Transaction Chopping: Algorithms and Performances Studies

François Llirbat, Dennis Shasha, Eric Simon, Patrick Valduriez

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Transaction Chopping: Algorithms and Performances Studies

Dennis Shasha, Francois Llirbat, Eric Simon, Patrick Valduriez

N 2531

Avril 1995

PROGRAMME 1

ISSN 0249-6399

Transaction Chopping: Algorithms and Performances Studies

Dennis Shasha*, Francois Llirbat**, Eric Simon**, Patrick Valduriez**

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Rodin

Rapport de recherche n°2531 — Avril 1995 — 49 pages

Abstract: Chopping transactions into pieces is good for performance but may lead to non-serializable 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 (i) choosing between degree 2 and degree 3 isolation levels (degree 2 corresponds to level 1 in the new ANSI SQL terminology) (ii) the ability to execute a portion of a transaction using multiversion read consistency, and (iii) 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 online or real-time transactional applications).

*Courant Institute, New York University, shasha@SHASHA.CS.NYU.EDU
**{Francois.Llirbat}{Eric.Simon}{Patrick.Valduriez}@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Té lé phone : (33 1) 39 63 55 11 —Té lé copie : (33 1) 39 63 53 30

Given this information, our algorithm finds the finest partitioning of a set of transactions TranSet with the following property: *if the partitioned transactions execute serializably, then TranSet executes serializably.* This permits users to obtain more concurrency while preserving correctness. Besides obtaining more inter-transaction concurrency, chopping transactions in this way can enhance intra-transaction 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.*** ****

Key-words: Database Systems, Transactions Performance, Concurrency Control.

(Résumé : tsvp)

***This work was supported by U.S. Office of Naval Research #N00014-91-J-1472, and #N00014-92-J-1719, U.S. National Science Foundation grants #IRI-89-01699 and #CCR-9103953.

****A preliminary version of this paper appeared in the proceedings of the ACM SIGMOD International Conference held in San-Diego, May 1992 under the title "Simple Rational Guidance for Chopping Up Transactions."

Découpage de transactions : algorithmes et étude de performances

Résumé : Découper des transactions en transactions plus petites améliore les performances mais peut mener à des exécutions non sérialisables. Face à ce problème, la plupart des chercheurs ont exploré deux types de solutions : inventer un nouveau mécanisme de contrôle de concurrence ou relâcher la contrainte de sérialisation. Nous proposons une approche différente.
Nous supposons un utilisateur qui :

- dispose uniquement d'outils de niveau utilisateur tels que : (i) Choisir entre le degré 2 et le degré 3 d'isolation. (ii) Choisir d'exécuter une portion de transaction en multiversion. (iii) Changer l'ordre d'exécution des opérations de sa transaction.

- connait l'ensemble des transactions concurrentes qui peuvent s'exécuter pendant un intervalle donné. (Ceci est en général vrai pour les applications transactionnelles temps réel ou exécutant des transactions en tâches de fond.)

A partir de ces informations, notre algorithme trouve le découpage le plus fin de l'ensemble des transactions TranSet qui vérifie la propriété suivante : *Si les transactions découpées s'exécutent de façon sérialisable alors TranSet s'exécute de façon sérialisable.* Ceci permet à l'utilisateur d'améliorer la concurrence entre les transactions tout en garantissant une exécution correcte. De plus, cet algorithme permet d'améliorer le parallélisme entre les pièces provenant d'une même transaction découpée.

L'algorithme est peu coûteux. Sa complexité est de $O(n \times (e + m))$ une fois que les conflits sont identifiés et en utilisant une implémentation naive où e est le nombre d'arcs de conflit entre les transactions, n est le nombre de transactions concurrentes dans l'intervalle donné et m est le nombre maximal d'accés à la base de donnée pour la plus longue des transactions concurrentes. Aussi est-il envisageable d'utiliser cet algorithme sur des systèmes réels pour régler les performances transactionnelles.

Mots-clé : Bases de données, performances transactionnelles, contrôle de concurrence.

1 Motivation

The database research literature has many excellent proposals describing new concurrency control methods. The proposals (some of which are described in the Related Work section of this paper and others cited in the book[26]) aim to help database management system 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 degree 3 isolation (i.e. serializability) and less restrictive locking methods to enforce degree 2 isolation (i.e. write locks obey two phase locking; read locks are released immediately after read completes — this is how vendors implement ANSI level 1 serializability). 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 $purchase(i, p)$ — 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 became 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.

End of Example 1 – Inventory and Cash

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 won't 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 Mini-Batching: 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 *mini-batching* technique in industrial transaction processing [3]. For example, suppose that a batch transaction does a sequential scan and update of 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 "mini-batch" transactions each of which updates 1 million records and writes the number of millions updated. On recovery, the application reads the last 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. Mini-batching will be required any time 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]

End of Example 2

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 not propose a new concurrency control algorithm, because we assume the user is tuning on top of a purchased database management system (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 between degree 2 and degree 3 serializability. We consider this to be an important point about tuning research as contrasted with classical research in database internals: *The tuner cannot 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.*[5]

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 the transaction programs[2] that may run in some time interval.

[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.[4] This research entails changing the internal algorithms of the database management system.

[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 respectively) where it is not clear from context.

The characterization may be parametrized. 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 certain no two instances of the same transaction program will execute concurrently. The more such information, the merrier.

- The goal is to achieve the guarantees of serializability (degree 3 consistency) — without paying for it.

  That is, the user would like either to use degree 2 consistency, 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 its modifications. Thus, the user should rearrange the code so rollbacks occur early. We will formalize this intuition below 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 mini-batching technique illustrated in the Remark of example 2).

Suppose there are n transaction instances $T_1, T_2, \ldots, T_n$ that can execute within some interval. Let us 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 will reexamine this assumption in section 4.
All along this paper, we consider transaction programs with simple control structures: sequences, loops, and if-then-else statements. As defined in [6], 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' derive from the same program text, but s' may be associated with a later iteration of a looping construct (e.g. while loop 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 the rollback statements of T are in its first piece. Further, all 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 serializable concurrency control algorithm and will commit its changes when it ends. (Extensions of this work to non-serializable environments or to multidatabase settings may find it necessary to relax this assumption.)

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] For example, if the transaction instance updates account X first and branch balance B second, then the piece that updates account X should complete before the piece that updates branch balance B begins.

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 *non-commutativity*.

   If we don't 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. However, 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.[5]

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.

[5] 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 $1 <= i < k$ and an edge between n_k and n_1; no edge is included twice.

"Equivalence" is defined using the serialization graph formalism of [1] 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'. Conflicts should be understood in the sense of non-commutativity as described earlier in this section. Following [1], if the serialization graph resulting from an execution is acyclic, then the execution is equivalent to a serial one. Further the book proves 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.

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. We need this to avoid the trivial result that the execution is serializable by being equivalent to a null execution.

For point 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 $T_i \rightarrow T_j \rightarrow \ldots \rightarrow T_i$. Identify the pieces of the chopping associated with each transaction instance that are involved in this cycle: $p \rightarrow p' \rightarrow \ldots \rightarrow p''$. 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 point 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 parametrized 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)
```

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)
```

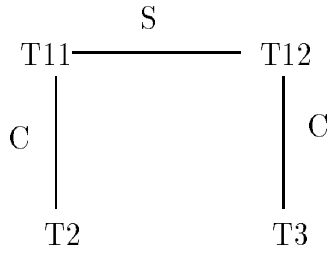

Figure 1:

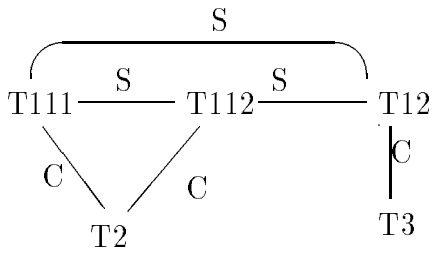

Figure 2:

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

Chopping Graph Example 3

Now, let us consider an example in which there are three types of transaction programs:

- A transaction program that updates a single depositor's account and the depositor's corresponding branch balance.
- A transaction program that reads a depositor's account balance.
- 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)
```

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)
```

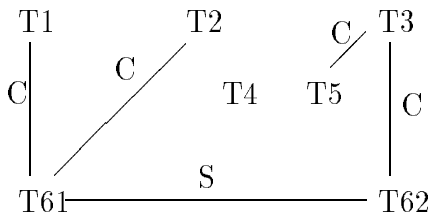

Figure 3:

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 5

Taking the transaction population from the previous example, let us now consider dividing T1 into two transactions giving the following transaction population (see Figure 4).

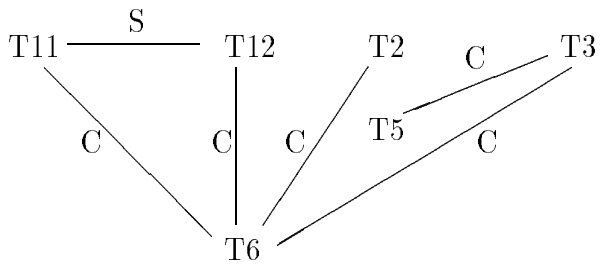

Figure 4:

```
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
```

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 particularly worrisome 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 sections of this chapter, we assume that the execution of a chopped transaction follows the initial ordering of operations defined in the corresponding transaction program. In the last section, 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 transaction program. The results will allow us to relax this overly restrictive assumption later.

Lemma 1: 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 and 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 ordering 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 semantic information, this assumption holds provided we never chop semantically non-conflicting 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 2: Suppose that in some chopping $chop_1$, two pieces, say p and p', of transaction instance T are in an SC-cycle and 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 by the Enclosing Conflict Assumption. 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, so will chopping T3 further.

These two lemmas lead directly to a systematic method for chopping transactions as finely as possible. Consider again the set of transaction instances $\{T_1, T_2, \ldots, T_n\}$

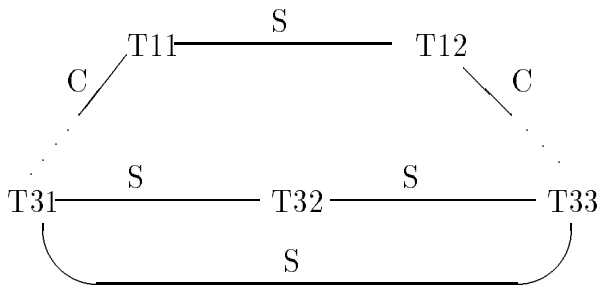

Figure 5:

that can run in this interval. 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
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 2: 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-safe: the chopping is rollback-safe because all its constituents are rollback-safe.
- No SC-cycles: if there were an SC-cycle that involved two pieces of private(T_i), then Lemma 2 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 2 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 $(T_1, ..., T_n)$
    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
        $\{Fine_1, Fine_2, ..., Fine_n\}$

We now give an algorithm to find the finest private chopping of T.

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[6]
  end
  $P := \{\{x\} \mid x$ is a primitive database access not in $p_1\} \cup \{p_1\}$;

  merging pieces of transaction T assuming $P = \{ p_1, \cdots, p_r \}$:

  Consider the graph having P and all unchopped transaction instances
  besides T as nodes and edges defined by C.

Construct the connected components of the graph induced by the C edges;

update P based on the following rule:
for each connected component $p_{e1}, p_{e2}, ..., p_{ek}$, if the k pieces
of P are such that $e_1 < e_2 < ... < e_k < r$, then
put all accesses of $p_{e1}, p_{e2}, ..., p_{ek}$ into p_{e1}
and then remove $p_{e2}, ..., p_{ek}$.

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 $\{\{a\}, \{b, d, f\}, \{c\}, \{e\}\}$. If $\{b, d, f\}$ were subdivided further, there would be an SC-cycle in the chopping graph.

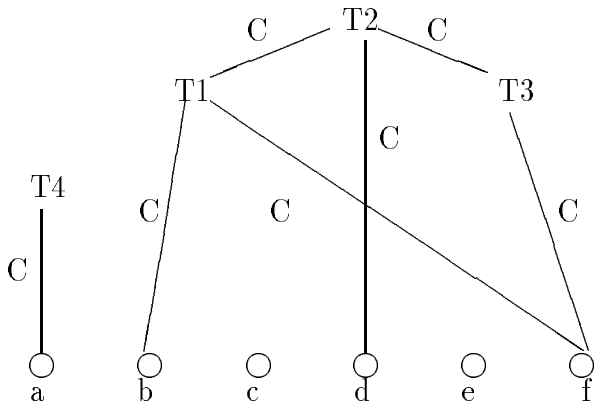

Figure 6:

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 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 the FineChop algorithm 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 two iterations may be in the same or different pieces. However in many cases, either all 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 will be chopped 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.

Consider now the Purchase transaction program of Example 1 - Inventory and Cash. Recall that this transaction is:

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

The initialization step of 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).

Suppose that we have another purchase transaction T. Then the graph induced by the C edges is shown on Figure 7. Pieces p_1 and p_2 will be merged. Observe that

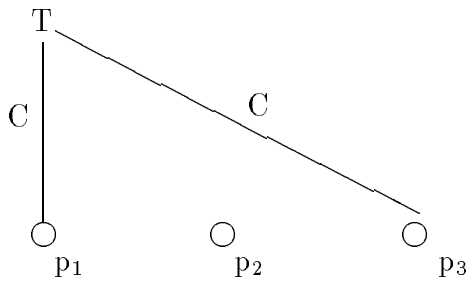

Figure 7:

this chopping does not depend on the number of concurrent purchase transactions considered while running the FineChop algorithm.

Theorem 3: 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 partition. 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 to 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 non-empty 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 order Dependency Graph for 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 the FineChop algorithm? In particular, if a piece does not include consecutive database operations, the program may have to be re-written or pieces will have to be merged.
For example, consider the Purchase transaction program.

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

In the previous section, we showed that the 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:

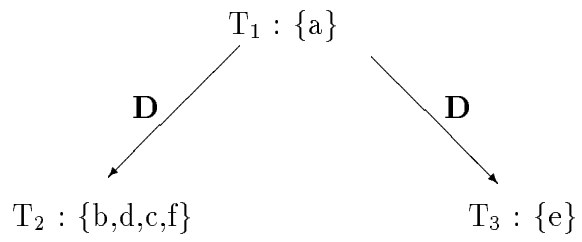

Figure 8: Example of Dependency Graph

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 accomplish this. 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 which "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 inter-transaction 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, ... 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

[7] The Precedence Graph represents the precedence relationship defined in Section 2

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 (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 same loop, all the iterations must be merged with this piece.

We summarize these considerations into the following algorithm:

Algorithm Preserve_Order (P : finest chopping of a transaction instance T)
Construct the dependency graph (P, D_edges);
Reduce each cycle in the dependency graph yielding a new, smaller set of superpieces P'
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 in [6], [7], [8]. 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 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 having D_edge arcs pointing to s complete. Thus, chopping can enhance intra-transaction 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 "mini-batch 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 in [9, 10]. 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 that 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 only 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. 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).

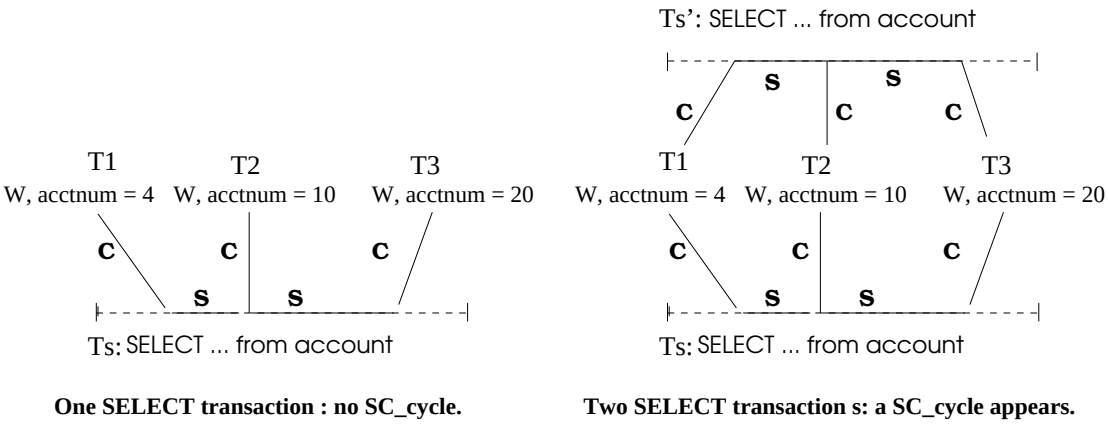

Figure 9: Applying Chopping to execute SELECT transactions at degree 2 isolation

Remark about Multiversion Read Consistency:

Several database management systems (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 non-serializable executions such as T1 begin R1(x) T2 begin W2(x) W2(y) T2 commit W1(y) T1 commit.

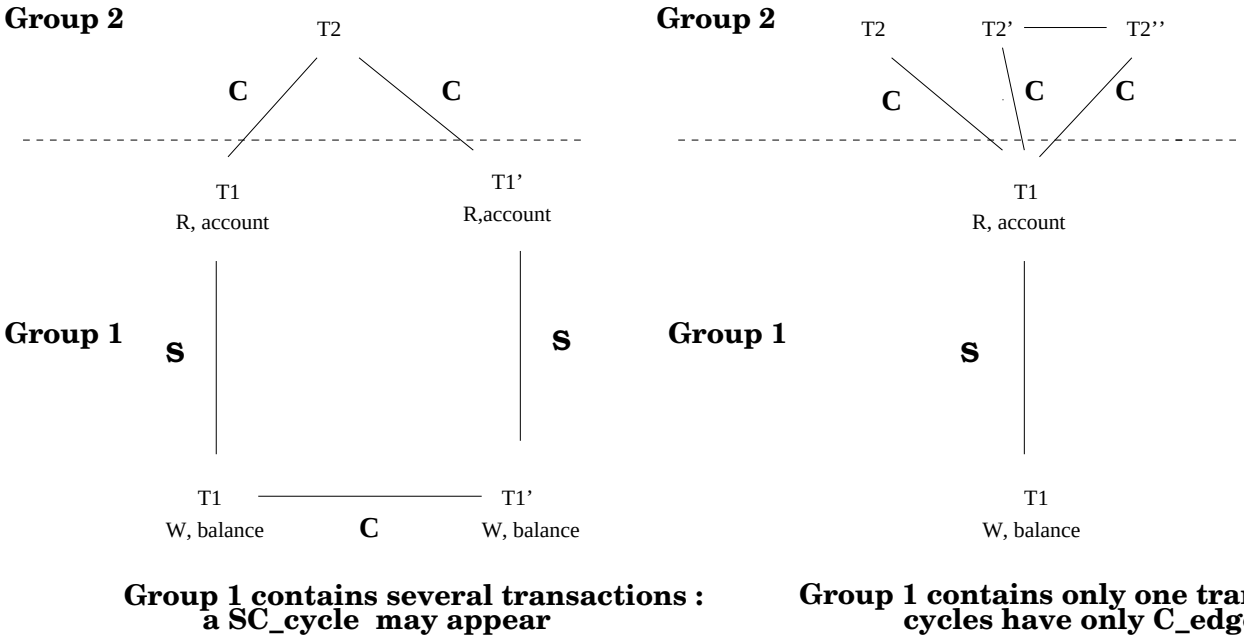

Figure 10: Applying Chopping to Multiversion Read Consistency — the SELECT statements using multiversion read consistency constitute one piece of the transaction instances

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, 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 the Group 1 selects 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^3AP[2]) benchmark using a commercial database management system (ORACLE [8] version 6). This section describes our SQL implementation, our experimental results, and discusses the performance tradeoffs 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

[8] ORACLE is a trademark of Oracle Corporation

which randomly updates a single tuple. Our goal is to quantify the value of chopping the long transaction while maintaining serializability. Our hypothesis was 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 following query :

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

and unmod_550_seq is :

```
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.
- The short non-conflicting 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 :

```
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 :

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

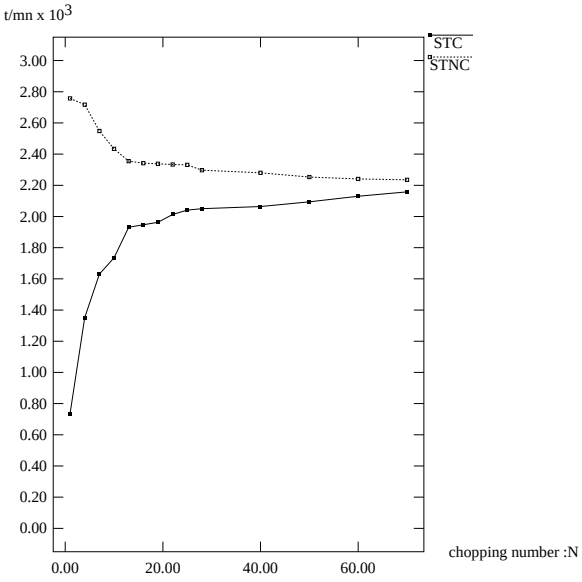

Figure 11: Throughputs of STC and STNC

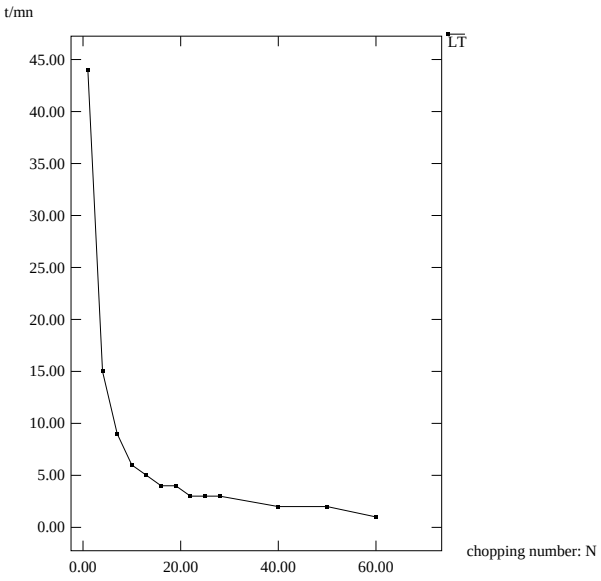

Figure 12: Throughput of LT

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:

- the number of short conflicting transaction instances, denoted nb_STC;

- the number of short non-conflicting transaction instances, denoted nb_STNC; and
- 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 Fine-Chop 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 process which exercises its transaction type in a loop. The execution continues for a predefined time T, yielding a transaction throughput for the long and short transactions.

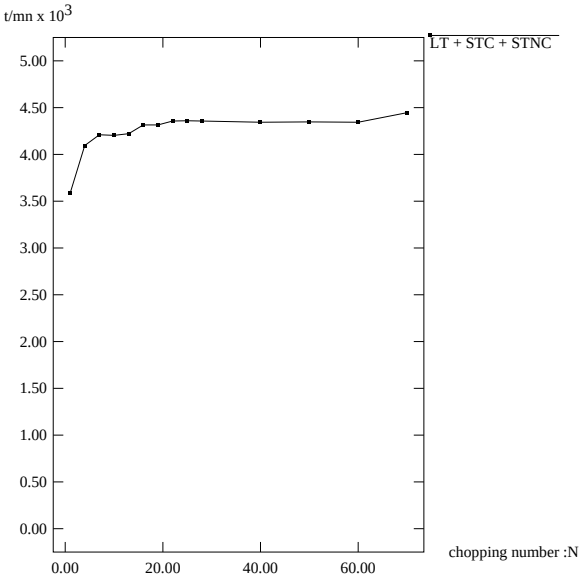

Figure 13: global throughput

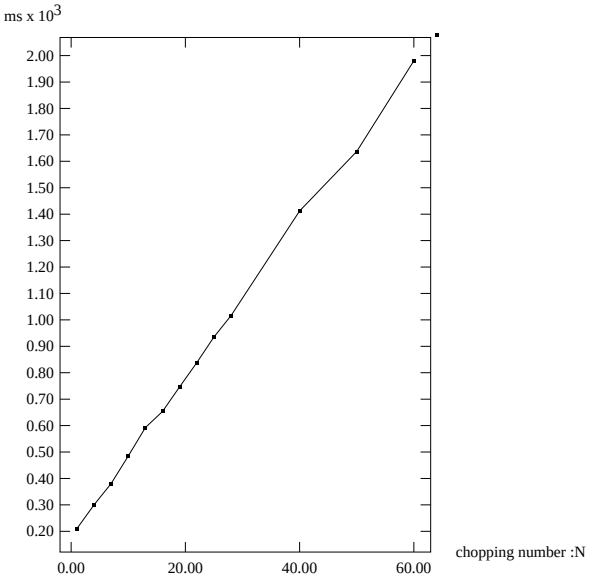

Figure 14: commit costs

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 is shown in figures 11 and 12. and 13.

Figure 11 shows that chopping the long transaction program (LT) significantly increases the throughput of short conflicting (STC) transactions. In fact, the throughput of STC transactions comes close to the throughput of non-conflicting transactions (STNC). That is the good news. The bad news is that the number of long transactions 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 conflicting short transaction instances for two reasons: (i)The maximum number of locks held at one time by the LT decreases. Consequently, the probability that a STC requests a lock held by a LT decreases. (ii) 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): (i) it entails more log writes because each piece commits separately (though group commits would mitigate this cost if available); (ii) Each log write causes a context switch.

Furthermore, since STC, STNC and LT transactions are sharing the same resources (CPUs and Disks), the increasing number of active STC transactions in the system affects the performances of STNC and LT transactions.

What is the bottom line? The short transactions have been made faster at some cost to the longer transactions. This is a reasonable tradeoff 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 (25% 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 has proposed a average case performance model for chopping [24]. Here, we briefly describe that model and show how it applies to this example.

As in Tay's work [23], 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 write-ahead logging [3].

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 which 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 developped and used a simulation model. We validated the simulation model by applying it to the Oracle experiments of the last section. The simulation model accurately reflected the relative benefits and costs of chopping in those experiments, though not the exact numbers.
The simulation results yielded 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 delays the thrashing of performance due to lock contention.

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

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

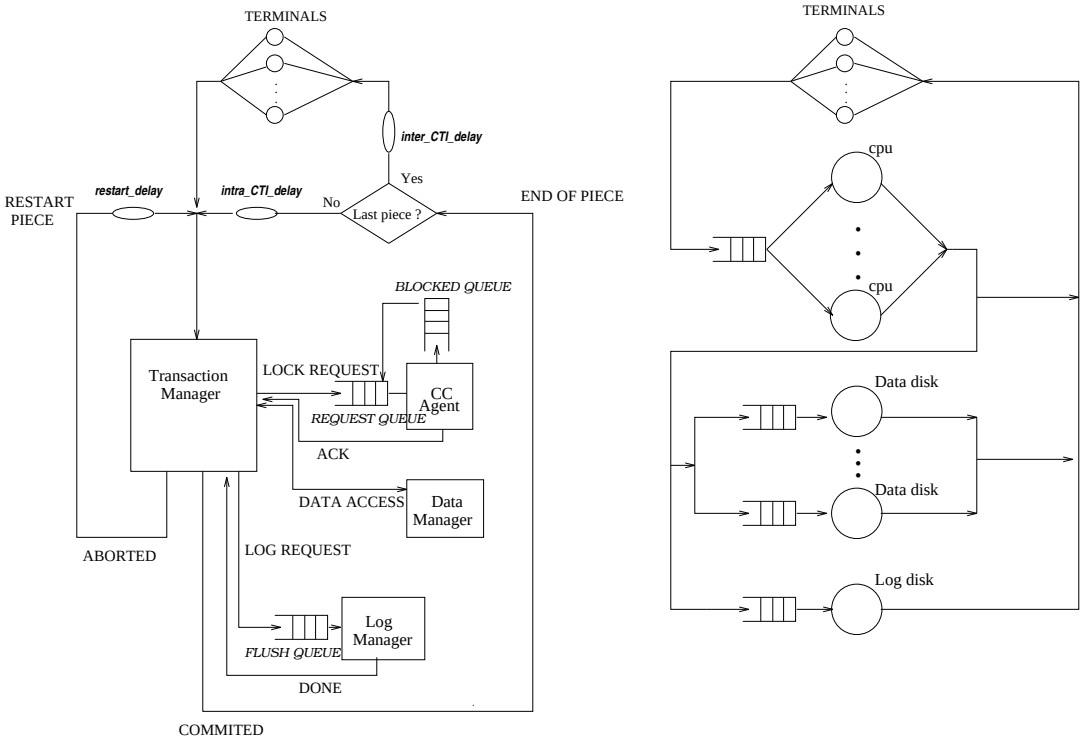

Figure 15: The Simulation Model

7.1 The Simulation Model

The database simulation model is based on [27], [28], [29], [30]. The main difference is the simulation of the log manager operations at commit time in order to take into account the cost of commits. This model was developed using the SIM package [31]. It is divided into four main components : a Transaction Manager (TM), a Concurrency Control Agent (CCA), a Data Manager (DM) and a 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

Parameter	Description	Value
db_size	Number of objects in database	20000 objects
num_terms	Number of terminals	1 to 100 terminal
N	Chopping number	1 to 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 to 4
obj_cpu	CPU time for executing an operation	1 millisecond
page_io_access	IO time for accessing a page	7 milliseconds
io_prob	Probabilty of IO page access	0.2
Log_disk_io	time for issuing a IO log access	7 milliseconds
Log_rec_io_w	IO time for writing 1 page on log disk	0.1 milliseconds
commit_cpu	cpu time for executing a commmit	2 milliseconds
abort_cpu	cpu time for executing an abort	2 milliseconds
inter_CTI_delay	time between two CTIs	10 milliseconds
intra_CTI_delay	time between two CTI's pieces	5 milliseconds
restart_delay	restart delay of one CTI's piece	5 milliseconds

Table 1: Simulation Model Parameter Definitions and Values

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 LM provides read and insert-flush interfaces to the log table. The DM is responsible for granting access to the physical data objects and executing the database operations.

Figure 15(a) represents a closed querying model of the execution of choppped transactions on a single-site database. Choppped Transaction Instances (CTI) 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 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

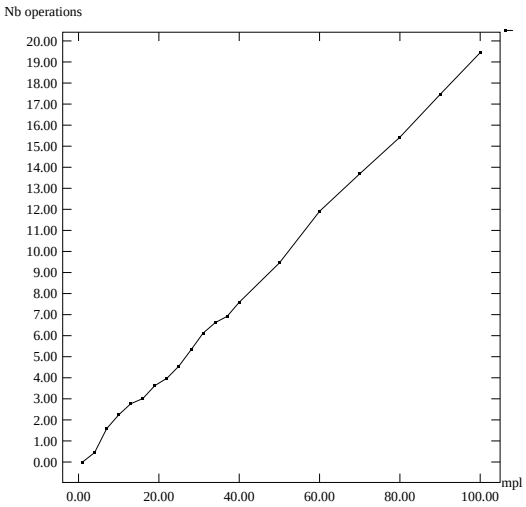

Figure 16: conflicts

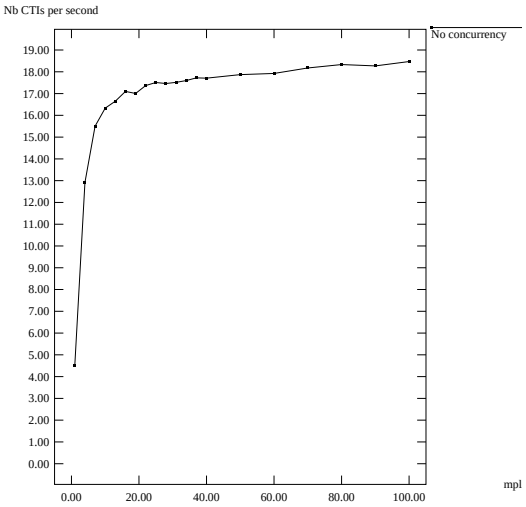

Figure 17: effects of resource sharing

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 wait-for graph. Whenever a deadlock is detected the youngest transaction is aborted. When a lock request can be granted, an aknowledgement 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[9] 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. [10]

The physical queuing model, shown in Figure 15(b) is quite similar to the one used in [27],[28],[30] 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

[9] We implemented a zero-wait timer group commit [32].

[10] Reading the log file usually requires no IO (see [3] p505).

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 which 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 consisted of a number of repetitions. A repetition is performed in three stages : (i) the generation of random transaction programs, (ii) the decomposition of these programs in pieces using the chopping algorithm, (iii) the simulation of the execution of the obtained CTI. The simulation time was fixed at 1000 seconds. The number of repetitions was chosen in order to achieve 90 percent confidence intervals for our results (30 repetitions). Table 1 summarizes the parameters and theirs 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 (16) shows that when we increase the multiprogramming level (*mpl*) then the number of conflicting operations per transaction [11] increases. In the curve (17) we measure the throughput of concurrent transactions running without concurrency control. 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. The graphs 18 and 19 illustrate respectively the throughput and the mean response time of the concurrent transactions. They show that chopping transactions improves 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.

[11] A data operation is conflicting if there exists a concurrent transaction which accesses the same object.

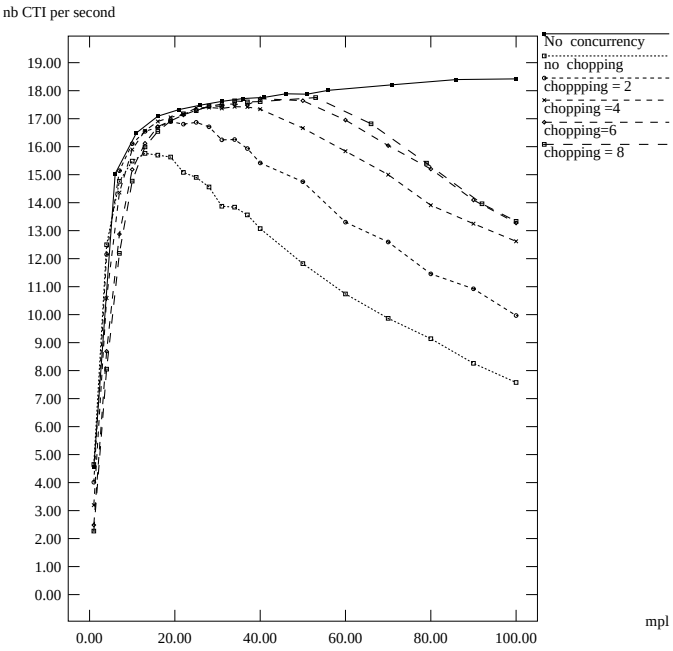

Figure 18: CTIs throughput

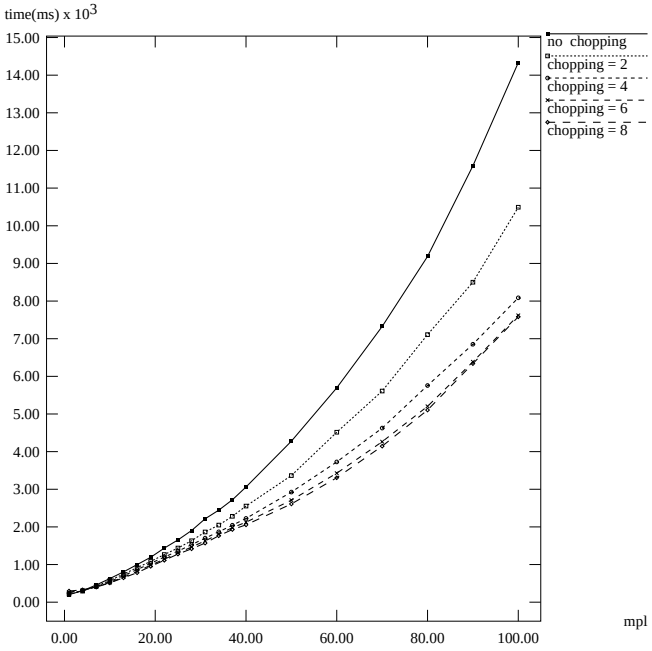

Figure 19: CTI's mean response time

The best improvements of performance (102% with $mpl = 100$) are provided by chopping with large multiprogramming levels when the throughput of the non-chopped transactions is in thrashing situation. Furthermore the throughput graph 18 shows that chopping transactions delays the thrashing point to larger multiprogramming levels. Specifically, the thrashing situation appears a $mpl = 20$ 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 [23],[33],[34],[35],[36] the thrashing behaviour is caused to a large extent by system under-utilization due to transation blocking and to a more limited extent by wasted processing caused by transaction aborts. In the graph (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 [12]. This effect is reduced by chopping. We also observed that chopping

[12] As explained in [34] , the strong increase of the mean waiting time is essentially due to cascading waits.

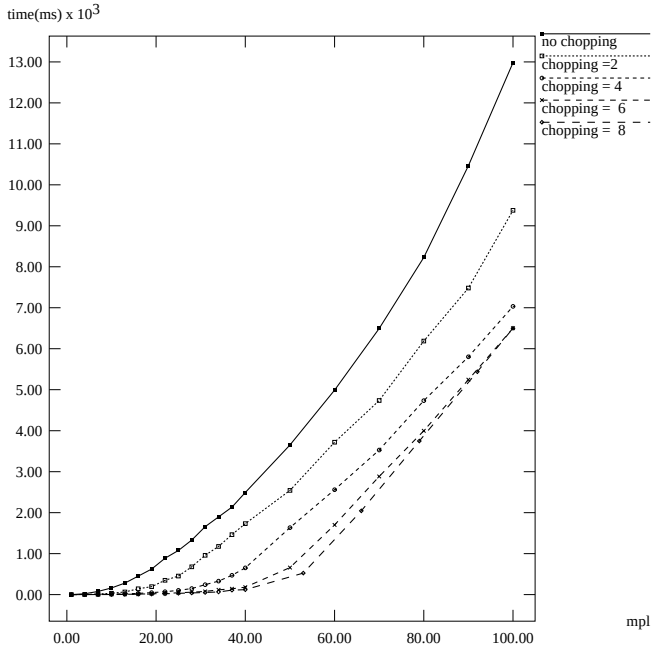

Figure 20: mean waiting time

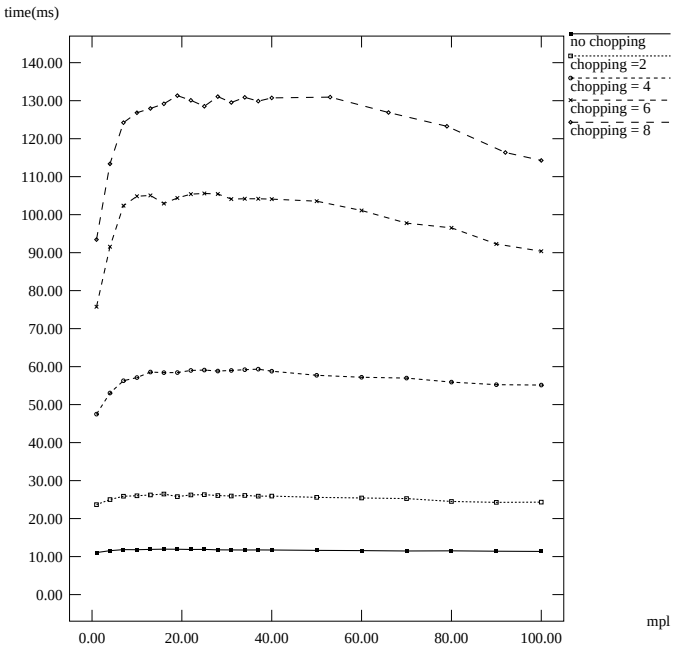

Figure 21: mean commit time

reduces the wasted processing caused by aborts by reducing the size of pieces to restart (30% reducing with $mpl = 100$).

On the other hand, chopping doesn't help much when the multiprogramming level is low. There are two reasons for this : (i) The probability that a transaction is involved in a conflict is small, eliminating the need to chop. (ii) 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. It 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[13]) increased from two to four . The 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

[13] see Table 1.

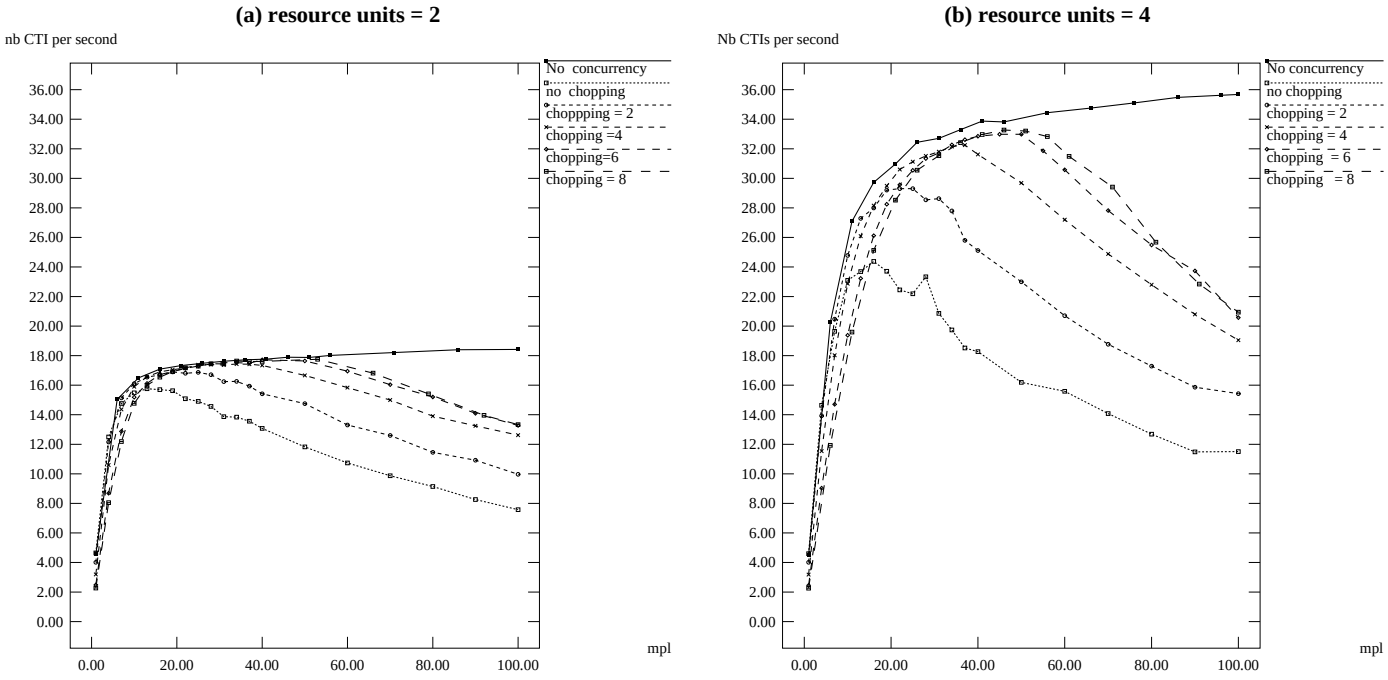

Figure 22: effects of resources

by chopping the CTIs in eight pieces represents a 39% improvement compared to the maximal throughput for $k = 4$ (see the graph 22b) obtained when there is no chopping. The maximal throughput for $k = 2$ (see the graph 22a) obtained by chopping into eight pieces represents only a 12% 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 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 ana-

lysis in the previous section allow us to point out three kinds of situations where increasing the chopping number is useless :

- There is resource contention. Once again, if resources are saturated, increasing the number of active transactions will not improve the global throughput. (Compare the graphs 22a and 22b)
- There is no lock contention. In these situations the marginal gain of chopping is too small because the probability of conflit is small. (see the graph 18 when mpl is lower than 15).
- The chopping algorithm chops only pieces involved in few conflicts. For example, according to the throughput graphs 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.
3. If the system is under-utilized because of lock contention then :
   - Find the pieces which are involved in a lot of conflicts.
   - 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 non-conflicting 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 database management system 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 contributions.

Farrag and Ozsu[11] consider 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 observe 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:

```
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 hotel reservation 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 note in conclusion that finding semantically acceptable interleavings is very hard. It is possible that chopping would make it easier.

Hector Garcia-Molina[12] suggested using semantics by partitioning transactions into classes. Transactions in the same class can run concurrently, whereas transactions in different classes must synchronize. He proposes 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.

Nancy Lynch[13] 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).

Rudolf Bayer[14] 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.

Meichun Hsu and Arvola Chan[15] have examined special concurrency control algorithms for situations in which data is 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 that data should be able to proceed without being blocked by reads of that data. That approach does not guarantee serializability, so chopping would not help.

Patrick O'Neil[16] takes 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.

Ouri Wolfson[17] presents an algorithm for releasing certain locks early without violating serializability based on an earlier theoretical condition given by Yannakakis[18]. He assumes that the user has complete control over the acquisition and release of locks. Using different formalism and different algorithms, Lausen, Soisalon-Soininen and Widmayer [22] also examine chopping algorithms under the assumption of complete control over locks. The setting here is a special case of those algorithmic approaches: the user can control only 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, Shipman and Rothnie[19] introduced the idea of conflict graphs in an experimental system called SDD-1 in the late 1970's. Their system divided transactions into classes such that transactions within a class executed serially whereas transactions between classes could execute without any synchronization.

Marco Casanova's thesis[20] 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[21] 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 are 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[25].

9 Conclusion

We propose 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 serializably, then the transaction instances will appear to execute serializably.

This permits database users to obtain more concurrency and intra-transaction 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.

We sketch the application of our algorithm to SQL-based systems. Our experiments on a real system using the AS^3AP benchmark showed that chopping improves performance by reducing lock contention, though it increased resource contention. These observations were confirmed in more general situations by using a simulation model. Moreover, the simulation results showed that chopping transactions can be a good method to prevent thrashing due to lock contention.

We note that read transactions show no tradeoff — using degree 2 isolation (i.e. ANSI level 1 isolation) always yields better performance. (This may be why using chopping to show that level 1 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? Ditto for workflow applications?
- 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 started on combining chopping with epsilon serializability [25].
- 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 database management system?
- Finally, if we could change the underlying database management system, then we could ask the transaction load controller to detect the formation of the SC-graph dynamically. This presents promising opportunities for optimization.

10 Acknowledgments

We would like to thank Gerhard Weikum for his astute comments regarding order-preserving serializability and intra-transaction 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 its effective contribution to the implementation of these tests, and to all 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 and helpful comments.

References

[1] "Concurrency Control and Recovery in Database Systems" P. A. Bernstein, V. Hadzilacos, and N. Goodman. Reading, Mass.: Addison-Wesley, 1987.

[2] Jim Gray Ed., *The Benchmark Handbook.* San Mateo, Calif.: Morgan-Kaufmann,1991.

[3] "Transaction Processing: Concepts and Techniques" J. Gray and A. Reuter. San Mateo, Calif.: Morgan-Kaufmann, 1992.

[4] "A Programming Tool to Support Long-Running Activities" B. Salzberg and D. Tombroff. Technical Report NU-CCS-94-10, Northeastern University, Boston, 1994.

[5] "Database Tuning: a principled approach" D. Shasha. Englewood Cliffs, New Jersy: Prentice-Hall, 1992.

[6] "Compilers: Principles, Techniques and Tools", chapter 10, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Addison-Wesley Publishing Company, 1986.

[7] "Program Slicing" M. Weiser. IEEE Transaction on software Engineering, july 1984

[8] "On Slicing Programs with Jump Statements" H. Agrawal. ACM Transactions on programming Languages and systems, 1994

[9] "Interval Hierarchies and their Application to Predicate Files" K. C. Wong and M. Edelberg. ACM transactions on Database Systems, September 1977, vol. 2, no. 3, pp. 223-232

[10] "A Predicate Oriented Locking Approach for Integrated Information Systems," P. Dadam, P. Pistor, and H-J. Schek. IFIP Congress, Paris, 1983, published by North-Holland, 1983.

[11] "Using Semantic Knowledge of Transactions to Increase Concurrency" Abdel Aziz Farrag and M. Tamer Ozsu. ACM transactions on Database Systems, December 1989, vol. 14, no. 4, pp. 503-525

[12] "Using Semantic Knowledge for Transaction Processing in a Distributed Database" Hector Garcia-Molina. ACM Transactions on Database System, June. 1983, vol. 8, no. 2, pp. 186-213.

[13] "Multi-level Atomicity — a new correctness criterion for database concurrency control" Nancy Lynch. ACM Transactions on Database System, Dec. 1983, vol. 8, no. 4, pp. 484-502.

[14] "Consistency of Transactions and Random Batch" R. Bayer. ACM Transactions on Database Systems, December 1986, vo. 11, no. 4, pp. 397-404

[15] "Partitioned Two-Phase Locking" M. Hsu and A. Chan. ACM Transactions on Database Systems, December 1986, vo. 11, no. 4, pp. 431-446

[16] "The Escrow Transactional Mechanism" Patrick O'Neil. ACM Transactions on Database Systems, December 1986, vo. 11, no. 4, pp. 405-430

[17] "The Virtues of Locking by Symbolic Names" Ouri Wolfson. Journal of Algorithms 1987 8, pp. 536-556, 1987

[18] "A Theory of Safe Locking Policies in Database Systems," Mihalis Yannakakis. JACM 29(3), pp. 718-740, (1982).

[19] "Concurrency Control in a System for Distributed Databases (SDD-1)" P. A. Bernstein, D. W. Shipman and J. B. Rothnie. ACM Transactions on Database Systems, March 1980, vol. 5, no. 1, pp. 18-51.

[20] "The Concurrency Control Problem for Database Systems" Marco Casanova. Springer-Verlag Lecture Notes in Computer Science no. 116, 1981

[21] "Efficient and Correct Execution of Parallel Programs that Share Memory" D. Shasha and M. Snir. ACM Transactions on Programming Languages and Systems, vol. 10, no. 2, pp. 282-312, April, 1988

[22] "Pre-analysis Locking" G. Lausen, E. Soisalon-Soininen and P. Widmayer. Information and Control, vo. 70, no. 2/3, August/September 1986 pp. 193-215

[23] "Locking Performance in Centralized Databases" Y.C TAY. Academic Press, Inc, 1987.

[24] "Analysis of Chopping Algorithm's Performance " F. LLIRBAT. Internal Report, Rodin Project, INRIA .

[25] "Chopping Up Epsilon Transactions" Hseuh, Wenwey and Pu, Calton. Technical Report : CUCS-037-093 Columbia University, New York.

[26] "Advances in Concurrency Control and Transaction Processing" Krithi Ramamritham and Panos K. Chrysanthis, eds. IEEE Press, to appear

[27] "Modeling and Evaluation of Database Concurrency Control Algorithms" M. J. Carey. PhD thesis, University of California, Berkeley, September 83.

[28] "Concurrency Control Performance Modeling : Alternatives and Implications" R. Agrawal, M.J. Carey and M. Livny. ACM Transactions on database systems.

[29] "On Being Optimistic about Real-Time Constraints" J. Haritsa, M. J. Carey and M. Livny. Proceedings of the Ninth ACM Symposium on Principles of Database Systems, pages 331-343, april 1990.

[30] "Using Delayed Commitment in Locking Protocols for Real-Time Databases" D. Agrawal, A El. Abbadi and R. Jeffers. Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data.

[31] "SIM : A C-based SIMulation Package" D. Adler, B. Dageville and Kam-Fai Wong. ECRC-92-27i.

[32] "Group Commit Timers and High Volume Transaction Systems" P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garret, and A. Reuter Second International Workshop on High Performance Transaction Systems, Asimolar, September 1987

[33] "Performance Analysis of Two-Phase Locking" Alexander Thomasian and Kyung Ryu. IEEE Transactions on Software Engineering, Vol.17, No. 5,May 1991.

[34] "Performance limits of Two-Phase Locking" Alexander Thomasian. 7 th IEEE Int. Conf. Data Eng., Kobe, Japan, Apr. 1991.

[35] "Analysis of Performance with Dynamic Locking" In Kyung Ryu and Alexander Thomasian. Journal of the Association for Computing Machinery, Vol.3, July 1990,pp.491-523.

[36] "Load Control for Locking : the 'half and half' approach" M. J. Carey , S. Krishnamurthy and M. Livny. Proc 9th ACM Symp. on the Principles of Database Systems, Nashville, April 90, pp 72-84.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399