A Practical Technique for Asynchronous Transaction Processing

Wenwey Hseush
Department of Computer Science
Columbia University
New York, NY 10027

Calton Pu
Department of Computer Science and Engineering
Oregon Graduate Institute
Portland, OR 97291-1000

Abstract

Asynchronous Transaction Processing extends traditional on-line transaction processing (TP) to improve performance of distributed systems by alleviating the serializability (SR) bottleneck. For example, epsilon serializability (ESR) uses divergence control algorithms to allow more concurrency by permitting limited non-SR interleavings. In a distributed environment, ESR relaxes commit and abort dependencies among transactions, allowing transactions to commit asynchronously. A second example, chopping up transactions allows more concurrency by dividing transactions into smaller pieces and thus reduces resource holding time. Chopping transactions enforces no commit protocols among pieces from one original transaction, allowing each piece to commit asynchronously. We combine the benefits of ESR and chopping transactions by designing three new methods that chop transactions and run them under ESR. The practical applicability of our technique is enhanced by two factors: (1) chopping transactions does not require changes in existing TP systems, and (2) ESR support has already been prototyped on a commercial TP system.

1 Introduction and Background

On-line transaction processing (OLTP) [2, 5] has been considered as an important technique for reliable computing in distributed systems. OLTP defines a transaction to be an atomic computation unit in the presence of possible system failures and concurrent executions. Serializability (SR) is the standard correctness criterion for atomic transaction behavior. However, guaranteeing global serializability for distributed transactions does not come free. Transaction executions must be synchronized for failure recovery and isolation with respect to concurrent access. For failure recovery, a commit protocol guarantees that either all sub-transactions of a distributed transaction commit or none of them commit. For isolation, since the union of locally serializable sub-transactions do not always make a globally serializable distributed transaction, a global validation process must ensure that all sites have the same serialization order. As a result, the synchronous nature of transaction processing limits distributed system performance as well as availability.

We use *data contention* as a general term to refer to the problems imposed by serializability. Asynchronous transaction processing addresses the data contention problems in distributed systems. Two distinct areas of work are under development. The first one is to design practical optimization techniques while preserving serializability as the correctness criterion. A good example is chopping up serializable transactions into shorter transactions by Shasha et al [11]. The second way to decrease data contention is to relax the restrictions of serializability. A good example is *epsilon serializability* (ESR) [10, 12], which allows a limited amount of inconsistency to be introduced to a transaction. These two areas have been perceived as alternatives, perhaps because many of the SR-relaxing techniques could not preserve compatibility with SR. Consequently, despite the wealth of proposals in both areas, little is known about the interaction between promising techniques from distinct areas when implemented in the same system.

Our main contribution is a concrete demonstration that carefully chosen asynchronous transaction processing techniques, e.g., ESR and transaction chopping, even though from apparently alternative areas of research, can be combined to further improve transaction processing system performance. This observation is important since it introduces a new dimension in the evaluation of asynchronous transaction processing techniques: how well the new method can be combined with other asynchronous transaction processing techniques. Our second contribution is in the technical solution of two problems: (1) adapting Shasha's chopping algorithm (referred to as SR-chopping in this paper) to work with a new chopping strategy under ESR (referred to as ESR-chopping), and (2) distribution of potential inconsistency, both actual and bound specifications, in distributed systems. Our third contribution is the design of three methods to combine ESR and transaction chopping, giving the application designer a good number of choices. We emphasize that our methods have practical importance for two reasons. First, chopping is an off-line method that can be used without changing the existing TP system (or DBMS systems). Second, ESR support has been prototyped on a production TP monitor, namely, Transarc Encina. The rest of this paper is organized as follows. Section 1.1 and Section 1.2 compare the main results of this paper to related work. Section 2 describes three new methods. It describes how to run SR-Chopping under divergence control (method 1) and the enhancement of chopping algorithms under ESR. It explains how the ESR-chopped transactions run under concurrency control (method 2) and divergence control (method 3). Section 4 describes how the three methods can be applied to distributed transaction processing. Section 5 discusses the costs and benefits of the three methods.

1063-6927/95 $4.00 © 1995 IEEE

1.1 Epsilon Serializability

Epsilon Serializability (ESR) is a generalization of classic serializability (SR). ESR is defined for database state spaces that have a distance measure (*e.g.*, integers and real numbers — metric spaces in general). This assumption is less stringent than it appears at first sight. For many database attributes that lack an obvious distance function, e.g., text strings, there are candidate distance functions that work for many applications. In the strings example, we can reduce the strings to sets and use the usual set comparison function (i.e., the distance between two sets is defined by the number of differing members in those sets).

ESR allows a limited amount of inconsistency in transaction processing. An *epsilon transaction* (ET) is a sequence of operations that maintains database consistency when executed atomically. However, an ET extends the standard notion of an atomic transaction in the sense that an ET includes a specification of the amount of permitted inconsistency, or *fuzziness*, called the ϵ-spec. For example, a query-only ET is allowed to view inconsistent data due to non-serializable interleavings of operations with concurrent update ETs. Consider an update ET that transfers money from a saving account to a checking account and a query-only ET that reads the balances of all checking and saving accounts to calculate the sum of accounts. Under serializability, it is illegal for the query-only ET to execute all read operations between the two write operations of the update ET. Under ESR, this execution may be legal. Assume that the query-only ET is issued by a bank manager who is responsible for investment planning and interested in knowing the amount in ten thousands. He can assign the amount of permitted inconsistency to be $10,000.00. For the above non-serializable execution, if the result (the sum of the account balances) is within $10,000.00 of a serializable result, the execution is legal under ESR. Such non-serializable interleavings can increase transaction processing system performance through added concurrency. With ESR, application programmers can specify the amount of inconsistency allowed for each ET. In this paper, we focus on the environments where query-only ETs may access inconsistent data through non-serializable interleavings with other update ETs but update ETs are serializable among themselves.

The bounded inconsistency in ESR is automatically maintained by *divergence control* (DC) algorithms similar to a way in which serializability is enforced by concurrency control (CC) algorithms in classic transaction processing systems. The function of divergence control algorithms is to guarantee that the result of any ET is within ϵ-spec from a serializable database state. Various divergence control algorithms for both *centralized* and *distributed* transaction processing systems have been described in [12, 8]. We briefly describe the two-phase locking divergence control, which is similar to the two-phase locking concurrency control except for the way they handle read-write conflicts. For accounting purposes, an import inconsistency limit is specified for each query ET and an export inconsistency limit is specified for each update ET. When a read-write conflict occurs between an update ET and a query ET, the query ET is said to import fuzziness and the update ET is said to export fuzziness. The query ET accumulates *importedfuzziness* and is allowed to proceed if the accumulated fuzziness is within its import inconsistency limit. Otherwise, the query ET is blocked as it is handled in the two-phase locking concurrency control. On the other hand, the update ET accumulates *exported fuzziness* and is allowed to proceed if the accumulated fuzziness is within its export inconsistency limit.

1.2 Chopping Up Transactions

Dennis Shasha et al. proposed an off-line approach [11] to improve performance in transaction processing by chopping up serializable transactions into pieces and allowing each of them to run as an individual transaction. This approach requires no change to the transaction system (*i.e.*, use the same concurrency control algorithms and commit protocols). It simply asks database users to restructure transactions (off-line) according to the guidelines provided by the chopping technique. The idea is to shorten transactions to reduce concurrent contention imposed by serializability. This approach takes advantage of two assumptions: (1) Database users (an administrator or a sophisticated application developer) have knowledge of all the transactions that will run during some time interval; (2) Database users have knowledge of all rollback statements in every transaction.

Given a set of transactions, $T = \{t_1, t_2, ..., t_n\}$, a *chopping* partitions each transaction t_i into a set of small transactions (referred to as *pieces*), $\{p_1,\ p_2,\ ...,\ p_k\}$, where the first piece p_1 must commit before other pieces can commit. Other pieces can execute concurrently as long as they obey the dependency orders imposed by the transaction program text. A chopping is *rollback-safe* if either t_i has no rollback statements or all rollback statements of t_i are in p_1. That is, if p_1 commits, then all other pieces must eventually commit. When a piece other than p_1 is aborted due to a lock conflict, it will be resubmitted repeatedly until it commits. However, when p_1 is aborted due to a rollback statement, all other pieces will not execute. In this paper, we use $\mathcal{CHOP}(t)$ to denote a partition of transaction t, $t \in T$.

$$\mathcal{CHOP}(t) = \{p_1,\ p_2,\ ...,\ p_k\}$$

We use $\mathcal{CHOP}(T)$ to denote a partition of transaction set T.

$$\mathcal{CHOP}(T) = \bigcup_{t \in T} \mathcal{CHOP}(t)$$

A chopping of T is represented by a *chopping graph*, where the vertex set is $\mathcal{CHOP}(T)$ and the edge set is a set of C edges and S edges.

- C edges - C stands for *conflict*. Two pieces p and q from different transactions ($p \in \mathcal{CHOP}(t_i)$, $q \in \mathcal{CHOP}(t_j)$ and $t_i \neq t_j$) conflict, if an operation a from p_i and an operation b from p_j conflict (i.e., a and b do not commute). A C edge is an edge, $(p,\ q)$, where p and q conflict.
- S edges - S stands for *sibling*. Two pieces p and q are siblings if they come from the same transaction t ($p \in \mathcal{CHOP}(t)$ and $q \in \mathcal{CHOP}(t)$). An S edge is an edge, $(p,\ q)$, where p and q are siblings.

An SC-cycle is a simple cycle that includes at least one S edge and at least one C edge. A C-cycle is a simple cycle formed by C edges only. A chopping is *correct* if any execution of the chopping (i.e., treating each piece in the chopping as an atomic transaction) is equivalent to a legal

execution of the original transactions. In traditional transaction processing systems, legal executions of transactions are serializable executions. We say that the chopping is an SR-chopping if it is correct under serializability. One of the main results in [11] is the following theorem.

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

Please distinguish C edges from runtime conflicts (denoted by $\mathcal{C_{SR}}$). "$t_1 \mathcal{C_{SR}} t_2$" means that t_1 serializability. A C edge is a conflict edge shown in the chopping graph (off-line) and "$\mathcal{C_{SR}}$" is a conflict at runtime. In this paper, we use conflict cycles to indicate the cycles ("$t\mathcal{C_{SR}}^* t$") formed at runtime, rather than a cycle of C edges.

Other examples of asynchronous transaction processing work include unilateral commit paradigm [6], optimistic two-phase commit protocol with compensating transactions [7] and split transactions [9]. Many of these proposals depend on specific assumptions about application semantics or operational environments. Whether they can be combined with techniques such as ESR and transaction chopping is a topic of future research.

2 Combining ESR and Chopping Transactions Techniques

In this section, we first study the relationship between ESR and chopping transactions and design algorithms to combine them. We combine the benefits of ESR and chopping transactions in three ways:

Method 1 : using divergence control for SR-choppings (original choppings based on SR correctness criterion);

Method 2: using concurrency control for ESR-choppings (fuzzy choppings based on ESR correctness criterion which allow bounded inconsistency); and

Method 3: using divergence control for ESR-choppings. The methods are shown to obtain the best of both worlds.

Chopping transactions is an off-line tuning mechanism. In addition to the original chopping strategy (i.e., SR-chopping), we design a new chopping strategy (referred to as ESR-chopping) under ESR. On the other hand, two on-line mechanisms, concurrency control for serializability and divergence control for ESR, are studied in conjunction with two different chopping strategies. The combination of two lines of techniques is analogous to the product of two orthogonal dimensions. Table 1 shows the combinations of the two off-line approaches and the two on-line approaches, where CC is concurrency control and DC is divergence control. Three entries, ESR^1, ESR^2 and ESR^3, indicate epsilon serializable executions, and SR indicates serializable executions. ESR^1 is achieved by method 1, which runs SR-chopped transactions under divergence control algorithms. ESR^2 is achieved by method 2, which enhances Shasha's algorithm to chop transactions under ESR, but run them under concurrency control. ESR^3 is achieved by method 3, which runs ESR-chopped transactions under divergence control. This combination shows that ESR and chopping

Off-Line	On-Line	
	CC	DC
SR-Chopping	SR	ESR^1
ESR-Chopping	ESR^2	ESR^3

Table 1: Off-line v.s. On-line Approaches

transactions technique can complement each other, rather than being alternatives to each other.

2.1 Applying Divergence Control to SR-Chopping — Method 1

When an SR-chopping operates under a concurrency-control method (consider the two-phase locking), no conflict cycles are formed among pieces as well as among the original transactions at runtime. There is no conflict cycle among pieces because the concurrency control prevents conflict cycles from being formed. There is no conflict cycle among the original transactions because there is no SC-cycle. In this section, we apply divergence control to an SR-chopping, allowing limited conflict cycles among pieces to be formed. We must carefully distribute ϵ-spec among pieces such that the execution of the pieces is epsilon-serializable with respect to the original transactions.

An execution of $\mathcal{CHOP}(T)$ is said to be *serializable with respect to* the original transaction set T if it is equivalent to a serializable execution of T. In a similar way, an execution of $\mathcal{CHOP}(T)$ is said to be *epsilon-serializable with respect to* the original transaction set T if it is equivalent to an ESR execution of T. We say that an execution of $\mathcal{CHOP}(T)$ is correct under serializability if it is serializable with respect to T. We say that an execution of $\mathcal{CHOP}(T)$ is correct under ESR if it is epsilon-serializable with respect to T.

Given an SR-chopping $\mathcal{CHOP}(T)$, each ET $t \in T$ is assigned an ϵ-spec, denoted by $Limit_t$. A piece $p \in \mathcal{CHOP}(t)$ is an update piece if t is an update ET, regardless of the types of its operations (e.g., p may only have read operations). Our job is to figure out a correct and efficient assignment of ϵ-spec, $Limit_p$, for each piece p. Divergence control guarantees that the fuzziness of each piece is less than or equal to its permitted inconsistency limit, as expressed in Condition 1 (referred to as *Safe(p)*).

$$Z_p \leq Limit_p \quad (1)$$

Z_p (the fuzziness of p) is the amount of inconsistency accumulated at runtime. An assignment of $Limit_p$ is correct if the following condition holds (Condition 2).

$$if\ Z_p \leq Limit_p, \forall\, p \in \mathcal{CHOP}(t),\ then\ Z_t \leq Limit_t \quad (2)$$

Z_t ($t \in T$) is the fuzziness of t, which is the amount of inconsistency introduced to t in an execution of $\mathcal{CHOP}(T)$. That is, if the fuzziness of every piece $p \in \mathcal{CHOP}(t)$ is bounded by $Limit_p$ then the fuzziness of the original transaction t is bounded by $Limit_t$. An execution of an SR-chopping is correct under ESR, the following condition must hold (the safe condition in ESR definition).

$$Z_t \leq Limit_t \quad \forall\, t \in T$$

Since t is chopped into smaller ETs, there is no direct mechanism to accumulate fuzziness for t; divergence control only accumulates fuzziness for each piece. Z_t can only be derived from $Z_p, p \in \mathcal{CHOP}(t)$.

Lemma 1 *Given an execution of an SR-chopping, $\mathcal{CHOP}(T)$, for every original transaction, $t \in T$, the fuzziness of t is the sum of the fuzziness of all pieces in $\mathcal{CHOP}(t)$. The condition is expressed as follows.*

$$Z_t = \sum_{p \in \mathcal{CHOP}(t)} Z_p$$

Theorem 2 *Given an SR-chopping, $\mathcal{CHOP}(T)$, divergence control guarantees ESR executions with respect to the original transaction set T.*

Proof: This can be simply proved by using the conditions 1, 2 and Lemma 1. □

2.2 Efficient Inconsistency Limit Distribution

While $t \in T$ is chopped into pieces, $Limit_t$ is also distributed into $\mathcal{CHOP}(t)$. How to distribute $Limit_t$ is an important component for efficient transaction executions. Instead of taking the naive approach of evenly distributing $Limit_t$ among $\mathcal{CHOP}(t)$, we propose an efficient distribution approach by exploiting the knowledge of chopping graphs. This approach may lead to a situation where the sum of the fuzziness limits of all pieces is greater than the fuzziness limit of the transaction.

Since the job stream of T is known in advance (the key assumption of chopping transactions), from the structure of a given SC-chopping, we are able to tell that a piece $p \in \mathcal{CHOP}(t)$ will never be a part of a conflict cycle (i.e., $p \ \mathcal{C}_{S\mathcal{R}}{}^* \ p$), if p is not associated with a C-cycle. Thus, p will never cause any inconsistency. Given a chopping $\mathcal{CHOP}(t)$, each piece $p \in \mathcal{CHOP}(t)$ is marked either *restricted* or *unrestricted*.

- Restricted pieces $\mathcal{CHOP}_R(t)$: a restricted piece is a piece associated with C-cycles.
- Unrestricted pieces $\mathcal{CHOP}_U(t)$: an unrestricted piece is not associated with C-cycles.

With such information about pieces, our strategy is to distribute $Limit_t$ among the restricted pieces, $\mathcal{CHOP}_R(t)$, but not among the unrestricted pieces, $\mathcal{CHOP}_U(t)$. According to Condition 2 and Lemma 1, we derive the following condition.

$$Limit_t = \sum_{p \in \mathcal{CHOP}_R(t)} Limit_p \qquad (3)$$

Each unrestricted piece is assigned an infinite limit instead of a zero limit so that it can bypass the divergence control. This is because divergence control estimates fuzziness by detecting (immediate) conflicts, even no conflict cycle is possibly formed. An unrestricted piece may conflict with other pieces, but will never be a part of a conflict cycle. Since it causes no inconsistency in any circumstance, it should not be caught by divergence control due to an immediate conflict with other pieces. We discuss two ways of distributing $Limit_t$ among $\mathcal{CHOP}_R(t)$, static distribution and dynamic distribution.

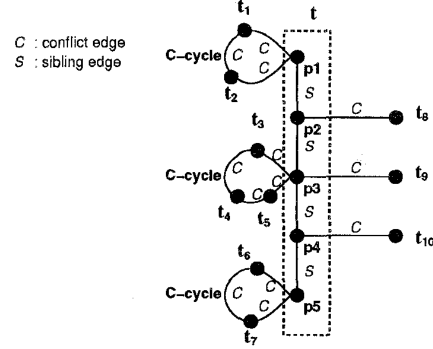

Figure 1: An Example of SR-Chopping

2.2.1 Static Distribution of Fuzziness Limits

We first consider an off-line approach of distributing $Limit_t$ among $\mathcal{CHOP}_R(t)$ according to the weights of the restricted pieces. For simplicity, we assume that all restricted pieces from t have the equal weights. Thus, each restricted piece $p \in \mathcal{CHOP}_R(t)$ is assigned a limit $Limit_p$ as follows.

$$Limit_p = \frac{Limit_t}{|\mathcal{CHOP}_R(t)|} \qquad p \in \mathcal{CHOP}_R(t)$$

Each unrestricted piece, q has $Limit_q = \infty$.

$$Limit_p = \infty \qquad p \in \mathcal{CHOP}_U(t)$$

For example, Figure 1 shows a partial chopping graph. Transaction t is chopped into five pieces: $\mathcal{CHOP}(t)=\{p_1, p_2, p_3, p_4, p_5\}$. There exist no SC-cycles in the chopping graph. However, three C-cycles are associated with p_1, p_3 and p_5. The C-cycle associated with p_1 contains p_1, t_1 and t_2. The second C-cycle contains four pieces (p_3, t_3, t_4 and t_5) and the third one contains three pieces (p_5, t_6 and t_7). The graph also shows some C edges that form neither SC-cycles nor C-cycles. This type of C edges do not cause inconsistency at runtime, since no conflict cycles can be formed by them. Assume $Limit_t$ is 51. Each of the three restricted pieces (p_1, p_3 and p_5) is assigned a limit of 17 (51/3) and both p_2 and p_4 (unrestricted pieces) are both assigned a limit of ∞. In the execution mentioned above, p_1 and p_2 have Z_1=10 and Z_2=5. They both commit, since their fuzziness does not exceed their limits. Assume that p_3 accumulates Z_3=20 at a certain point. It exceeds its limit; a proper action (blocked or rolled back) must be taken. Assume that the piece rolls back (and reset Z_3 to zero) and retrys until it eventually commits (see Section 4 for details). Both p_4 and p_5 will also execute and commit. With an infinite limit (∞) assigned to an unrestricted piece (p_2 or p_4), divergence control may accumulate an unlimited amount of import or export fuzziness for the unrestricted piece. The fuzziness accumulated by divergence control is over-estimated. For example, the two-phase locking divergence control detects immediate lock conflicts rather than conflict cycles. Actually, the unrestricted piece has no inconsistency at all, because it is not associated with either C-cycles or SC-cycles and thus it is impossible to form a conflict cycle at runtime.

2.2.2 Dynamic Distribution of Fuzziness Limits

In the previous example, we see the total fuzziness (totally 35) of t does not exceed its total limit ($Limit_t$=51) at the time when p_3 exceeds its local limit and rolls back. The problem arises due to an inefficient distribution among the restricted pieces; some pieces exceed their limits while the other still have "unused" quota. This problem can be alleviated by dynamically distributing $Limit_t$ among the restricted pieces of t. By dynamic distribution, the divergence control is able to utilize the total fuzziness limit of t. The system performance is increased since these unnecessary rollback situations are eliminated.

Based on the knowledge of the job stream of $\mathcal{CHOP}(T)$, let the dependency order imposed by the transaction program text be represented by DG($\mathcal{CHOP}(t)$), where nodes are $\mathcal{CHOP}(t)$ and edges are dependencies determined by the program text. For example, a transaction that transfers money from one account to another account is chopped into two pieces, p_1 that subtracts the amount from an account and p_2 that adds the amount to another account. We know that p_2 depends p_1 and there is an edge from p_1 to p_2. DG($\mathcal{CHOP}(t)$) maintains static (off-line) dependency information about pieces, but not the runtime dependencies due to operations conflicts. For simplicity, we assume DG($\mathcal{CHOP}(t)$) is a tree and the root of the tree is the first piece of t.

The algorithm in Figure 2 starts with procedure `DynamicExecution`, which schedules the first piece p_1 to run and assigns the entire limit ($Limit_t$) to the piece. `Schedule(S,`$Limit_S$`)` is a procedure which evenly distributes $Limit_S$ among all pieces in S. If p is a restricted piece, it executes with the assigned limit (*i.e.*, $\frac{Limit}{|S|}$). If p is an unrestricted piece, it executes with an infinite limit so that it is guaranteed to commit without the possibility of rollback. All pieces in S are scheduled to run in parallel. `ExecuteTransaction(p,`*Limit*`)` is a procedure that executes p as an individual transaction with a fuzziness limit, *Limit*. The procedure returns the "leftover" limit for p (denoted LO_p),

$$LO_p = Limit - Z_p$$

The "leftover" limit is the unused limit quota. If p is a restricted piece, it passes LO_p to the dependent pieces (S_p) (the dependent pieces of p), which are determined through the edge set of the dependency graph of $\mathcal{CHOP}$(t). S_p is then scheduled to run (by `Schedule`) with the leftover limit of p. If p is an unrestricted piece, it does not consume its limit quota at all (since it causes no fuzziness). When it completes, it schedules all its dependent pieces with the limit that it was originally assigned.

```
DynamicExecution(CHOP(t), Limit_t)
begin
  S = { p1 }
  Schedule(S, Limit_t);
end
Schedule(S, Limit_S)
begin
  Limit = Limit_S / |S|;
  for all (p ∈ S) in parallel
  begin
    if (p ∈ CHOP_R(t))
      LO_p = ExecuteTransaction(p, Limit);
    else if (p ∈ CHOP_U(t))
      begin
        ExecuteTransaction(p, ∞);
        LO_p = Limit
      end
    S_p = { q | (p, q) an edge of DG(CHOP(t)) }
    Schedule(S_p, LO_p)
  end
end
```

Figure 2: Dynamic Fuzziness Limit Distribution

3 ESR-Chopping – Method 2 and 3

In this section, we discuss the chopping strategy that allows limited SC-cycles in chopping graphs. We refer to this type of chopping as *ESR-chopping*. In addition to the knowledge of a chopping graph for a set of transactions T, we also assume the knowledge of the potential fuzziness that can be caused by a conflict corresponding to each C-edge in the chopping graph. We call the potential fuzziness the *weight* of a C-edge (W_C). For example, a bank customer is allowed to withdraw at most \$500.00 per day. The maximum fuzziness for the conflict between a piece that withdraws money and a piece that calculates the sum of all accounts is \$500.00. In case that the potential fuzziness of a C-edge is not possible to predict, the weight of the C-edge is simply considered to be ∞. This off-line information is crucial to ESR-chopping.

Someone may argue that the weights of C-edges are difficult to know in some applications. This is true. In those applications, the ESR-chopping strategy reduces to SR-chopping simply by assigning ∞ to the weight of every C-edge. This corresponds to the upward-compatibility of ESR. However, it is shown in many applications that the potential fuzziness of conflicts are often bounded. Besides the banking example, airline reservation systems often require a limit for each reservation. Also, a payroll system may limit the salary raise for each employee per year. Many database applications limit the updated amounts for security reason. For these applications, once the job stream is known in the first place, the weights of C-edges can be predicted (infinity for the worst case).

In general, we consider two types of fuzziness in a chopping.

- **Inter-sibling fuzziness** (Z^{is}): inconsistency between two pieces of an original transaction.
- **Inter-transaction fuzziness** (Z^{it}): inconsistency between two pieces from two different original transactions.

Chopping t into smaller pieces may introduce fuzziness into t, since it allows conflicting pieces (from a transaction other than t) to execute interleavingly with pieces in $\mathcal{CHOP}(t)$. Such fuzziness is the inter-sibling fuzziness. When pieces in $\mathcal{CHOP}(T)$ are executed serializably, there

is no inter-transaction fuzziness among pieces, but there may be non-zero inter-sibling fuzziness between two sibling pieces. The job of divergence control is to control and bound inter-transaction fuzziness at runtime. Import fuzziness and export fuzziness are inter-transaction fuzziness. It is obvious to see that an SR-chopping has no inter-sibling fuzziness. ESR-choppings have non-zero inter-sibling fuzziness, since limited numbers of SC-cycles are allowed. In this section, we discuss inter-sibling fuzziness, which is considered as off-line information between two sibling pieces.

Since we focus only on environments where database consistency cannot be compromised, we do not allow a chopping to have an SC-cycle that consists of two update pieces connected by a C edge (A piece $p \in \mathcal{CHOP}(t)$ is an update piece if t is an update ET). A chopping with an SC-cycle consisting of two update pieces connected by a C edge may cause permanent, inconsistent database states. We show this point in an example. Given two transactions, the first one t_1 is to transfer 100 from account X to account Y, and the second one t_2 is to compute the new amounts of X and Y by adding 10% interest to X and Y. Assume both X and Y are 1000 initially. Assume a chopping that divides t_1 into p_1^1, which subtracts d from X, and p_1^2, which adds d to Y. Since both ETs are update ETs and there exists an SC-cycle consisting of both ETs, one possible serializable execution is p_1^1, t_2 and p_1^2 in sequence, which causes X=990 and Y=1000 in the database. Consider two original transactions, this execution produces permanent inconsistent data in the database.

We show an example of inter-sibling fuzziness. Given a set of transactions $T = \{t_1, t_2\}$, transaction t_1 transfers D dollars from account X to account Y, and transaction t_2 reads both X and Y and displays the sum of two account balances. Consider a chopping, $\mathcal{CHOP}(T)$=$\mathcal{CHOP}(t_1) \bigcup \mathcal{CHOP}(t_2)$, where $\mathcal{CHOP}(t_1) = \{p_1^1, p_1^2\}$, and $\mathcal{CHOP}(t_2) = \{t_2\}$. The first piece p_1^1 subtracts amount D from account X and the second piece p_1^2 adds D to account Y. The chopping is a rollback-safe chopping, but not an SR-chopping (since there is an SR-cycle). $\mathcal{CHOP}(t_1)$ has a potential inter-sibling fuzziness of D. That is, in the worst case (consider the execution, $p_1^1 \rightarrow t_2 \rightarrow p_1^2$), p_1^1 and p_1^2 can introduce fuzziness D to t_2. In the next section, we study how to calculate the inter-sibling fuzziness.

3.1 Finding ESR-Chopping

A chopping is *ESR-correct* if any serializable execution of the chopping is equivalent to an ESR execution of the original transactions. To define an ESR-chopping, we first define the weight for each SC-edge, assuming that we are given a chopping $\mathcal{CHOP}(T)$ and the weights of all C-edges. The weight of an S edge (denoted W_S) is calculated as follows. Assume s is an S edge in the chopping graph of $\mathcal{CHOP}(T)$. First, we define the *C-edge set* associated with s (denoted $CE(s)$). Let pieces p and q be the two sibling pieces connected by s. $CE(s)$ contains every C edge connected to either p or q and in an SC-cycle.

The weight of s is the sum of $W_C(c)$, for all c in $CE(s)$ (Equation 4).

$$W_S(s) = \sum_{c \in CE(s)} W_C(c) \quad (4)$$

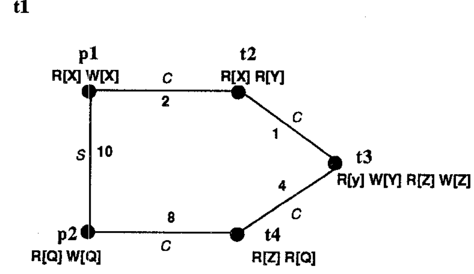

Figure 3: An Example of Inter-Sibling Fuzziness

For each transaction t, where $\mathcal{CHOP}(t)$=$\{p_1, p_2, \ldots, p_k\}$, the inter-sibling fuzziness of t ($Z^{is}{}_t$) is defined to be the sum of the weights of all S edges in $\mathcal{CHOP}(t)$.

$$Z^{is}{}_t = \sum_{s \text{ in } \mathcal{CHOP}(t)} W_S(s)$$

For example, given four transactions, t_1, t_2, t_3 and t_4. Transactions t_1 and t_3 are update ETs and t_2 and t_4 are query ETs. A chopping partitions t_1 into two pieces, p_1 and p_2. There is an SC-cycle (see Figure 3), formed by one S edge (s) and four C edges (c_1=(p_1, t_2), c_2=(t_2, t_3), c_3=(t_3, t_4), and c_4=(t_4, p_2)). Assume $W_{c_1} = 2$, $W_{c_2} = 1$, $W_{c_3} = 4$, and $W_{c_4} = 8$. The weight of the S edge is 10 (= 2 + 8).

Definition 1 *A chopping $\mathcal{CHOP}(T)$ is an ESR-chopping, iff (1) it is rollback-safe, (2) there exist no SC-cycles consisting of two update pieces connected by a C edge, and (3) the inter-sibling fuzziness is less than or equal to the fuzziness of the ET (Condition 5).*

$$\forall t \in T, \; Z^{is}{}_t \leq \text{Limit}_t \quad (5)$$

The first condition ensures that either all pieces in $\mathcal{CHOP}(t)$ commit or none of them commit. The second condition ensures no permanent inconsistency stored in the database system. The third condition ensures the safe condition of an ESR execution.

3.2 Applying Concurrency Control to ESR-chopping

We discuss the case where no inter-transaction fuzziness is allowed ($Z^{it}{}_p = 0, \forall p \in \mathcal{CHOP}(T)$) in an execution of a chopping.

Theorem 3 *Given an ESR-chopping, $\mathcal{CHOP}(T)$, concurrency-control methods guarantee ESR executions with respect to the original transaction set T.*

3.3 Applying Divergence Control to ESR-chopping

We discuss the case where both inter-transaction fuzziness and inter-sibling fuzziness are allowed ($Z^{it}{}_p \leq 0, \forall p \in \mathcal{CHOP}(T)$ and $Z^{is}{}_t \leq 0, \forall t \in T$) in

an execution of a chopping, $\mathcal{CHOP}(T)$. In general, a divergence control (DC) method controls Z^{it} and ESR-chopping controls Z^{is}. The ϵ-spec, (denoted by $Limit_t$), assigned to a transaction t is not considereded entirely as the import limit or export limit used by the divergence control for accounting purpose, since ${Z^{is}}_t$ is considered as part of the fuzziness of t. Let $Limit_t^{DC}$ be the limit actually used by the divergence control method, shown as follows (Equation 6):

$$Limit_t^{DC} = Limit_t - {Z^{is}}_t \quad (6)$$

The limit equal to ${Z^{is}}_t$ must be reserved for tolerating the fuzziness caused by conflicts from SC-cycles. Given a chopping, $\mathcal{CHOP}(t)$, and the limit ($Limit_t$ derived from Equation 6), the same static or dynamic distribution algorithm in Figure 2 is used to bound fuzziness for the execution of an ESR-chopping.

Theorem 4 *Given an ESR-chopping, $\mathcal{CHOP}(T)$, divergence control methods guarantee ESR executions with respect to the original transaction set T.*

4 Chopping Up Epsilon Transactions in Distributed Environments

The traditional approach to distributed transaction processing [4, 1] requires the use of a commit protocol to ensure failure atomicity of distributed transactions; either all sub-transactions of a distributed transaction commit or none of them commits. Secondly, a global validation process is needed to ensure concurrency atomicity (i.e., serializable executions of transactions); all sites have the same serialization order. In some distributed transaction processing systems, the commit protocol also carry out the global validation process. Unfortunately, the traditional commit protocols such as the two-phase commit protocol have several potential problems. First, they are expensive in general. Usually, several rounds of message passing among sub-transactions are needed in order to commit a distributed transaction. Second, they may be blocked for an indefinite period due to a node failure or a link failure. Such a blocking condition for a transaction can further cause other dependent transactions to be blocked. In these cases, performance and availability are severely degraded.

In contrast to the synchronous nature of serializable transaction processing systems, ESR and chopping transactions are two asynchronous transaction processing approaches. ESR relaxes commit and abort dependencies among transactions by allowing them to commit with a limited amount of inconsistencies. On the other hand, chopping transactions enforces no commit protocol among the pieces from an original transaction and allows individual piece to commit asynchronously.

4.1 Chopping Technique Complementing Distributed Divergence Control

Two categories of distributed divergence control algorithms have been proposed in [8]. The first category of algorithms require that the local orderings of all the sub-transactions of a distributed transaction are the same. In such an environment, the fuzziness of a distributed transaction is simply the sum of its sub-transactions' local fuzziness. For the second category of algorithms, the local orderings of all the sub-transactions of a distributed transaction may not be the same and an additional amount of global inconsistency is introduced into the distributed transaction. In terms of conflict cycles, the first category of algorithms only allow (limited) conflict cycles to be formed locally and the second category of algorithms further allow (limited) conflict cycles to be formed among multiple sites. Both cases require a commit protocol to ensure failure atomicity as well as a bounded degree of concurrency atomicity, and thus have the potential problems discussed above.

Chopping transactions technique can be used to complement these divergence control algorithms by reducing the need of commit protocols. At run time, each chopped piece is an individual transaction and they can commit independently. For a banking example, given three transactions (t_1, t_2 and t_3), t_1 is to transfer money from checking account x to saving account y, t_2 is to get the account balance from x and t_3 is to get the account balance from y. We also assume that x and y are located at two different sites. Transaction t_1 can be chopped into two pieces, p_1 and p_2. Piece p_1 subtracts the amount from x and p_2 adds the amount to y. No commit protocol is needed between p_1 and p_2, as long as we can guarantee the p_2 will eventually commit once p_1 commits.

To guarantee that the subsequent pieces of a distributed transaction will eventually commit once the first piece commits, we adopt the mechanism of persistent transmission [6] or recoverable queues [3], which are offered in commercial transaction monitors. A *recoverable queue* is an inter-site communication channel, through which data are guaranteed to survive site failures as well as link failures. Unlike other types of inter-site communication channels, both the sender and the receiver for a recoverable queue must be transactions instead of arbitrary programs. Messages sent through a recoverable queue are parts of transaction effects; either all messages become deliverable when the sending transaction commits or none of them is delivered when the transaction aborts. Once a message becomes deliverable, it must be consumed by a transaction that eventually commits. When the receiving transaction aborts, all messages read by the transaction are put back to the recoverable queue and remain deliverable.

For simplicity, we assume that each piece of a distributed transaction resides at only one site (i.e., no piece across multiple sites). Instead of using the regular communication channel, each piece uses recoverable queues to communicate with its sibling pieces. Once the first piece commits, all other pieces can commit asynchronously as long as they are guaranteed to commit eventually. The eventual commitment of a piece can be guaranteed since (1) it is rollback-safe (see the rule of SR-chopping), and (2) for the cases of rollback due to a lock conflict or deadlock, a process handler can be used to resubmit the piece until it commits. Using recoverable queues is a key component that allows a piece to re-run.

In this approach, we use recoverable queues but we eliminate the use of a commit protocol for a distributed transaction. Let us examine the performance tradeoff. In order to implement the transaction property of recoverable queues, it is necessary for a recoverable queue to log all sending and re-

ceiving messages with timestamps. We assume a simple approach where messages are received only after the sender's transaction commits. Such an implementation of recoverable queue does not cost an additional (control) message than any other reliable communication channel does. On the other hand, a commit protocol (consider the two-phase commit) requires at least two rounds of message passing. In the distributed environments where communication time is long and less predictable, we increase transaction performance by reducing two rounds of messages. Certainly, we increase the system availability, since no commit protocol is needed.

For example, assume that a distributed transaction u transfers money from one bank branch in New York and another bank branch in Los Angeles, while another distributed transaction q is calculating the sum of all accounts in both branches. The transaction q has an import inconsistency limit \$10,000 and the transaction u has an export inconsistency limit \$10,000. We chop u into two pieces, u_1 and u_2, each of which is assigned an export inconsistency limit of \$5,000 (i.e., \$10,000/2). The first piece u_1 subtracts the amount from the account in New York branch, sends a message to the second piece in Los Angeles branch through a recoverable queue and then commits. The second piece u_2 reads the amount from the recoverable queue and adds it into the account in Los Angeles branch. We also chop q into two pieces, q_1 and q_2, each of which is assigned an import limit \$5,000. The first piece q_1 calculates the sum of the accounts in the New York branch, sends the result to q_2 through a recoverable queue and then commits. The second piece q_2 calculates the sum of the accounts in the Los Angeles branch, reads the amount from the recoverable queue, and then commits. A conflict cycle is possible, since both pieces in the same branch access one common account. However, as long as the amount of being transferred is less than \$5,000, the local divergence control allows both pieces to proceed. Without using the chopping transaction technique, both distributed transactions must perform a global validation process at commit time to ensure that both sites have the same serialization order. Remember that if two sites have different serialization orders, there may be global fuzziness (due to the cross-site conflict cycles). By knowing the chopping graph, all pieces can commit without going through the global validation. A round trip of message passing can takes from a few hundreds milliseconds to a few seconds. As a result, this approach takes a few hundreds milliseconds or a few seconds less than the traditional approach does.

5 Evaluation

In this paper, we have designed three methods to combine ESR and transaction chopping:

1. SR-chopping under ESR divergence control.
2. ESR-chopping under SR concurrency control.
3. ESR-chopping under ESR divergence control.

Method 1 is conceptually the simplest of the three, since it combines an existing SR-chopping algorithm with an existing ESR divergence control algorithm. Although we needed to solve the problem of distributing ϵ-spec in distributed systems, the practical implemented of Method 1 is straightforward. Method 2 is somewhat more complex, since ESR-chopping must take into account the fuzziness propagation between chopped transaction pieces. On the practical side, it runs on existing TP and DBMS systems without requiring any database support for ESR. From the performance point of view, Method 2 gives finer-grain chopping than Method 1. However, since the concurrency control algorithms are more restrictive than divergence control algorithms during run-time, there are scenarios where SR-chopping on divergence control wins and others in which ESR-chopping on concurrency control wins. Method 3 is the most sophisticated of the three techniques we devised. It achieves the finest static chopping of the algorithms among the three methods, as well as the original SR-chopping running under concurrency control. Similarly, Method 3 also achieves the best concurrency at run-time, since divergence control methods allow more concurrency than classic concurrency control. On the negative side, this possibility asks for the most investment on the project.

References

[1] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–222, June 1981.

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

[3] P.A. Bernstein, M. Hsu, and B. Mann. Implementing recoverable requests using queues. In *Proceedings of 1990 SIGMOD International Conference on Management of Data*, pages 112–122, May 1990.

[4] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

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

[6] M. Hsu and A. Silberschatz. Unilateral commit: A new paradigm for reliable distributed transaction processing. In *Proceedings of the Seventh International Conference on Data Engineering*, Kobe, Japan, February 1990.

[7] E. Levy, H. Korth, and A. Silberschatz. An optimistic commit protocol for distributed transaction management. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, May 1991.

[8] C. Pu, W.W. Hseush, G.E. Kaiser, P. S. Yu, and K.L. Wu. Distributed divergence control algorithms for epsilon serializability. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, Pittsburgh, May 1993.

[9] Calton Pu, Gail E. Kaiser, and Norman Hutchinson. Split-transactions for open-ended activities. In Francois Bancilhon and David J. Dewitt, editors, *14th International Conference on Very Large Data Bases*, pages 26–37, Los Angeles CA, August 1988.

[10] K. Ramamrithan and C. Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, to appear 1994.

[11] D. Shasha, E. Simon, and P. Valduriez. Simple rational guidance for chopping up transactions. In *Proceedings of 1992 SIGMOD International Conference on Management of Data*, pages 298–307, May 1992.

[12] K.L. Wu, P. S. Yu, and C. Pu. Divergence control for epsilon-serializability. In *Proceedings of Eighth International Conference on Data Engineering*, pages 506–515, Phoenix, February 1992. IEEE/Computer Society.