteness, we adopt the syntax of ORACLE. ORACLE **SELECT** statements (as opposed to **SELECT** for update statements) acquire no locks. Instead, they appear to execute when the enclosing transaction instance began: They return values based on the data state at the time of the first SQL statement of the transaction. If T is entirely read-only, then multiversion read consistency ensures serializability. If T contains writes, however, then using multiversion read consistency for selects may lead to nonserializable executions such as **begin(T1) R1(x) begin(T2) W2(x) W2(y) commit(T2) W1(y) commit(T1)**.

[8] This does not hold for insertions, however Concurrent insertions of single records could result in a phantom.

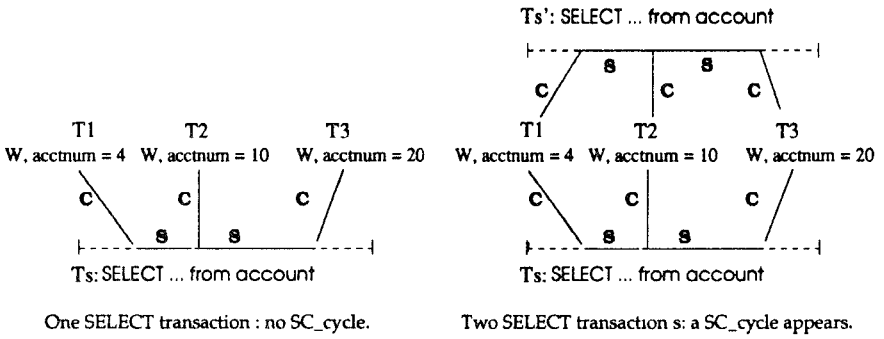

Fig. 9 Applying chopping to execute **SELECT** transactions at degree 2 isolation.

From the point of view of chopping theory, multiversion read consistency puts all reads that use it into one piece and all other operations into another piece. So Algorithm FineChop will tell us when and how much multiversion read consistency to use.

For example, consider a situation in which one transaction program has the form

SELECT...
FROM account
WHERE...
UPDATE balance...

producing a set of possibly concurrent transaction instances we call Group 1; and another transaction program updates possibly several records of account, producing a set of transaction instances we call Group 2. There are no other transactions. Using multiversion read consistency for the Group 1 transactions will not work because there could be a conflict between the **SELECT** piece of a Group 1 transaction instance T and a Group 2 transaction instance, and a second conflict between the Group 2 transaction and the **SELECT** piece of a second Group 1 transaction T'. The **UPDATE** piece of T' might then conflict with the **UPDATE** piece of T. If, however, Group 1 contained only a single transaction instance (i.e., no two instances of the first transaction could execute concurrently), then using multiversion read consistency for Group 1 **SELECT**s would work, because the conflicts with Group 2 transactions yield cycles having only C-edges (see Figure 10).

6. EXPERIENCE WITH CHOPPING ON A REAL DBMS

We have implemented the chopping algorithm as a preprocessor for SQL-86 programs. To simulate its applicability in real life, we have tested its performance on a variant of the AS^3 AP [Gray 1991]) benchmark using a commercial DBMS (ORACLE[9] version 6). This section describes our SQL

[9]ORACLE is a trademark of Oracle Corporation.

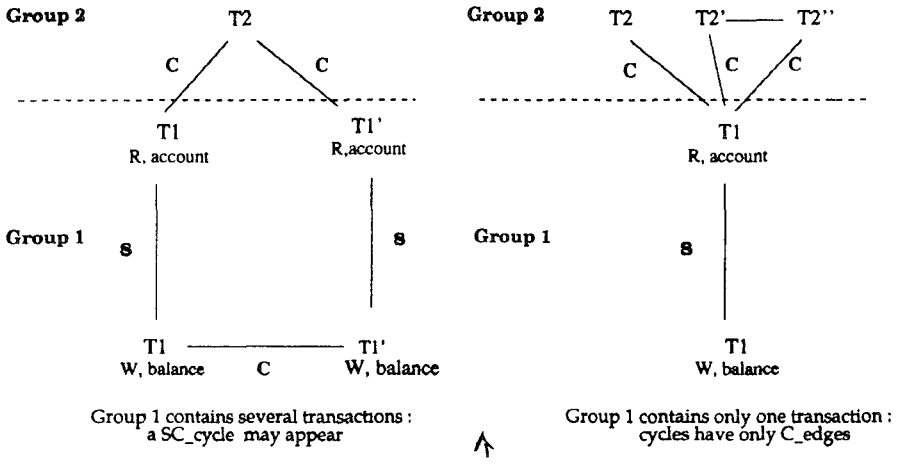

Fig. 10. Applying chopping to multiversion read consistency: The **SELECT** statements using multiversion read consistency constitute one piece of the transaction instances.

implementation and our experimental results, and discusses the performance trade-offs of chopping.

6.1 Benchmark Experiment on a Real System

We consider the situation where a long transaction instance that updates a large amount of data is run concurrently with many short conflicting transactions, each of which randomly updates a single tuple. Our goal is to quantify the value of chopping the long transaction while maintaining serializability. Our hypothesis is that we should get a higher transaction throughput for the short transactions and a slightly lower throughput for multiple instances of the long transaction. Our test executes on a relation called "updates." The schema of the updates relation is the following: updates (*key* integer, *int* integer, *signed* signed integer, *double* double precision). Attribute *key* is the primary key in the relation.

The long transaction, denoted LT, updates tuples with an even key. It is of the form

```
BEGIN TRANSACTION
EXEC SQL mod_550_seq; EXEC SQL unmod_550_seq;
EXEC SQL commit;
END TRANSACTION
```

where **mod_550_seq** is the query

```
update updates set double = double + 10000000
where key between 100 and 1200 and mod(key, 2) = 0;
```

and **unmod_550_seq** is the query

```
update updates set double = double − 10000000
where key between 100 and 1200 and mod(key, 2) = 0;
```

There are two kinds of short transaction instances in our tests:

—the short conflicting transactions, denoted STC, which update a record in common with the LT transaction; and

—the short nonconflicting transactions, denoted STNC, which do not update any record in common with the LT transaction.

The STC transactions are of the form

```
BEGIN TRANSACTION
EXEC SQL oltp_update_Conflict; EXEC SQL commit;
END TRANSACTION
```

where **oltp_update_Conflict** is as follows:

```
update updates set double = 0
where key = random_number(100, 1200) and mod(key, 2) = 0;
```

The STNC transactions are of the form

```
BEGIN TRANSACTION
EXEC SQL oltp_update_NConflict; EXEC SQL commit;
END TRANSACTION
```

where **oltp_update_NConflict** is as follows:

```
update updates set double = 0
where key = random_number(100, 1200) and mod(key, 2)! = 0;
```

In our tests, we generate random numbers uniformly in the interval [100..1200]. ORACLE's locking granularity is tuple-level, so the STNC will never have to wait for a lock held by LT, though it will have to wait for short page-level latches (latches are released immediately after a page is updated, rather than at the end of the enclosing transaction).

We vary three parameters:

(1) the number of short, conflicting transaction instances, denoted nb_STC;

(2) the number of short nonconflicting transaction instances, denoted nb_STNC; and

(3) the number of pieces into which the long transaction instance is decomposed, denoted N.

We perform our tests in two stages. First, the long transaction program is chopped into N pieces. (Algorithm FineChop would allow us to make a transaction out of each tuple access so N can be as great as 550.) The long transaction is built up as the succession of the pieces encoded as embedded SQL subprograms and separated by **COMMIT** statements.

The total number of concurrent users is $1 + nb_{STC} + nb_{STNC}$, where the 1 stands for the long transaction. Each user is modeled as a UNIX[10] process that exercises its transaction type in a loop. The execution continues for a

[10] UNIX is a registered trademark of AT & T Bell Laboratories.

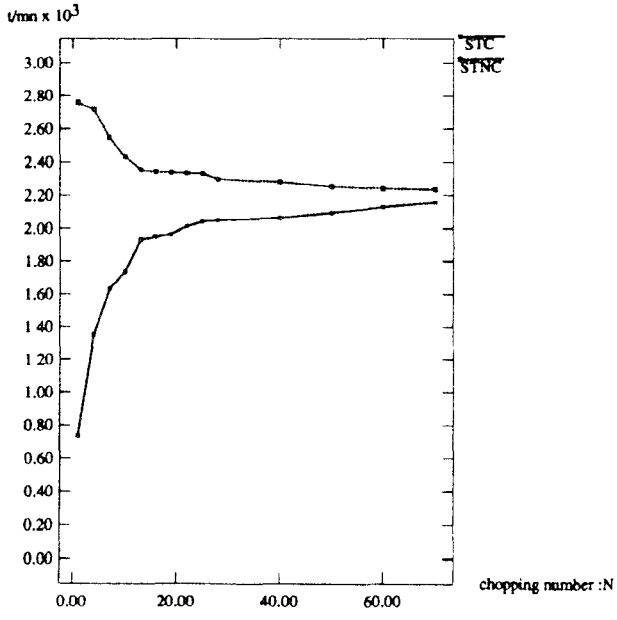

Fig. 11. Throughputs of STC and STNC.

predefined time T, yielding a transaction throughput for the long and short transactions.

6.2 Results

The experiments use ORACLE version 6.0.30 running under B.O.S. 2.0.0 on a BULL DPX20. We used dedicated disks for the log and the database in order to reduce seek times and rotational delays for writing the log.

The results of a first set of experiments are shown in Figures 11–13. Figure 11 shows that chopping the LT program significantly increases the throughput of STCs. In fact, the throughput of STCs comes close to the throughput of STNCs. That is the good news. The bad news is that the number of LTs that can be executed within time T decreases as N increases (Figure 12). The throughput of the STNC also decreases, probably due to resource contention.

Intuitively, chopping improves the performance of STC instances for two reasons: (1) The maximum number of locks held at one time by the LT decreases. Consequently, the probability that an STC requests a lock held by an LT decreases. (2) The LT releases its locks earlier; the lock waiting time of the STC becomes shorter when a conflict occurs.

Chopping also imposes two costs on LT (Figure 12): (1) It entails more log writes because each piece commits separately (though group commits would mitigate this cost if available), and (2) each log write causes a context switch.

Furthermore, since STCs, STNCs, and LTs are sharing the same resources (CPUs and disks), the increasing number of active STCs in the system affects the performances of STNCs and LTs.

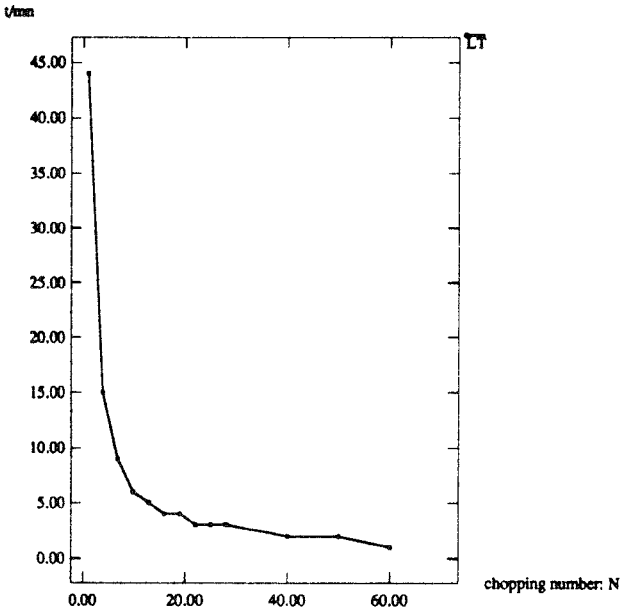

Fig. 12. Throughput of LT.

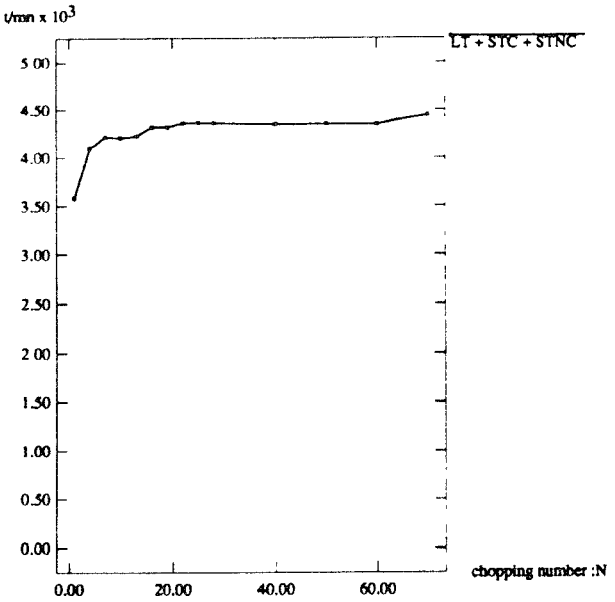

Fig. 13. Global throughput.

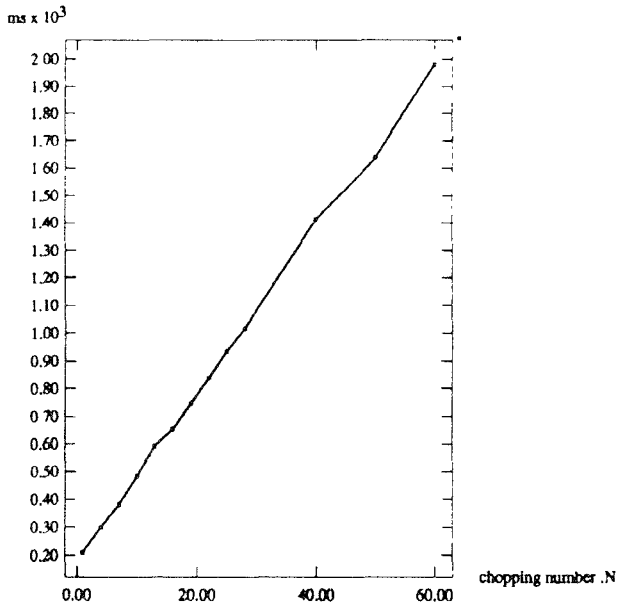

Fig. 14. Commit costs.

What is the bottom line? The short transactions have been made faster at some cost to the longer transactions. This is a reasonable trade-off since in most applications the short transactions require the best response time. There is, moreover, an overall improvement of the global throughput resulting from chopping, as Figure 13 shows (20 percent improvement). The biggest improvement is given at relatively low chopping numbers. The global throughput curve reaches a plateau after a chopping number of 20.

One of us proposed an average case performance model for chopping [Llirbat 1994]. Here, we briefly describe that model and show how it applies to this example.

As in Tay's [1987] work, we model the probability of conflict as proportional to the average number of items held locked (in conflicting lock modes) at any time. The expected waiting time of a transaction instance T if it does encounter a lock conflict is proportional to the size of the transaction instance for which T has to wait. Thus, to a first approximation and in the absence of resource contention, the expected waiting time is proportional to the square of the average number of items held locked. (The approximation assumes that the duration of a transaction instance is proportional to the number of locks it obtains, a reasonable assumption for LT in this case.)

Since the average number of items held is inversely proportional to the number of pieces of the chopping, the expected waiting time is proportional to $1/(choppings^2)$, in the absence of resource contention and assuming a logging protocol such as writeahead logging [Gray and Reuter 1992].

On the other hand, chopping's costs are linearly proportional to the number of pieces, as shown in Figure 14. As mentioned above, this cost can be

reduced by group commits that combine the commits of several transaction instances into one write.

7. SIMULATION

In this section we evaluate the performance of chopping in the general case where random transactions are running concurrently on a centralized database. To test the effects of chopping in a wide range of operating conditions, we have implemented a simulation model and validated it against the ORACLE experiments of the last section. The simulation model accurately reflects the relative benefits and costs of chopping in those experiments, though not the exact numbers.

The simulation results yield the following conclusions:

(1) Chopping a transaction decreases the waiting time of the other conflicting transactions, and thus decreases lock contention and improves the throughput of the concurrent transactions. In particular, the obtained results show that chopping transactions can delay or prevent lock contention thrashing, due to lock contention.

(2) This performance improvement is reduced by two resource effects: (1) the added commit cost and (2) the increased resource contention (waiting for CPUs and disks).

As we will see, performance analysis of chopping in different operating situations helps us to describe a feedback-based approach to find a suitable chopping number.

7.1 The Simulation Model

The database simulation model is based on Carey 1983, Agrawal et al. 1987, Haritsa et al. 1990, and Agrawal et al. 1992. The main difference is the simulation of the log manager operations at commit time to take into account the cost of commits. This model was developed using the SIM package [Adler et al. 1992]. It is divided into four main components: a transaction manager (TM), a concurrency control agent (CCA), a data manager (DM), and log manager (LM). The TM is responsible for issuing lock requests and their corresponding database operations. It also provides the durability property by flushing all log records of committed transactions to durable memory. In case of failure or aborted transactions, the TM reads these log records in order to ensure the atomicity property using undo–redo operations. The CCA schedules the lock requests according to the two-phase locking protocol. The DM is responsible for granting access to the physical data objects and executing the database operations. The LM provides read and insert-flush interfaces to the log table.

Figure 15a represents a closed querying model of the execution of chopped transactions on a single-site database. Chopped transaction instances (CTIs) are generated from the terminals. A CTI is decomposed into N pieces, which are executed in a serial order as chained transactions. The *intra_CTI_delay* parameter represents the mean time delay between the completion of a piece

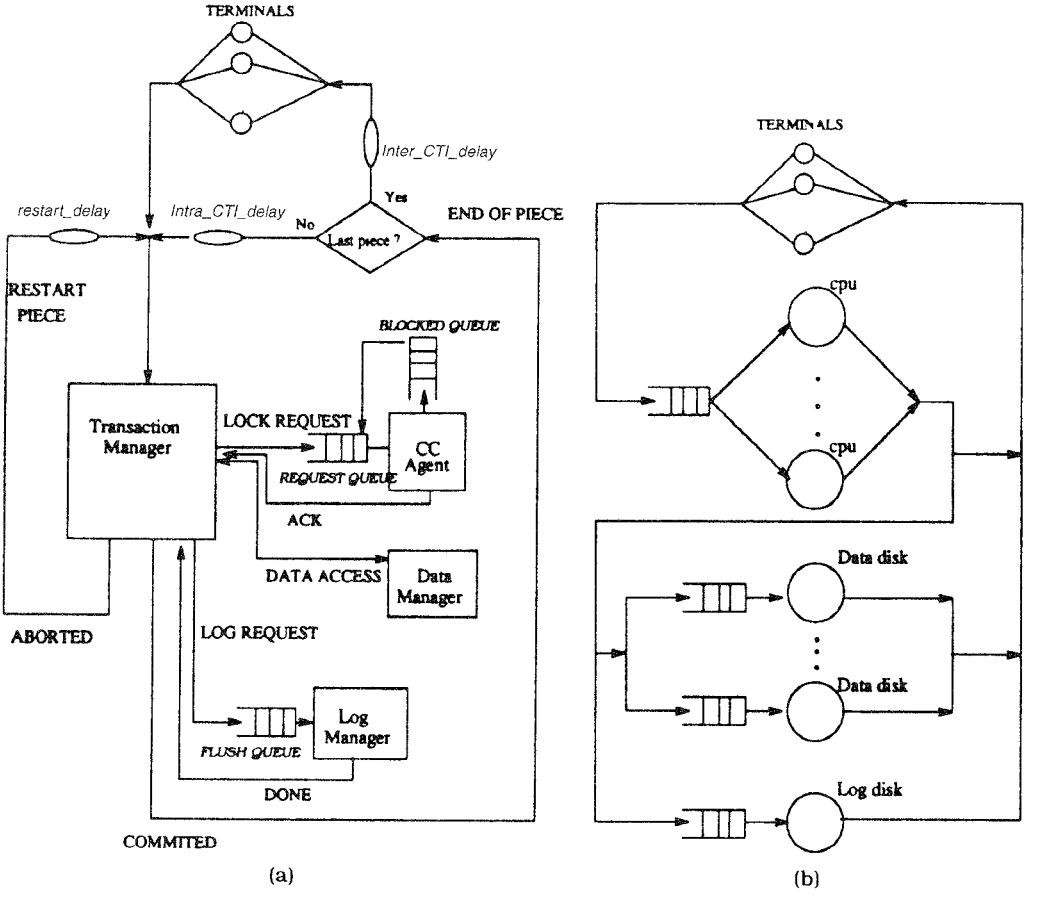

Fig. 15 Simulation model. (a) Logical queueing model. (b) Physical queueing model.

and the submission of the next piece. If a piece is aborted because of a lock conflict, then it is repeatedly restarted until it commits. The parameter *restart_delay* represents the mean time between the abort of a piece and its restart. A CTI completes when its last piece is committed. The parameter *inter_CTI_delay* represents the time delay between the completion of a CTI and the initiation of a new CTI from a terminal.

The pieces are executed as normal transactions by the TM. The TM sends lock requests to the CCA's request queue, and the CCA processes these requests. If the transaction must block, it enters the *blocked_queue* until the conflicting operations have released their locks. We use a deadlock-detection strategy based on a waits-for graph. Whenever a deadlock is detected, the youngest transaction is aborted. When a lock request can be granted, an acknowledgment is sent back to the TM, which then forwards the database operation to the DM. At commit time, the LM receives a *Log_flush* request from the TM through the *flush_queue*. The LM uses a group **COMMIT**

algorithm[11] to execute all waiting flush requests with a single IO. When the log records are written to durable storage, the LM sends a message back to the TM, which releases locks and completes the transaction with the **COMMITTED** message. In case of abort, the TM sends a *Log_read* request to the LM, executes the **undo** operations, releases the locks, and ends the transactions with the **ABORTED** message. For convenience, we assume in our simulation that a *Log_read* request never requires IO.[12]

The physical queueing model, shown in Figure 15b, is quite similar to the one used by Carey [1983], Agrawal et al. [1987], and Agrawal et al. [1992], in which the parameters *num_cpus* and *num_disks* specify the number of CPU servers and the number of IO servers. The requests to the CPU queue and the IO queues are serviced FCFS (first come, first serve). The parameter *obj_cpu* is the amount of CPU time associated with executing a database operation on a single object. The parameter *page_io_access* is the amount of IO time associated with accessing a data page from the disk. The *io_prob* parameter is the probability that a database read operation on an object forces a data page IO. We add to this model one separate IO server that is dedicated to the log file. The parameter *Log_disk_io* represents the fixed IO time overhead associated with issuing the IO. The parameter *Log_rec_io_w* is the amount of IO time associated with writing a log record on the log disk in sequential order. The *commit_cpu* parameter is the amount of CPU time associated with executing the **COMMIT** (releasing locks and so on). The *abort_cpu* parameter is the amount of CPU time associated with executing the **ABORT** statement (executing **undo** operations, releasing locks, and so on).

Each simulation consists of a number of repetitions. A repetition is performed in three stages: (1) the generation of random transaction programs, (2) the decomposition of these programs in pieces using the chopping algorithm, and (3) the simulation of the execution of the obtained CTI. The simulation time was fixed at 1000 s. The number of repetitions was chosen to achieve 90 percent confidence intervals for our results (30 repetitions). Table I summarizes the parameters and their values in the simulation model and experiments.

7.2 Simulation Results

In our experiments, we vary the multiprogramming level (number of concurrent transaction programs) from 1 to 100 (one program per terminal). This allows us to obtain a wide range of operating conditions with respect to conflict probability and resource contention. The curve in Figure 16 shows that when we increase the multiprogramming level (*mpl*) the number of conflicting operations per transaction[13] increases. In Figure 17 we measure the throughput of concurrent transactions running without concurrency con-

[11] We implemented a zero-wait timer group commit [Helland et al. 1987].
[12] Reading the log file usually requires no IO (see Gray and Reuter 1992, p. 505)
[13] A data operation is conflicting if there exists a concurrent transaction that accesses the same object.

Table I. Simulation Model Parameter Definitions and Values

Parameter	*Description*	*Value*
db_size	Number of objects in database	20,000 objects
num_terms	Number of terminals	1–100 terminal
N	Chopping number	1–8
txn_size	Transaction size	80 operations
write_op_pct	Write operation percentage	40%
num_cpus	Number of CPUs	*k* CPUs
num_disks	Number of data disks	*k* data disks
k	Resource unit	2–4
obj_cpu	CPU time for executing an operation	1 ms
page_io_access	IO time for accessing a page	7 ms
io_prob	Probability of IO page access	0.2
Log_disk_io	Time for issuing an IO log access	7 ms
Log_rec_io_w	IO time for writing 1 page on log disk	0 1 ms
commit_cpu	CPU time for executing a **COMMIT**	2 ms
Abort_cpu	CPU time for executing an **ABORT**	2 ms
inter_CTI_delay	Time between two CTIs	10 ms
intra_CTI_delay	Time between two CTI's pieces	5 ms
restart_delay	Restart delay of one CTI's piece	5 ms

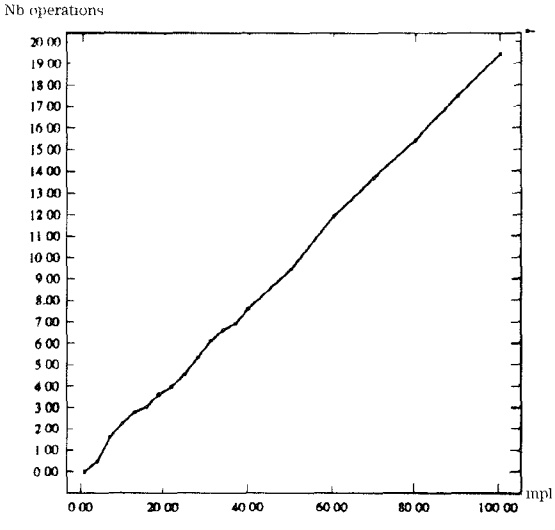

Fig. 16 Conflicts.

trol. This figure shows how resource contention limits the throughput of transactions. A plateau is reached at a multiprogramming level of 20 because the resources are saturated.

We now present the results of the simulation experiments when the long transaction is chopped into 1 (no chopping), 2, 4, 6, and 8 pieces. Figures 18 and 19 illustrate, respectively, the throughput and the mean response time of the concurrent transactions. They show that chopping transactions improves

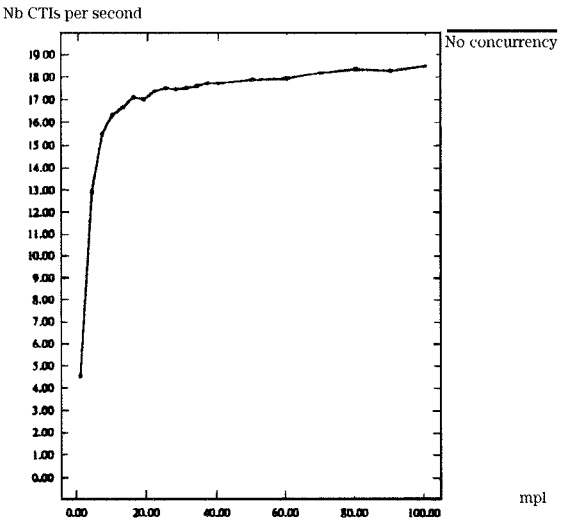

Fig. 17. Effects of resource sharing.

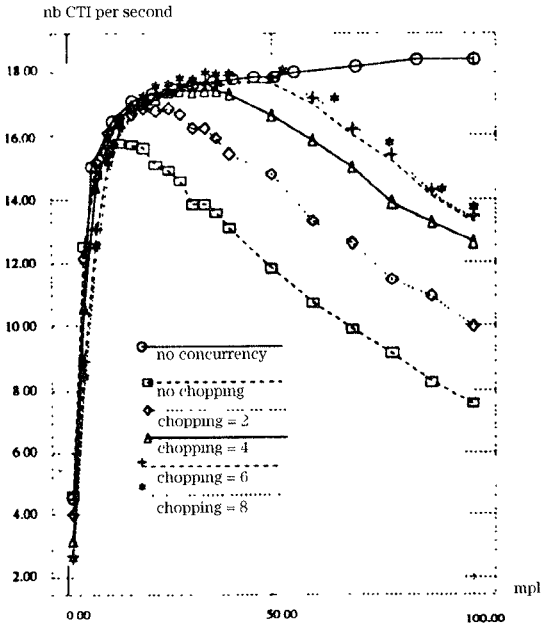

Fig. 18. CTI's throughput.

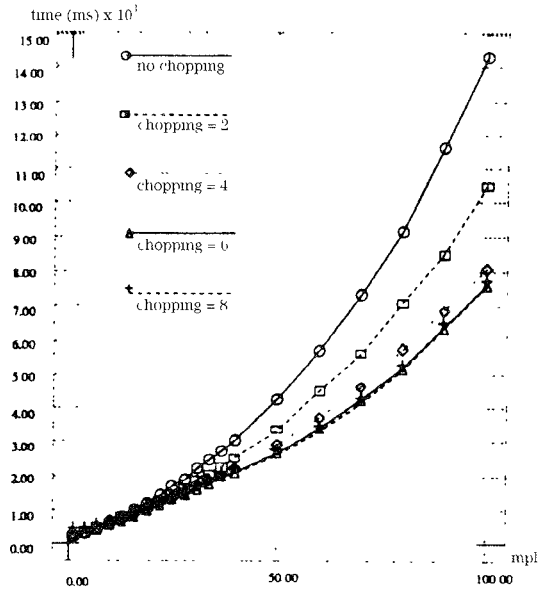

Fig. 19 CTI's mean response time.

the performance significantly. Chopping = 8 gives the best performances. Note that from $mpl = 20$ to $mpl = 60$ the obtained throughput is quite close to the maximal throughput indicated by the "no concurrency" curve.

The best improvements of performance (120 percent with $mpl = 100$) are provided by chopping with large multiprogramming levels when the throughput of the nonchopped transactions is in a thrashing situation. Furthermore, Figure 18 shows that chopping transactions delays the thrashing point to larger multiprogramming levels. Specifically, the thrashing situation appears at $mpl = 20$ if there is no chopping, at $mpl = 30$ when chopping is equal to 2, at $mpl = 40$ when chopping is equal to 4, at $mpl = 50$ when chopping is equal to 6, and after $mpl = 55$ when chopping is equal to 8. As observed and explained in several studies [Tay 1987; Thomasian and Ryu 1991; Thomasian 1991; Ryu and Thomasian 1990; Carey et al. 1990], the thrashing behavior is caused to a large extent by system underutilization due to transaction blocking, and to a more limited extent by wasted processing caused by transaction aborts. In Figure 20 we measure the mean time during which each transaction is waiting for a lock. We observe that the mean waiting time increases with the mpl.[14] This effect is reduced by chopping. We also observe that chopping reduces the wasted processing caused by aborts by reducing the size of pieces to restart (30 percent reducing with $mpl = 100$).

[14]As explained by Thomasian [1991], the strong increase of the mean waiting time is essentially due to cascading waits.

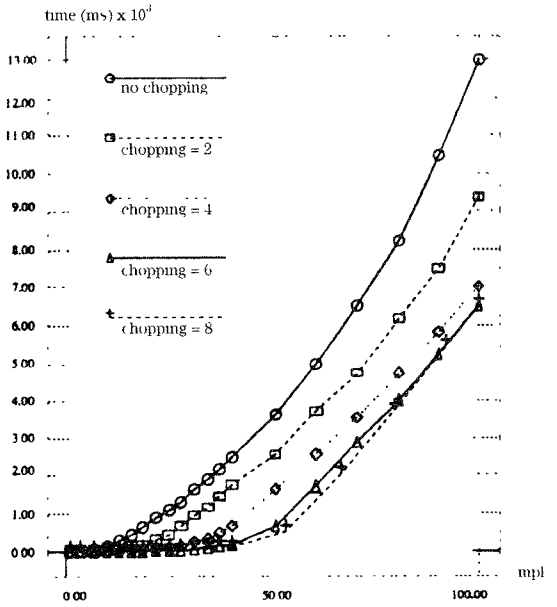

Fig. 20. Mean waiting time.

On the other hand, chopping does not help much when the *mpl* is low. There are two reasons for this: (1) The probability that a transaction is involved in a conflict is small, eliminating the need to chop; and (2) since the marginal gain of chopping is small, effects of added commit cost are noticeable. The mean **COMMIT**-time graph (see Figure 21) illustrates the added **COMMIT** cost due to chopping. The figure shows that chopping increases the **COMMIT** cost in a linear manner. On the other hand, the **COMMIT** cost is not very sensitive to the number of concurrent transactions. This is due to the fact that we use group **COMMIT**.

To evaluate the effects of resource constraints on the performance of chopping, we conducted new experiments with the number k of resource units (the number of CPUs and disks in the system[15]) increased from two to four. Figure 22 reports the obtained throughputs. The benefit of chopping is significantly greater when more resources are provided. In particular, the maximal throughput for $k = 4$ obtained by chopping the CTIs in eight pieces represents a 39 percent improvement, compared to the maximal throughput for $k = 4$ (see Figure 22b) obtained when there is no chopping. The maximal throughput for $k = 2$ (see Figure 22a) obtained by chopping into eight pieces represents only a 12 percent improvement, compared to the maximal throughput for $k = 2$ obtained when there is no chopping. Hence, chopping transactions is particularly useful if there is less resource contention. Indeed, chopping transactions decreases the lock contention and so increases the

[15]See Table I.

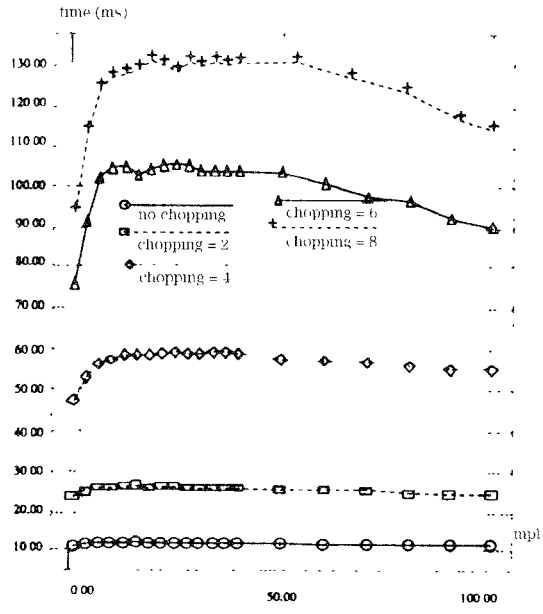

Fig. 21. Mean **COMMIT** time.

number of active transactions in the system. If there is no resource contention, these transactions are executed efficiently. On the other hand, if resources are saturated these transactions have to wait for available resources to be executed, and the global throughput is not improved.

7.3 Rule-of-Thumb Lessons from These Experiments

As already observed in the experiments on ORACLE, the simulation results show that low chopping numbers give the biggest benefit. Furthermore, the performance analysis in the previous section allows us to point out three kinds of situations where increasing the chopping number is useless:

(1) *There is resource contention.* Once again, if resources are saturated, increasing the number of active transactions will not improve the global throughput (compare Figure 22a and b).

(2) *There is minimal lock contention.* In these situations the marginal gain of chopping is too small because the probability of conflict is small (see Figure 18, when *mpl* is lower than 15).

(3) *The chopping algorithm chops only pieces involved in no conflicts.* For example, according to Figures 18 and 22, it is ineffective to chop transactions from 6 to 8 pieces.

These considerations lead to a simple method to obtain a suitable chopping number:

(1) Evaluate the system load.

(2) Evaluate the lock contention.

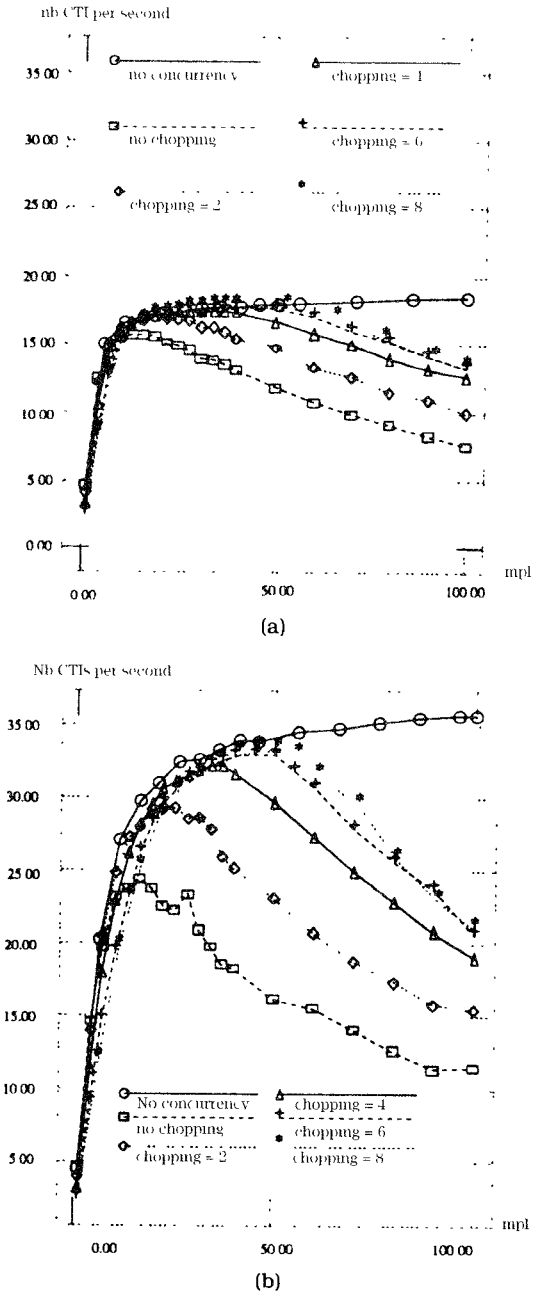

Fig. 22. Effects of resources. (a) Resource units = 2. (b) Resource units = 4.

(3) If the system is underutilized because of lock contention, then

—find the pieces that are involved in a lot of conflicts, and
—try to chop them.

(4) If resources are heavily loaded or if the only choppable pieces have no conflicts with other pieces, then stop chopping. (The reason for the last part is that chopping such a nonconflicting piece only adds to **COMMIT** and transaction starting overhead without improving throughput.)

8. RELATED WORK

There is a rich body of work in the literature on the subject of chopping up transactions or changing concurrency-control mechanisms, some of which we review here, although the work is not strictly comparable. The reason is that this paper is aimed at database users, rather than DBMS implementors. Database users normally cannot change the concurrency-control algorithms of the underlying system, but must use two-phase locking and its variants. Even if users could change the concurrency-control algorithms, they probably should avoid doing so, as the bugs that might result can easily corrupt a system.

The literature offers many good ideas, however. Here is a brief summary of some of the major relevant contributions.

Farrag and Ozsu [1987] proposed a concurrency-control algorithm that generalizes both two-phase locking and time-stamp ordering. Whereas our chopping paper has been oriented toward two-phase locking, their framework might permit chopping approaches to time-stamping as well.

Garcia-Molina [1983] suggested using semantics by partitioning transactions into classes. Transactions in the same class can run concurrently, whereas transactions in different classes must synchronize. He proposed using semantic notions of consistency to allow more concurrency than serializability would allow and using counterstep transactions to undo the effect of transactions that should not have committed.

Lynch [1983] generalized Garcia-Molina's model by making the unit of recovery different from the unit of locking (this is also possible with the checkout/checkin model offered by some object-oriented database systems).

Farrag and Ozsu [1989] also considered the possibility of chopping up transactions by using "semantic" knowledge and a new locking mechanism. For example, consider a hotel reservations system that supports a single transaction Reserve. Reserve performs the following two steps:

(1) Decrement the number of available rooms, or roll back if that number is already 0.
(2) Find a free room, and allocate it to a guest.

If reservation transactions are the only ones running, then the authors observed that each reservation can be broken up into two transactions, one for each step. Our mechanism might or might not come to the same conclusion, depending on the operation semantics known. To see this, suppose that

the variable A represents the number of available rooms, and r and r' represent distinct rooms. Suppose we can represent two reservation transactions by the following:

T1: RW(A) RW(r)
T2: RW(A) R(r) RW(r′)

Chopping these will result in

T11: RW(A)
T12: RW(r)
T21: RW(A)
T22: R(r) RW(r′)

which will create an SC-cycle because of the conflicts on A and r. However, the semantics of the hotel reservation system tell us that it does not matter if one transaction decrements A first but gets room r'. That would suggest that **T12** and **T22** in fact commute and should be construed as primitive accesses. If that is the case, then there is no SC-cycle. The authors noted in conclusion that finding semantically acceptable interleavings is very hard. It is possible that chopping would make it easier.

Agrawal et al. [1994] proposed a scheme that unifies the semantic approach of Farrag and Ozsu with the relative atomicity approach of Lynch. Using semantic relative atomicity as a correctness criterion can lead to finer choppings.

Bayer [1986] showed how to change the concurrency-control and recovery subsystems to allow a single batch transaction to run concurrently with many short transactions. The results are consistent with chopping theory.

Hsu and Chan [1986] examined special concurrency-control algorithms for situations in which data are divided into raw data and derived data. The idea is that the recency of the raw data is not so important in many applications, so updates to those data should be able to proceed without being blocked by reads of the data. That approach does not guarantee serializably, so chopping would not help.

O'Neil [1986] took advantage of the commutativity of increments to release locks early even in an environment of concurrent writes. From the chopping point of view, the increment is a separate piece.

Wolson [1987] presented an algorithm for releasing certain locks early without violating serializability based on an earlier theoretical condition given by Yannakakis [1982]. He assumed that the user has complete control over the acquisition and release of locks. Using different formalisms and different algorithms, Lausen et al. [1986] also examined chopping algorithms under the assumption of complete control over locks. The setting here was a special case of those algorithmic approaches: The user can only control how to chop up a transaction or whether to allow reads to give up their locks immediately. As mentioned above, we have restricted the user's control in this way for the simple pragmatic reason that systems restrict the user's control in the same way.

Bernstein et al. [1980] introduced the idea of conflict graphs in an experimental system called SDD-1 in the late 1970s. Their system divided

transactions into classes such that transactions within a class executed serially, whereas transactions between classes could execute without any synchronization.

Casanova's [1981] thesis extended the SDD-1 work by representing each transaction by its flowchart and by generalizing the notion of conflict. A cycle in his graphs indicated the need for synchronization if it included both conflict and flow edges.

Shasha and Snir [1988] explored graphs that combine conflict, program flow, and atomicity constraints in a study of the correct execution of parallel shared-memory programs that have critical sections. The graphs used there were directed, since the only correctness criterion is one due to Lamport known as sequential consistency.

In summary, any approach that attempts to preserve serializability without changing the system concurrency mechanism can be used in combination with chopping. Approaches that relax serializability may also use chopping as an additional mechanism [Hseuh and Pu 1993].

9. CONCLUSION

We have proposed a simple, efficient algorithm to partition transaction programs into the smallest pieces possible with the following property:

If the small pieces of transaction instances execute serializability, then the transaction instances will appear to execute serializably.

This permits database users to obtain more concurrency and intratransaction parallelism without requiring any changes to database-system locking algorithms. The only information required is some characterization of the transaction instances that can execute during a certain interval and the location of any rollback statements within the transaction. Information about semantics may help by reducing the number of conflict edges.

Our experiments on a real system using the AS^3AP benchmark have shown that chopping improves performance by reducing lock contention, though it increases resource contention. These observations were confirmed in more general situations by using a simulation model. Moreover, the simulation results have shown that chopping transactions can be a good method to prevent thrashing due to lock contention.

We note that read transactions show no trade-off: Using degree 2 isolation always yields better performance. (This may be why using chopping to show that degree 2 isolation is sufficient can make you popular with your clients.)

Several interesting problems remain open:

—Might chopping help the design of multidatabase concurrency-control methods? We think so, because a multidatabase transaction may consist of many separately committed pieces. Better performance would result if some could commit even while others were still executing. Chopping can say when this is possible.

—How does chopping work in the context of parallel transactions where backing up one piece of a transaction may create **ABORT** dependencies with respect to other pieces?

—How does chopping interact with relaxed concurrency-control methods that make use of semantic approximation? Some work has already been started on combining chopping with epsilon serializability [Hseuh and Pu 1993].

—A pragmatic question: Suppose that an administrator asks how best to partition transaction populations into time windows, so that the transactions in each window can be chopped as much as possible. For example, a good heuristic is to put global update transactions in a partition by themselves while allowing point updates to interact with global reads. What precise guidance could theory offer? The general pragmatic question is: What is a good architecture for incorporating chopping among the tuning knobs for the DBMS?

—Finally, if we could change the underlying DBMS, then we could ask the transaction load controller to detect the formation of the SC-graph dynamically. This presents promising opportunities for optimization.

ACKNOWLEDGMENTS

We would like to thank Gerhard Weikum for his astute comments regarding order-preserving serializability and intratransaction parallelism, Victor Vianu for a bus-ride discussion concerning transitive closure, and Rick Hull for initial discussions concerning partitioned accesses.

We would also like to thank Elisabeth Baque for applying her artistry to make the figures of the manuscript and Fabienne Cirio for her initial drafts of those figures. Many thanks to Willy Goldgewicht and Bertrand Betonneau for allowing us to do the experiments on ORACLE, to Dimitri Tombroff for his effective contribution to the implementation of these tests, and to all of the ORASCOPE BULL team for its generous support. Last, but not least, we would like to thank the three referees for their extremely careful reviews (they did two!) and very thoughtful comments.

REFERENCES

ADLER, D., DAGEVILLE, B., AND WONG, K.-F. 1992. A C-based simulation package. Tech. Rep. ECRC-92-27i, ECRC, Munich.

AGRAWAL, D., ABBADI, A. E., AND JEFFERS, R. 1992. Using delayed commitment in locking protocols for real-time databases. In *ACM SIGMOD International Conference on Management of Data* (San Diego, Calif., June 2–5), 104–113. ACM, New York.

AGRAWAL, D., BRUNO, J. L., ABBADI, A. E., AND KRISHNASAWAMY, V. 1994. Relative serializability: An approach for relaxing the atomicity of transactions. In *ACM Principles of Database Systems*. V. Vianu, Ed., ACM, New York, 139–149.

AGRAWAL, H. 1994. On slicing programs with jump statements. *ACM SIGPLAN Not. 29*, 6 (June), 302–311.

AGRAWAL, R., CAREY, M., AND LIVNY, M. 1987. Concurrency control performance modeling: Alternatives and implications. *ACM. Trans. Database Syst. 12*, 4 (Dec), 609–654.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass.

BAYER, R. 1986. Consistency of transactions and random batch. *ACM Trans. Database Syst. 11*, 4 (Dec.), 397–404.

BERNSTEIN, P., SHIPMAN, D. W., AND ROTHNIE, J. B. 1980. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst. 5*, 1 (Mar.), 18–51.

BERNSTEIN, P A., HADZILACOS, V., AND GOODMAN, N. 1991 *Concurrency Control and Recovery in Database Systems* Addison-Wesley, Reading, Mass.

CAREY, M. J. 1983. Modeling and evaluation of database concurrency control algorithms Tech Rep. and Ph D. thesis, Dept. of Computer Science, Univ. of California, Berkeley, Sept.

CAREY, M. J., KRISHNAMURTHY, S., AND LIVNY, M. 1990. Load control for locking: The "half and half" approach. In *The 9th ACM Symposium on the Principles of Database Systems* (Nashville, Tenn , May 20–22) ACM, New York, 72–84.

CASANOVA, M. 1981. *The Concurrency Control Problem for Database Systems.* Lecture Notes in Computer Science, vol. 116. Springer-Verlag, New York.

DADAM, P., PISTOR, P., AND SCHEK, H.-J. 1983 A predicate oriented locking approach for integrated information systems. In *IFIP 9th World Computer Congress* (Paris, April). North-Holland, Amsterdam, 110–121.

FARRAG, A. A , AND OZSU, M. T 1987. Towards a general concurrency control algorithm for database systems. *IEEE Trans. Softw. Eng. SE-13*, 10 (Oct.), 1073–1078.

FARRAG, A. A., AND OZSU, M. T. 1989 Using semantic knowledge of transactions to increase concurrency *ACM Trans. Database Syst 14*, 4 (Dec.), 503–525

GARCIA-MOLINA, H. 1983. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst 8*, 2 (June), 186–213

GRAY, J , ED 1991. *The Benchmark Handbook* Morgan-Kaufmann, San Mateo, Calif.

GRAY, J., AND REUTER, A 1992. *Transaction Processing: Concepts and Techniques* Morgan-Kaufmann, San Mateo, Calif.

HARITSA, J., CAREY, M. J., AND LIVNY, M. 1990. On being optimistic about real-time constraints. In *The 9th ACM Symposium on Principles of Database Systems* (Nashville, Tenn., May 20–22). ACM, New York, 331–343

HELLAND, P , SAMMER, H., LYON, J., CARR, R., GARRET, P., AND REUTER, A. 1987. Group commit timers and high volume transaction systems. In *Second International Workshop on High Performance Transaction Systems.* Lecture Notes in Computer Science, Vol. 359, Springer Verlag, New York, 301–328.

HSEUH, W. AND PU, C. 1993. Chopping up epsilon transactions. Tech Rep. CUCS-037-093, Dept. Computer Science, Columbia Univ., New York

HSU, M., AND CHAN, A. 1986. Partitioned two-phase locking. *ACM Trans. Database Syst 11*, 4 (Dec.), 431–446.

LAUSEN, G., SOISALON-SOININEN, E., AND WIDMAYER, P 1986 Pre-analysis locking. *Inf Control 70*, 2–3 (Aug.), 193–215

LLIRBAT, F. 1994. Analysis of chopping algorithm's performance Intern Rep., Rodin Project, INRIA Rocquencourt, France.

LYNCH, N. 1983 Multi-level atomicity—A new correctness criterion for database concurrency control. *ACM Trans Database Systems 8*, 4 (Dec.), 484–502

O'NEIL, P. 1986. The escrow transactional mechanism. *ACM Trans. Database Syst. 11*, 4 (Dec), 405–430.

RAMAMRITHAM, K. AND CHRYSANTHIS, P K. 1995. *Advances in Concurrency Control and Transaction Processing* IEEE Press, to appear.

RYU, I. K., AND THOMASIAN, A. 1990. Analysis of performance with dynamic locking. *J. ACM 37*, 3 (Sept.), 491–523.

SALZBERG, B., AND TOMBROFF, D. 1994. A programming tool to support long-running activities. Tech. Rep. NU-CCS-94-10, Dept. Computer Science, Northeastern Univ., Boston, Mass.

SHASHA, D. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang Syst. 10*, 2 (Apr), 282–312.

SHASHA, D. 1992. *Database Tuning: A Principled Approach.* Prentice-Hall, Englewood Cliffs, N.J.

TAY, Y. 1987 *Locking Performance in Centralized Databases.* Academic Press, New York.

THOMASIAN, A. 1991. Performance limits of two-phase locking. In *7th IEEE International Conference Data Engineering* (Kobe, Japan). IEEE, New York, 426–435.

THOMASIAN A., AND RYU, K 1991. Performance analysis of two-phase locking. *IEEE Trans. Softw Eng 17*, 5 (May), 386–402.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. SE 10*, 4 (July), 352–357.
WOLSON, O. 1987. The virtues of locking by symbolic names. *J. Algorithms 8* (March) 536–556.
WONG, K. C., AND EDELBERG, M. 1977. Interval hierarchies and their application to predicate files. *ACM Trans. Database Syst. 2*, 3 (Sept.), 223–232.
YANNAKAKIS, M. 1982. A theory of safe locking policies in database systems. *J. ACM 29*, 3, (July), 718–740.

Received March 1994; revised January 1995; accepted June 1995