Correctness and Parallelism in Composite Systems

Gustavo Alonso
Database Research Group
Institute of Inf. Systems
ETH Zentrum
Zürich CH-8092, Switzerland
alonso@inf.ethz.ch

Stephen Blott*
Lucent Technologies
(Bell Labs)
600 Mountain Avenue
Murray Hill, NJ 07974, USA
blott@research.bell-labs.com

Armin Fessler
Database Research Group
Institute of Inf. Systems
ETH Zentrum
Zürich CH-8092, Switzerland
fessler@inf.ethz.ch

Hans-Jörg Schek
Database Research Group
Institute of Inf. Systems
ETH Zentrum
Zürich CH-8092, Switzerland
schek@inf.ethz.ch

Abstract

In recent years, databases have started to be used as intelligent repositories for a variety of semantically-richer systems. A consequence of such architectures is that transaction scheduling takes place throughout composite systems consisting of layered subsystems. Such transaction architectures have been studied extensively. Existing theory, however, limits the degree of parallelism, and makes a number of simplifying assumptions which cannot be taken for granted in practice.

This paper proposes a new model and correctness criterion, *stack conflict consistency*, for composite transactional systems. The main contribution of the new model is to establish the correctness conditions under which higher degrees of parallelism can be achieved between operations of the same transaction, as well as between conflicting operations of different transactions, in a uniform way. This possibility, although hinted at previously, has not yet been exploited in practical composite systems. Hence, we hope to improve the practical impact of many key results in this area.

1 Introduction

In general, a composite system is one in which scheduling takes place simultaneously and independently at several levels. For such systems, the most relevant transaction models are those based on a multilevel approach, i.e., those in which each scheduling level is implemented as a function of the operations and objects provided by the immediately-lower level [Wei91, WS92]. The use of several levels of scheduling allows implementation details to be abstracted out, with the degree of parallelism being determined by higher-level semantics. This configuration has received quite some attention since it corresponds to the internal architecture of many database management systems [GMB+81, MHL+92, Lom92, WH93]. One important aspect of such systems is the degree of independence possible between schedulers at different levels. Sufficiency conditions for independent scheduling at different levels have been elaborated in the literature [BBG89, Wei91]. Beeri, Bernstein and Goodman identify *order preservation* as a sufficient handshaking mechanism between schedulers [BBG89]. Weikum proposes *conflict preservation* ('Axiom 1' in [Wei91]) to the same end. Although both order and conflict preservation hold for many practical systems, they cannot always be taken for granted, and indeed cannot be guaranteed for a number of commercial systems (including Oracle, and ObjectStore in multi-version mode) [BSW88].

Beyond these restrictions, existing models exhibit two main problems:

1. Intra-transaction parallelism is only possible between unordered operations, and
2. Parallelism is often restricted to non-conflicting operations in current implementations.

With respect to intra-transaction parallelism, we observe the following severe short-coming of the state-of-the-art. If within a single transaction two complex operations are invoked, they can either be unordered (may be executed in parallel) or ordered. The latter case requires the second operation to be executed only after the first has terminated. This is the case traditionally considered [BHG87, BBG89, Wei91]. However, this approach is unnecessarily restrictive. Consider, for instance, the example in Figure 1. One transaction is illustrated in which all employees are first awarded a bonus, and then the new average salary is computed. The ordering of these two operations matters. Existing transaction models require ordered operations to be executed serially, the second must wait for the first to complete. However, as illustrated in the example, executions can in this case be interleaved, so long as the serialisation graph is respected at all levels. Note that, the intuition is similar regardless of whether the operations belong to the same or different transactions.

This observation is more fundamental than it may appear at a first glance. Assume in a composite system we want to exploit the ability of databases to execute many transactions in parallel, but now on behalf of one client's global transaction. Or consider problems beyond OLTP databases such as the parallel computation of complex algorithms and other tasks which can be decomposed into many smaller tasks and parallelized. Could not database transaction technology (repeatedly) be used for a parallel execution of many task simply by "selling" the tasks to a database scheduler as transactions? Database systems execute many independent

*This work was done while the author was a member of the Database Research Group of ETH Zürich.

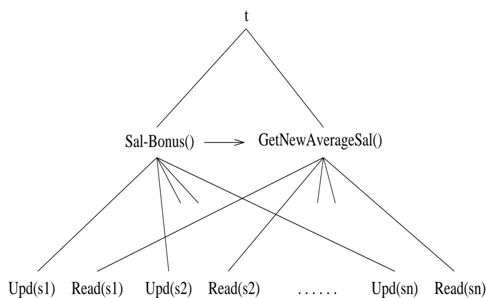

Figure 1: Intra-transaction parallelism: Interleaved execution of ordered operations within a transaction

transactions in parallel even if they conflict. Serialisability takes care of correctness in that it makes sure that there exists at least one serial execution being equivalent to the parallel one. However there are two basic assumptions behind this idea. First, information is passed from one transaction to the other only via shared (database) objects and only via conflicting operations of different transactions. Second, all serial executions are acceptable, i.e., transactions are independent of each other.

Both assumptions are not necessarily true in the cases considered here. Since we are interested in composite systems, the notion of transaction is somewhat more complex than in the traditional model. Transaction with the same parent are certainly not independent of each other in all cases, therefore, not all equivalent serial schedulers are acceptable, only those that match the internal logic of the transactions. In addition, there may be flow of information not only via shared database objects but also via (local) variables, not controlled by any scheduler. To address this issue, the classical read/write model assumes write operations are always a function of previous read operations. A scheduler must therefore observe all intra-transaction orders in order to preserve transaction semantics. Thus, in the classical model, there is no parallel execution of actions. There is also no need for it because of the atomicity of reads and write operations.

In the case of composite systems, however, the model needs to be extended. For this purpose, two orderings are considered:

Weak Order: This is the case in which all data flow takes place through database objects. Transactions or operations which are weakly ordered must appear "as if" the operations are serialised in the order indicated. Hence, weakly-ordered transactions or operations can be executed in parallel as long as the final result is an execution which is serialisable in the order indicated.

Strong Order: The strong orders, on the other hand, capture external flow of information between operations or transactions. In this case, execution is serial, with a transaction or operation being delayed until the previous one completes.

Note that it is also possible for transactions and operations to be executed serially, but have no control- or data-flow between them. These, however, need not be ordered in the strong or weak orders. In particular there is no specific order for transactions that are executed temporally one after the other but without flow of information between them.

Such a transaction model with two partial orders may seem unnecessarily complex. But, as the paper will show, it complements elegantly and directly our second objective, namely allowing the parallel execution of conflicting operations. In the classical theory [BHG87] conflicting operations must be executed serially. This assumption, again, is reasonable for short, low-level operations such as `read` an `write`, but must be questioned for high-level operations where serial execution may have a severe impact on performance. Consider, for instance, the executions illustrated in Figure 2. Two different updates to a collection of salaries are depicted. The first adjusts all salaries according to general inflation, and the second awards all employees a $100 bonus. These are conflicting updates, and existing implementations, e.g., based on either high-level or table locking, force them to be executed serially. If the table fits in main memory, then performance may not be seriously affected. If, on the other hand, the table does not fit in main memory, then serial execution forces two consecutive scans, which seriously degrades performance.

There is a second important practical aspect: Detecting conflicts at a high level is frequently complex. In practice simplifications are applied. For example, predicate locking is often based on approximated predicates [DPS83]. This introduces artificial conflicts which force transactions to be delayed even if no actual conflict exists. As a consequence, existing multilevel transaction systems are forced to delay operations as long as potentially-conflicting operations are executing. Since this can occur at any level, the overall degree of parallelism is severely limited.

From a theoretical point of view [BBG89, Wei91], the main result regarding parallel execution of conflicting operations is that serialisation orders obtained at different levels of a composite system must not contradict each other. More precisely, if t and t' conflict and if their parent transactions T and T' are serialised T before T', then it is required that t and t' are also serialised t before t' if they have conflicting children. Note that high-level locking is clearly correct in that it transforms the weak order restriction into a strong execution order: If t' requires a lock that t holds it cannot be executed before t terminates. This is similar to the strong order case. Although turning all execution orders into strong ones solves the problem, it is unnecessarily restrictive in many cases.

As can be seen from both aspects, the intra-transaction parallelism and the parallel execution of conflicting tasks, it seems quite natural to investigate more closely a model where a clean distinction between the strong order and the weak order is made. Thus, the main question addressed in this paper is how these order constraints are passed between schedulers and how and under what circumstances they must be observed. We provide a new model and correctness criterion, called *stack conflict consistency* — the name reflects the treatment of ordering contraints which are imposed by higher level schedulers onto the lower level schedulers. The new model allows the parallel execution of weakly ordered operations, as long as their serialisation graph is preserved. This provides the theoretical foundation for a 'parallel-do' primitive applicable as long as external dependencies do not force sequential execution. We expect such functionality to have a significant impact on the performance of parallel databases. A second contribution of this paper is pedagogical. An important advantage of the new formulation is that it is based on a simple extension of the classical theory. It explains a number of advanced models, including multilevel and nested transactions [Wei91, BBG89], as

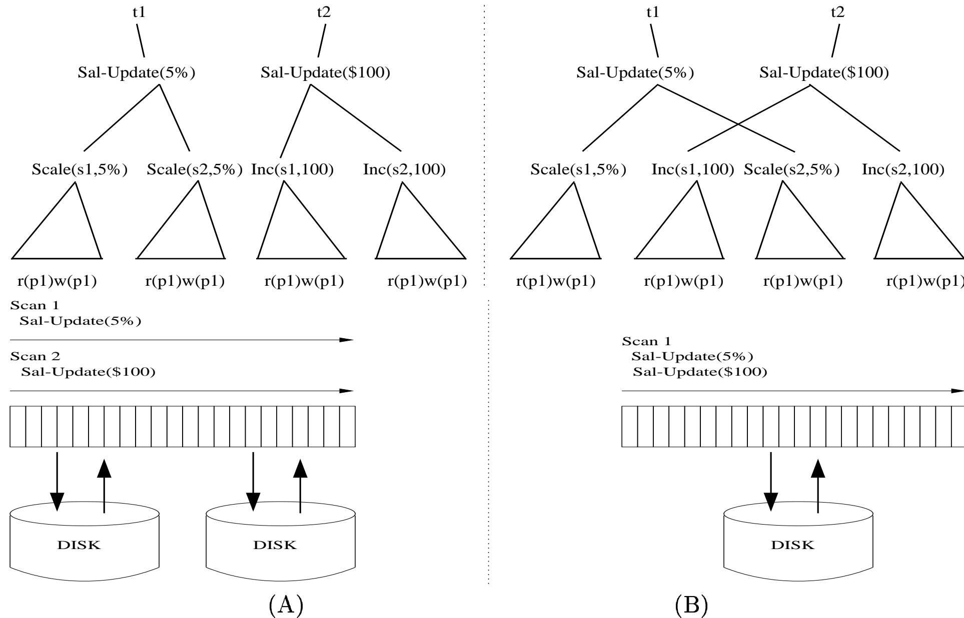

Figure 2: Serial (A) and Parallel (B) Execution of Conflicting Operations

straight-forward extensions of the well-known and widely-accepted concepts of the classical theory [BHG87]. We hope thereby that advanced concurrency-control results can achieve the same wide acceptance afforded the classical theory.

This paper is organized as follows. Section 2 introduces the new correctness criterion, Section 3 provides a comparison with existing models, Section 4 discusses the new model and its impact in practice, and Section 5 concludes.

2 Stack Conflict Consistency

This section develops the new model and correctness criterion, *stack conflict consistency*, as a straight-forward extension of classical transaction theory [BHG87]. Section 2.2 extends conflict serialisability to the criterion *conflict consistency* for flat systems, and Section 2.3 extends this criterion further to *stack conflict consistency*, our correctness criterion for composite systems.

2.1 Basic Model

Traditionally, operations act upon data items on a database. In composite systems, transactions interact with services and resource managers at higher levels of abstraction. Thus, the 'operations' considered here are in fact invocations of the services provided by the next scheduler. Only the operations at the lowest level interact directly with data items in the traditional sense. Transactions are sets of such invocations.

Conventionally, data- and control-flow dependencies are enforced by sequential execution. Thus, in the traditional transaction model, only one type of ordering is considered, usually called intra-transaction order. As the example in Figure 2 shows, it is possible to maintain data- and control-flow dependencies, even if there is no sequential execution. Since our goal is to expand correctness criteria for parallel executions, two forms of ordering are introduced here.

Definition 1 (Transaction) *A* transaction*, t, is a triple $(O_t, <_t, \ll_t)$, where O_t is a set of* operations*, $<_t$ is a partial order on O_t, termed the* weak (intra-)transaction order*, and $\ll_t$ is a partial order on O_t termed the* strong (intra-)transaction order. □

The transaction orders are constraints to be observed when a transaction is executed. The *strong transaction order* $o \ll_t o'$ between two operations o and o' guarantees that o' is invoked only after o has returned. This corresponds to the classical intratransaction order. Such a constraint is used to enforce sequential execution and is necessary when return values of o are needed as input parameters for o'. The weak transaction order $o <_t o'$ allows *parallel* execution of o and o' but under the constraint that its effects to the database are the same as if executed with $o \ll_t o'$.

It should be obvious that the weak transaction order only is useful when operations are complex and are considered as subtransactions in a composite system. Otherwise, if we are at some low level where operations are atomic, interleaved execution under a weak order constraint does not make sense. On the other hand, turning all weak transaction orders into strong ones generally will convert our transaction model into the classical one, but the degree of parallelism would be severely restricted.

Recovery issues are not discussed in detail here. They can be dealt with using the *unified model* [SWY93, VYBS95], in which all transactions ultimately commit. In order to simplify the presentation, it suffices to assume committed projections.

A *schedule* (sometimes termed a 'history' [BHG87], or a 'log' [MGG85]) models the interleaved execution of a number of transactions. The classical model assumes transactions are independent, a fact often expressed by the statement that *every* serial execution of the transactions as considered correct. This is too general for two reasons. First, in a composite system, operations within a transaction are treated as transactions at the next lower level. The dependencies be-

tween the sub-transactions of the same parent transactions need to be passed on to the next scheduler. Second, even in a flat model, a transaction may trigger other transactions as in the case of workflow definitions linking several transactions. Thus, the definition of a schedule must take into consideration the need to express these relations.

Definition 2 (Schedule)

A schedule *is a six-tuple* $(T, \rightarrow, \mapsto, Con, <, \ll)$ *where:*

- *T is a set of transactions.*
  Let O denote the set of all operations of T's transactions, i.e., $O = \bigcup_{t \in T} O_t$.
- *Con, the* conflict predicate*, is a symmetric binary relation over $O \times O$.*
- *$\rightarrow$ and $\mapsto$ are the* weak *and* strong input orders*, partial orders over T. We further require $(\rightarrow \cup \mapsto)$ to be acyclic.*
- *$<$ and $\ll$ are the* weak *and* strong execution orders*, partial orders over O such that:*
  1. *$\forall o, o' \in O : (o, o') \in Con \Rightarrow (o < o') \vee (o' < o)$, i.e., all pairs of conflicting operations are at least weakly ordered,*
  2. *(a) $\forall t \in T,\ \forall o, o' \in O_t : (o <_t o') \Rightarrow (o < o')$,*
     (b) $\forall t \in T,\ \forall o, o' \in O_t : (o \ll_t o') \Rightarrow (o \ll o')$,
     that is, all weak and strong transaction orders are contained in the weak and strong execution orders, respectively,
  3. *Whenever $t \mapsto t'$, then $\forall o \in O_t, \forall o' \in O_{t'}$: $o \ll o'$, that is, a strong input ordering is always propagated downwards, and*
  4. *$< \cup \ll$ is acyclic.* □

The weak and strong input orders are used to specify dependecies between transactions. Note that the weak and strong input and execution orders are partial orders, hence transitive and acyclic. The major difference between this and traditional definitions is the inclusion of the input orders, and the distinction between weak and strong orders. The strong input order requires that one transaction completes before the next transaction starts. In the classical model, such restrictions are not imposed; transactions are considered independent.

Point 3 above forces a strong input ordering between transactions to be reflected by a strong execution ordering between their children. This effectively separates the executions of strongly ordered transactions. The weak input order will be used when transactions can be executed in parallel, but the final effect of the execution must be the same as when executed serially. As it will be shown below, the weak input order restricts possible serialisation orders by forcing them to agree with that of weakly-ordered transactions. Note that the weak input order is not automatically propagated downwards. This point will be crucial when discussing stack schedules. The weak execution order arises from weak intra-transaction orderings (Point 2(a) above), and from conflicting operations (Point 1 above). Weak transaction orders are contained in the weak execution order, and strong transaction orders are contained in the strong execution order (Points 2(a) and 2(b) above). Point 4 enforces *well formed* schedules in which the orderings do not contradict each other. Schedules violating this point correspond to abstract cases without a realistic counterpart as a cycle would imply that operations that have been executed in a given order (the strong order) have a conflicting relation in the opposite direction. Note that this point also requires well formed transactions, i.e., the union of the strong and weak transaction orders must be acyclic.

In the classical model, correctness of schedules is defined in terms of *conflict equivalence* between schedules. A schedule is *conflict serialisable* if and only if it is conflict equivalent to a serial schedule. In this paper, and similarly to the traditional theory [BHG87], a schedule is *serial* if for every pair of transactions, all of the operations of one are executed before any operations of the other. From here,

Definition 3 (Conflict Equivalence) *Two schedules S and S' are* conflict equivalent *if they are over the same set of transactions and have the same operations, they have the same conflict predicates, and for any pair of conflicting operations, the weak output order is the same.*

This definition is essentially the same as in conventional serialisability theory [BHG87] in that it considers only the ordering of conflicting operations. Note that it is possible that two schedules are conflict equivalent while having different input orders.

2.2 Conflict Consistency for Single-Level Schedules

In the new approach, the weak and the strong input orders restrict the freedom of the scheduler in choosing a serialisation graph. We formulate this formally as follows.

Definition 4 (Conflict Consistency (CC)) *A schedule S is* conflict consistent *if it is conflict equivalent to a serial schedule S', which contains the weak and the strong input order, i.e., $\rightarrow_S \subseteq \rightarrow_{S'}$ and $\mapsto_S \subseteq \mapsto_{S'}$.* □

Note that two schedules may be conflict equivalent while one of them is conflict consistent and the other is not. The rationale behind this lies on the notion of consistency. Conflict equivalence implies that the schedules are similar from the system's point of view, i.e., they produce the same results. From the user's point of view, however, schedules are only valid when they preserve the specified order, i.e., are consistent with the input orders. Thus, it is possible for two schedules to be conflict equivalent, and yet one be consistent with what the user specified while the other is not. Hence, one is conflict consistent and the other is not, although from the system's point of view the schedules are the same.

Definition 5 (Serialisation Graph $\hookrightarrow$) *Given a schedule S, its* serialisation graph*, denoted $\hookrightarrow$ here, is an irreflexive binary relation on $T \times T$, in which $t \hookrightarrow t'$ is contained if $t \neq t'$ and: $\exists o \in O_t,\ \exists o' \in O_{t'} :\ Con(o, o') \wedge (o < o')$* □

Intuitively, the serialisation graph gives the derived execution order of transactions based on the weak execution order of conflicting operations. The correctness of a scheduler can be derived from the serialisation graph as follows:

Theorem 1 *A schedule S is conflict consistent iff the union of the weak and the strong input order and the serialisation graph, that is $(\hookrightarrow \cup \rightarrow \cup \mapsto)$, is acyclic.*

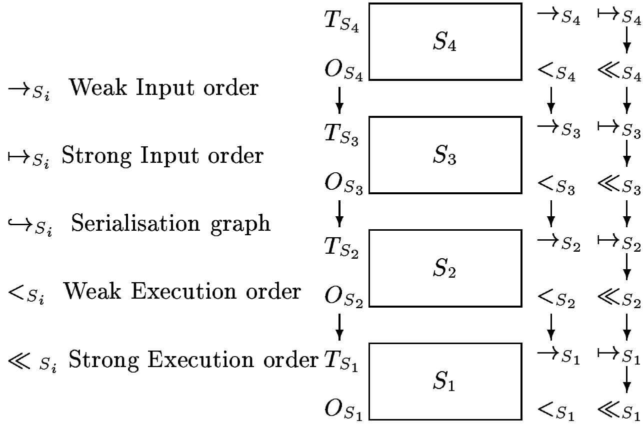

Figure 3: A 4-level stack schedule sketch. The execution orders of S_{i+1} are 'passed' as the input orders to S_i. Stack conflict consistency requires: $\forall i \in \{1, \ldots, 4\}$: $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i} \cup \mapsto_{S_i})$ are acyclic.

Proof. *(If)*. Since $(\hookrightarrow \cup \rightarrow \cup \mapsto)$ is acyclic, there exists a total ordering over transactions which contains $(\hookrightarrow \cup \rightarrow \cup \mapsto)$. Let $:<:$ be such an ordering of transactions. Let S' be a schedule which differs from S only in that, for pairs of non-conflicting operations o_1 and o_2 from distinct transactions t_1 and t_2, if $t_1 :<: t_2$ then $o_1 < o_2$ and $o_1 \ll o_2$. By definition 3, S' is conflict equivalent to S. S' is a serial schedule. Hence, by definition 4, S' is conflict consistent, as required.

(Only if). If S is conflict consistent, then there must exist a conflict equivalent serial schedule. Let S' be such a serial schedule. Then, S' defines a total order $:<:$ on the transactions of S. Since S and S' are conflict consistent, we have:

1. $(\rightarrow_S \cup \mapsto_S) \subseteq :<:$
2. $\hookrightarrow_S \subseteq :<:$

Therefore, $(\hookrightarrow_S \cup \rightarrow_S \cup \mapsto_S)$ cannot contain a cycle, as required. □

The intuition behind conflict consistency is that the serialisation order captures causal dependencies within a schedule, i.e., the flow of information that takes place via shared database objects. The requirement that the serialisation order does not contradict the strong input order guarantees that the "internal" flow of information does not contradict the "external" flow of information (which is the dependency captured by the strong order). The requirement that the serialisation order does not contradict the weak input order gurantees that operations that conflict are ordered in the same way as specified by the weak input order. If no conflict is detected, the operations can be executed in parallel, regardless of the weak input order specified. This is the main difference and advantage of the weak order compared with the strong one. With respect to the weak input order, similar ideas were hinted at by Moss, Griffith and Graham [MGG85], though they were not pursued further [MGG86].

2.3 Stack Conflict Consistency

The advantage of the formulation above is that conflict consistency can be used to capture a variety of important dependencies between schedulers in composite systems. In particular, the weak input order allows the behavior of schedulers to be restricted such that inconsistent serialisation graphs are avoided. Composite systems are modeled as a *stack* of schedules, one on top of the other. The outputs of one schedule are 'plugged' to the inputs of the next, as illustrated in Figure 3. This leads naturally to the following formulation.

Definition 6 (Stack Schedule (SS))
SS, an n-level stack schedule*, consists of n schedules* $S_1, \ldots, S_n$,[1] *such that, for* $1 < i \leq n$*:*

- $T_{S_{i-1}} = O_{S_i}$,
- $\rightarrow_{S_{i-1}} = <_{S_i}$, *and*
- $\mapsto_{S_{i-1}} = \ll_{S_i}$. □

As Figure 3 shows, in a stack schedule, the operations, weak execution order and strong execution order are passed from one schedule as the transactions, weak input order and strong input order of the next. Notice that only the strong ordering is automatically propagated to all levels. This is a consequence of Definitions 2 and 6. Whether the weak order is passed from one level to the next depends upon whether conflicting operations are involved. Note also, that by Point 4 of Definition 2, $(\rightarrow \cup \mapsto)$ of the next lower level is acyclic.

The advantage of modeling the input order explicitly is that the extension of conflict consistency to stack schedules becomes straight-forward.

Definition 7 (Stack Conflict Consistency (SCC))
An n-level stack schedule SS is stack conflict consistent *iff each individual schedule* S_i *in SS is conflict consistent, for* $1 \leq i \leq n$. □

2.4 Discussion of Stack Conflict Consistency

The novel aspect of SCC is the use of the weak input order. The weak input order indicates how to serialise transactions *in the case* they need to be serialised. For example, a specification $t_1 \rightarrow t_2$ prevents the serialisation order from being $t_2 \hookrightarrow t_1$. If the serialisation order does not establish any relation between t_1 and t_2, then their execution trees can be interleaved, and might even be reversed. This increases the possibility of parallelizing operations.

While the strong input order is pushed down through all levels and forces sequential executions, in stack schedules the weak input order is never propagated automatically. The weak input order only indicates to a schedule the restrictions imposed by the schedule immediately above, i.e., the serialisation order cannot contradict the weak input order. The weak input order need only order conflicting operations and operations ordered by transactions themselves. In practice, this means that a weak input ordering can even disappear when a level is reached at which operations do not conflict. An example of this is the schedule illustrated in Figure 4(A). Although the weak input order indicates $t_1 \rightarrow t_2$, the children do not conflict and are not ordered by the execution order. Therefore, the input is not propagated, allowing the next level to reverse the execution and serialisation orders.

Figure 4(B) shows a similar example in which the children of t_1 and t_2 conflict. In this case, the weak execution order must respect the weak input order, therefore it

[1] Frequently, a number of schedules are considered together; here, these are denoted $S_1, \ldots, S_n$. We implicitly use the notation $T_{S_1}, \ldots, T_{S_n}$ to refer to the sets of transactions of these schedules, and $<_{S_1}, \ldots, <_{S_n}$ to refer to their execution orderings, etc.

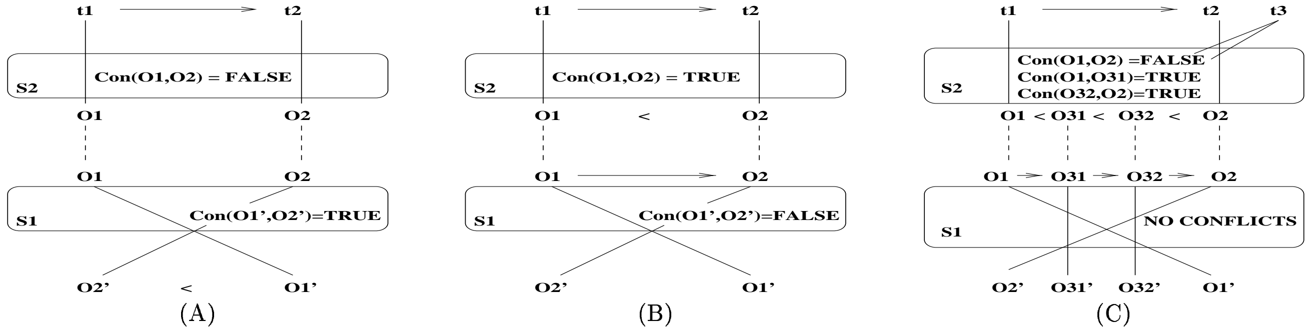

Figure 4: Correct executions allowing different interleavings depending on the conflict relations

is propagated downwards. However, the operations at the next level do not conflict and are not related by the serialisation order. Therefore, their weak ordering could now be reversed. Finally, Figure 4(C) shows the effect of indirect conflicts. Transaction t_3 forces a serialisation order between t_1 and t_2, even though no direct conflict exists. This serialisation order must match the weak input order. But similarly to the previous case, the next level finds that there are no conflicts among the children. Therefore, no serialisation order is imposed, and the execution at that level can be parallel or even reversed.

This applies to both conflicting operations, as well as intra-transaction parallelism for weakly ordered operations. Figure 2 illustrates the former case. Although the two update operations conflict, they can be executed in parallel, so long as the serialisation orders match the weak input orders, as guaranteed by conflict consistency. In the latter case, the intra-transaction parallelism illustrated in Figure 1 is similarly correct, since here also stack conflict consistency guarantees that the input order is respected.

Intuitively, correctness of stack conflict consistency can be argued as follows. At every level, execution is serialisable since conflict consistency requires the serialisation order to be acyclic. Between adjacent levels, a *caller* passes its execution order as the input to a *callee*. By Definition 2, this must contain all orderings which matter to the caller; that is, transaction orders and orders on conflicting operations. Conflict consistency of the callee ensures that these orderings are respected wherever they matter. That is, the serialisation order of the callee and the execution order of the caller match each other. Between non-adjacent levels, the serialisation orders might contradict one-another. However, this is only possible if an intermediate level has determined that the operations commute. In this case, restrictions imposed from the caller need not be propagated further. The intermediate level assumes responsibility for finding an equivalent execution order that matches that required by the caller.

As in the classical theory, the correctness of a stack schedule depends on the correctness of the conflict relations. It is possible to define conflict relations in such a way that executions are trivially incorrect; for example, *write* and *read* do not conflict. However this implies only that the conflict relations are incorrect, not that scheduling is incorrect. A correct definition of conflicts yields a correct set of executions. Frequently, specific definitions of conflict relations are given based on concepts such as states, arguments, return values, and classes of operations. How conflict tables are determined is orthogonal to scheduling and, here, we assume that conflict tables are correct.

3 Comparison with Existing Models

This section provides a comparison of the correctness criterion stack conflict consistency with two key correctness criteria proposed in the literature [BBG89, Wei91].

3.1 Order-Preserving Serialisability

A comprehensive theoretical treatment of order preservation has been provided by Beeri, Bernstein and Goodman [BSW88, BBG89]. Order preservation is proposed as a sufficient precondition for independent schedulers within subsystems. Briefly stated, order preservation is based on the idea that lower scheduling levels must always preserve the order seen by higher levels.

In practice, order preservation implies that only the strong input order is considered. Moreover, the serialisation order at any level must conform to the strong input order. The two principal differences between Beeri *et al*'s criterion of order-preserving serialisability and SCC are the following. Firstly, operations which are executed sequentially, are serialised in the same order as they are executed. Thus, the strong order affects not only operations with flow of information between them. Second, in [BBG89] intra-transaction order is always a strong order, thereby eliminating the possibility of intra-transaction parallelism for ordered operations. This observation leads to the conclusion that SCC characterizes a larger class of correct executions. In the following paragraphs we formulate this property formally and prove it.

Before discussing order preservation, two preliminary definitions are required.

Definition 8 (Downward Order Compatibility) *A given schedule, S, is* downward order compatible *iff, for all $t, t' \in T$, whenever $t \to t'$, then $\forall o \in O_t,\ \forall o' \in O_{t'}:\ o < o'$.* □

Definition 9 (Upward Order Compatibility) *A schedule S is* upward order compatible *iff, for all $t, t' \in T$, whenever $\forall o \in O_t,\ \forall o' \in O_{t'}:\ o < o'$, then $t \to t'$.* □

Downward order compatibility states that, whenever two transactions are ordered in the input order, their executions are separated and ordered according to the input order. Upward order compatibility is the converse. Whenever two transactions' executions are separated and ordered, they are ordered correspondingly in the input order.

Definition 10 (Order Preservation) *A schedule S is* order preserving *iff:*

$$\forall t,t' \in T: \ t \to t' \Rightarrow t' \not\to t.$$ □

That is, order preservation requires that, whenever two transactions are executed serially, their serialisation order is not reversed.

Definition 11 (Order Preserving Stack Schedule) *An n-level stack schedule, SS, is said to be* order preserving *iff, for $1 \leq i \leq n$, S_i is upward and downward order compatible, and order preserving.* □

Notice that, within a stack, downward order compatibility separates entire transaction trees. In particular, downward order compatibility ensures that an input order is propagated, within a single schedule, from transactions to the execution order on their children. Definition 2 ensures that the execution order at one level is propagated as the input order to the next. Hence, if two transactions are ordered at one non-leaf level, then all their subtransactions are ordered at the next; that is, downward order compatibility forces entire execution trees to be separated and ordered. Figure 2(A) illustrates an order-preserving execution. Figure 2(B) shows the same transactions interleaved, thereby violating downward order propagation.

The advantage of order preservation is that independent schedulers can be used at different levels, as long as each one guarantees order preservation and serialisability independently. This leads to the following correctness criterion [BBG89].

Definition 12 (Order-Preserving Serialisability) *An n-level order-preserving stack schedule is* order-preserving serialisable *(OPSR) iff, for $1 \leq i \leq n$, S_i is conflict serialisable.* □

Order-preserving serialisability asserts that, whenever each schedule in a stack is order-preserving, upward and downward order compatible, and serialisable, the entire execution is serialisable.

We now show that the class of order-preserving serialisable executions is a proper subset of the class of stack conflict consistent executions.

Theorem 2 *OPSR $\subset$ SCC.*

Proof. In a first step, containment is proven by contradiction. Assume there is a stack schedule, SS, which is order preserving serializable but not stack conflict consistent. In order for SS not to be conflict consistent, there must be a level, i, in which the corresponding schedule, S_i, is not conflict consistent. Thus, S_i contains a cycle in $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i} \cup \mapsto_{S_i})$ of the form:

$$t_1 \ (\hookrightarrow_{S_i} \cup \rightarrow_{S_i} \cup \mapsto_{S_i}) \ t_2 \ \cdots \ (\hookrightarrow_{S_i} \cup \rightarrow_{S_i} \cup \mapsto_{S_i}) \ t_n \ (\hookrightarrow_{S_i} \cup \rightarrow_{S_i} \cup \mapsto_{S_i}) \ t_1$$

Furthermore, this cycle cannot contain exclusively relations from $\hookrightarrow_{S_i}$, since S_i is conflict serialisable (given that SS is order preserving serialisable). In addition, the cycle cannot contain exclusively $(\rightarrow_{S_i} \cup \mapsto_{S_i})$ since $(\rightarrow_{S_i} \cup \mapsto_{S_i})$ is acyclic by definition. Therefore, without loss of generality, we assume $t_1 \ (\rightarrow_{S_i} \cup \mapsto_{S_i}) \ t_2$. By strictness of S_i in the input order, all operations of t_1 are ordered before all operations of t_2 in $<_{S_i}$. But for each relationship $t \ (\hookrightarrow_{S_i} \cup \rightarrow_{S_i} \cup \mapsto_{S_i}) \ t'$, at least one operation of t must be before at least one operation of t' in $<_{S_i}$, either by the definition of the serialisation graph in the case $t \hookrightarrow_{S_i} t'$, or by strictness of the input order in the case $t \ (\rightarrow_{S_i} \cup \mapsto_{S_i}) \ t'$. Hence, transitively, at least one operation of t_2 must be before at least one operation of t_1 in $<_{S_i}$. This is a contradiction since all operations of t_1 are before all those of t_2, and $<_{S_i}$ is a partial order. Hence, if SS is order preserving serializable, it is also stack conflict consistent.
Proper containment is proven by the the example in Figure 2(B) which is SCC but not OPSR, since SCC does not require downward order compatibility. □

A potentially misleading difference between stack schedules and Beeri *et al*'s model concerns transaction structure. The transaction structures considered by Beeri *et al* form a so-called *computational forest*. Not all transactions need have the same depth. A consequence of Axiom C6 in [BBG89], however, is that no operation can be ordered with respect to any of its ancestors or descendants. Therefore, transactions, subtransactions and operations are in fact organized into levels, where the top level contains all root transactions, the next lowest level contains all operations at distance one from the top level, etc. Stack schedules also model such mixed-depth tree structures. Unlike in Weikum's multilevel model, a subtransaction at some arbitrary level may have no children, hence have no descendants at any lower level. In fact, the admitted transaction structures are the same as the computational forests characterized by Beeri *et al.*

3.2 Level-by-Level Serialisability

Weikum proposes a weaker condition than order preservation for independent scheduling in composite systems. This condition forces conflicting operations at a non-leaf level, to have conflicting descendants at all lower levels (this is Axiom 1 in [Wei91]). This restriction, though frequently natural, restricts the scope of Weikum's model. For example, it does not hold for multi-version concurrency-control algorithms.

In level-by-level serialisability, when two operations conflict (that is, they are ordered by the weak execution order), they must also be strongly ordered. Consequently, the execution trees of conflicting operations cannot be interleaved. As shown in Figure 2, this is not the case for stack conflict consistency.

Level-by-level serialisability is based on a more general criterion *multilevel serialisability*. Unlike stack conflict consistency, multilevel serialisability assumes all transactions in a multilevel schedule are of the same depth. That is, every transaction has operations at the next lower level. Reasoning about correctness is performed 'bottom-up'. An execution order is given for the lowest level only, serialisation orders are derived at all other levels from this original order. As in Beeri *et al*'s model, multilevel serialisability does not differentiate between strong and weak orders. Since stack conflict consistency allows parallel execution of operations from the same transaction although dependencies might exist between them, stack conflict consistency models aspects of transactions which are not considered by multilevel serialisability. In this sense, stack conflict consistency allows a larger class of correct executions, which we show formally in the following. This consideration aside, multilevel serialisability and stack conflict consistency have similar scopes.

A multilevel schedule in Weikum's sense [Wei91] can be expressed using stack schedule notation as follows. Based on the execution order $<_{S_1}$, serialisation orders are derived at all other levels, the so-called *quasi orders*, as follows.

Definition 13 (Quasi Orders) *For an n-level stack schedule, SS, the* quasi orders*, denoted* $\tilde{<}_i$*, for* $0 \leq i \leq n$*, are derived as follows:*

$$t \ \tilde{<}_i \ t' \ \Leftrightarrow \begin{cases} i = 0: & t <_{S_1} t' \ \wedge \ Con_{S_i}(t, t') \\ 0 < i \leq n: & t, t' \in T_{S_i} \wedge \exists o \in O_t, \exists o' \in O_{t'}. \\ & Con(o, o') \wedge o \ \tilde{<}_{i-1} \ o' \end{cases}$$

□

Notice that the lowest level execution ordering defined by Weikum ($<_0$ at L_0) corresponds to the $<_{S_1}$ execution order within a stack schedule. Based on the quasi ordering, the correctness criterion *level-by-level serialisability* is defined as follows.

Definition 14 (Level-by-Level Serialisability (LLSR)) *An n-level stack schedule is* level-by-level serialisable *iff:*

1. *All non-leaf operations have descendants at all lower levels;*
2. *Weikum's Axiom 1 holds;*
3. *For* $1 \leq i \leq n$*,* S_i *is downward order compatible; and,*
4. *For* $1 \leq i \leq n$*, the quasi-ordering* $\tilde{<}_i$ *is acyclic.* □

We now show that the class of level-by-level serialisable executions as defined by Weikum [Wei91] is a proper subset of the class of stack conflict consistent schedules.

Theorem 3 *LLSR* $\subset$ *SCC.*

Proof. The containment is first proven by construction. Given a LLSR schedule, we construct an equivalent stack schedule which is in SCC. Equivalence here means that schedules have the same transaction structure, conflict relation and execution ordering of leaf operations. Let SS be an LLSR schedule. SS', an equivalent stack schedule, is constructed as follows. $<_{S_i}$ must be constructed for $1 < i \leq n$, and $\rightarrow_{S_i}$ for $1 \leq i \leq n$ such that all schedules involved are conflict consistent, and the resulting stack is well formed. This is achieved with $\rightarrow_{S_i} = \tilde{<}_i$ for $i \in \{1, \ldots, n\}$, and $<_{S_i} = \tilde{<}_i$ for $i \in \{2, \ldots, n\}$. Firstly, observe that SS' is well formed. Since SS is level-by-level serialisable, its quasi orderings are all acyclic. Further, each pair of conflicting operations is ordered by each execution ordering as a consequence of Weikum's Axiom 1 and Definition 13. Therefore, SS' is well formed. Finally, each individual S_i is conflict consistent, for $1 \leq i \leq n$, since, $\hookrightarrow_{S_i} = \tilde{<}_i$ and $\rightarrow_{S_i} = \tilde{<}_i$, and $\tilde{<}_i$ are acyclic, hence $(\hookrightarrow_{S_i} \cup \rightarrow_{S_i})$ are acyclic. From here, it follows that every LLSR schedulke is also SCC.
Strict containment follows from the example in Figure 2(B) which is SCC but not LLSR. □

3.3 Implementation Aspects

The key dependency between levels in a stack schedule is the weak input order. The advantage of order-preserving [BBG89] and level-by-level [Wei91] serialisability is that they provide sufficiency conditions for independent scheduling at different levels. The input ordering, however, would appear to require information to be exchanged between levels, thereby contradicting independence.

Ideally, a scheduler would accept the input ordering as an additional parameter. Unfortunately, current systems do not provide such functionality; rather this functionality should be placed on the 'wish-list' for future systems. There are some techniques, mainly developed in the area of multi-databases, for influencing the serialisation order of autonomous subsystems, e.g., commit-order-serialisability [AGMS87, BGRS91, Raz92, Pu88, PLC81, DE89, WV90, GR91]. Any of these approaches can be used to implement versions of the weak input order, although they may result in further restrictions in concurrency.

4 Applications

Stack conflict consistency provides a larger class of correct executions, and introduces the possibility for parallelism which was not possible in previous models. We expect it to be of use in applications where multilevel transactions are currently being used.

One of the most interesting aspects of stack conflict consistency is the possibility of executing operations in parallel, even if they conflict. This raises the possibility of implementing a *parallel-do* operation. By parallel-do we mean that the operations specified are executed in parallel, while preserving any serialisation dependencies between them. In the example shown in Figure 1, the programmer will specify something like:

Begin_Parallel_Do
  Sal-Bonus() → *GetNewAverageSal();*
End_Parallel_Do

thereby indicating that they should be executed in parallel while preserving the indicated ordering. In full generality, stack conflict consistency provides a theoretical foundation for a parallelization operation which takes the weak input order as an input, and generates a parallel execution for which the serialisation order matches the order provided.

4.1 Objects and Documents

As a first example of possible applications of the proposed model, consider an object-relational database. These systems consist of an object oriented model implemented on top of a relational database and are, therefore, natural composite systems. Following this idea, Rys *et al* have shown how intra-transaction parallelism at the object level can be transformed into inter-transaction parallelism at the relational level, thereby enhancing the parallelism of update operations [RNS96]. For performance reasons, replication is used to store the information related to the objects. Thus, individual object operations correspond to complex sequences of operations at the relational level. The advantage of this approach is that these sequences can be predefined for each operation of the object algebra. The disadvantage is that degree of parallelism is hard-wired in the predefined sequences and, because of the high-level conflict detection approach used, tends to be too conservative.

A possible solution to this problem is to use the model proposed in this paper. The specifications translating an object algebra operation into a sequence of relational operations could use weak and strong orders to establish the dependencies between the different components of the sequence. The degree of parallelism will be then determined by the scheduler which can do a more accurate job in assessing possible interferences between transactions.

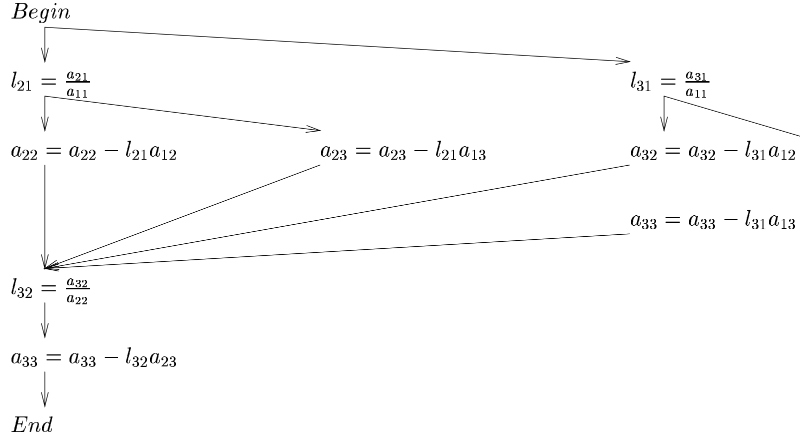

Figure 5: Parallelization of a nested loop

Note that the system could now exploit not only the advantages of transforming intra-transaction parallelism into inter-transaction parallelism but also the possibility of executing in parallel conflicting operations within the same sequence, thereby enhancing the overall system performance.

As another example of a composite systems in which these ideas could be used, in [KS96] a document management system is described in which a TP-monitor is used as the upper level in charge of managing the concurrent insertion of documents in a textual database acting as the lower level. Insertion is performed based on a well defined sequence of operations involving deriving indexing words, inserting indexing terms and inserting the document itself. There are clear strong order dependencies between these operations and weak order dependencies between the operations of the insertion sequences of two different documents. The proposed model would allow to increase the parallelism by interleaving the insertion of index terms and documents while still preserving correctness.

4.2 Parallelization of Numerical Algorithms

An intriguing possibility is that of applying parallel execution of transactions to non-database applications, such as the parallelization of numerical computations. Conventionally, such computations are either parallelized by the programmer using parallel programming languages [Bra89], or by so called super-compilers, which translate a sequential program into a parallel one [ZC91]. In here, an alternative approach is proposed: to decompose numerical algorithms into 'artificial' transactions and subtransactions which are executed by a scheduler.

As an example of how to do this, consider the first step of the gauss-algorithm for $n = 3$. This requires $A = C_1 A$, $A = C_2 A$, where:

$$A = \begin{pmatrix} a^1 \\ a^2 \\ a^3 \end{pmatrix}$$

$$C_1 = \begin{pmatrix} 1 & 0 & 0 \\ -\frac{a_{21}}{a_{11}} & 1 & 0 \\ -\frac{a_{31}}{a_{11}} & 1 & 0 \end{pmatrix} = \begin{pmatrix} c_1^1 \\ c_1^2 \\ c_1^3 \end{pmatrix}$$

$$C_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{a_{32}}{a_{22}} & 1 \end{pmatrix} = \begin{pmatrix} c_2^1 \\ c_2^2 \\ c_2^3 \end{pmatrix}$$

This can be parallelized using two parallel and one sequential loops:

```
DO j=1, n-1
  INDEPENDENT DO i=j+1, n
    l(i,j) = a(i,j)/a(j,j)
    INDEPENDENT DO k=j, n
      a(i,k) = a(i,k) - l(i,k) * a(j,k)
    END DO
  END DO
END DO
```

The dependency graph for this code fragment (Figure 5) shows that there are unnecessary operation orders. Assuming three processors are available, these unnecessary orderings force a total computation of five steps (one assignment requires one step).

Stack conflict consistency can be used to decompose the semantically-rich, numerical operations into simpler ones which can then be executed in parallel. The idea consists in decomposing a numerical algorithm into steps and considering them as transactions that run concurrently, observing some given external partial order. Each step, in turn, is decomposed into simpler steps that are considered as operations in the database terminology. Operations from different steps that semantically commute can be executed in parallel even though they may conflict at a lower level of abstraction, i.e., even if they access some shared data. The transaction manager at such a lower level will take care of correct shared data access.

In this application of stack conflict consistency the database consists of all the variables used in the algorithm, i.e., there are no variables external to the scheduler. Hence, information is passed from one transaction to the other only via shared database objects and only via conflicting operations of different transactions, which can be modelled using the weak input order. In this case, the particular semantics of the application guarantee that if two transactions are weakly ordered, their children will necessarily conflict.

The computation is defined as a single transaction at the highest level. This transaction is a sequence of invocations,

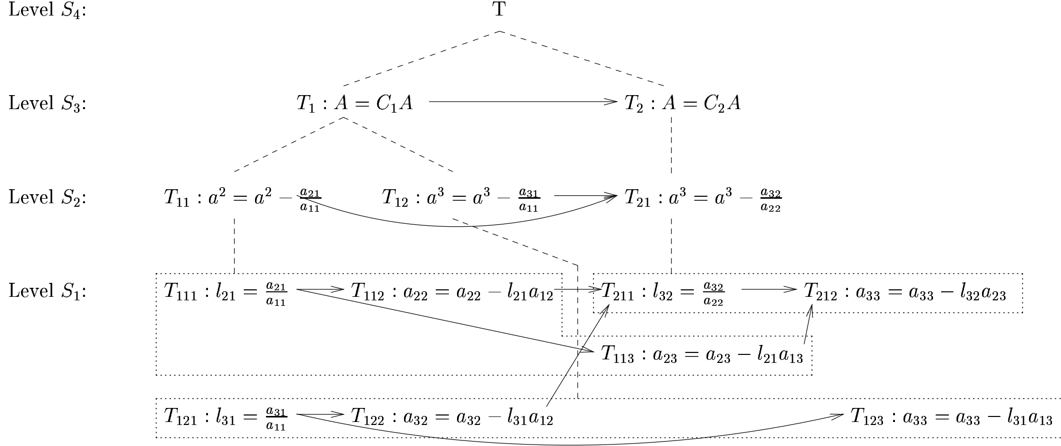

Figure 6: Interleaved execution of ordered steps of a parallel program (without level S_0)

which are, in this case, matrix multiplications. The program order becomes a weak order, allowing the executions of ordered operations to be interleaved.

Now, the first step is to define the operations at each level, and their conflicts. For our example and many other calculations the following set of operations (relative to the different levels) suffice:

- on the lowest level the set of all read and write operations on scalars,
- on the next higher level the set of all assignments of scalars (including all mathematical functions),
- on the next higher level the set of all assignments of scalars and vectors,
- on the next higher level the set of all assignments of scalars, vectors and matrices.

Hence, an operation of the three higher levels is of the form $x = f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ generally can be scalars, vectors or matrices.

Conflicts on the lowest level are as usual. For a criterion on the higher levels, we fist have to introduce the overlap operator $\triangle$:

Definition 15 (Overlapping) *Be x or y, resp., a scalar, vector or matrix.*

$$\begin{aligned}
&x \triangle y \text{ (``}x\text{ overlaps }y\text{'') } :\Longleftrightarrow \\
&x \textit{ scalar } \wedge\ y \textit{ scalar } \wedge\ x = y\ \vee \\
&x \textit{ scalar } \wedge\ x \in y\ \vee \\
&y \textit{ scalar } \wedge\ y \in x\ \vee \\
&x \textit{ no scalar } \wedge\ y \textit{ no scalar } \wedge\ \exists \alpha \in x \mid \alpha \in y
\end{aligned}$$

□

The following criterion for the higher levels can be derived from the conventional definition of a conflict, i.e., commutativity. However, it is given without proof. Notice that the operator "$\cdot$" is restricted to be scalar multiplication.

Definition 16 (Conflict) *Two operations on the same level, $a_1 = (x = f(x_1, \ldots, x_n))$ and $a_2 = (y = f(y_1, \ldots, y_m))$ are* in conflict*, if:*

$$\begin{aligned}
&((x \triangle y)\ \vee \\
&(\exists z \in \{y_1, \ldots, y_m\} \mid x \triangle z)\ \vee \\
&(\exists z \in \{x_1, \ldots, x_n\} \mid y \triangle z)) \\
&\wedge \\
&a_1 \neq a_2 \\
&\wedge \\
&\neg(a_1 = (x = x + expr_1) \wedge a_2 = (x = x + expr_2) \wedge \\
&(\forall z \in expr_1 :\ z \not\triangle x)\ \wedge\ (\forall z \in expr_2 :\ z \not\triangle x)) \\
&\wedge \\
&\neg(a_1 = (x = x \cdot expr_1)\ \wedge\ a_2 = (x = x \cdot expr_2)\ \wedge \\
&(\forall z \in expr_1 :\ z \not\triangle x)\ \wedge\ (\forall z \in expr_2 :\ z \not\triangle x))
\end{aligned}$$

□

The first part of this criterion covers assignments in which the left hand side of one assignment overlaps with the right hand side of the other. The second part excludes from this class of assignments pairs of identical assignments because they commute. The third part excludes pairs of increments in a general form and the fourth part excludes pairs of "incremental multiplication"; they also can be commuted, likewise. There may be other pairs of functions which commute and can be excluded.

In figure 6 an example of this criterion is shown. T_1 and T_2 conflict because in both operations/transactions A is assigned and none of the exceptions holds. T_{11} and T_{12} commute; they only share a_{11} which in both transactions is only read. T_{12} and T_{21}, however, don't commute because, although they seem to be increments on the same variable a^3 (with negative increment item), a^3 contains a_{31} (and a_{31}).

Further, the figure shows that T_{123} can be executed after T_{211}, although the weak input order of their transactions is $T_{12} \to T_{21}$. This type of parallelism does not exist in conventional parallel programs.

5 Conclusion

The background to this work is numerous research projects carried out within our group in both theoretical [Has96, AVA+94, SWY93] and practical aspects of transaction management in composite systems [NWM+94, WH93, DSW94, RNS96, KS96, BRS96]. This previous work convinced us of the need of providing a better theoretical formulation for composite systems to be able to extend their applicability

to environments where a greater degree of parallelism is required. Previous approaches to the problem of composite systems have been very successful in formulating the problem and providing solutions which work very well in conventional databases. When these same ideas are applied in more demanding environments, however, their assumptions become limitations.

To address these limitations, this paper provides a simplified formulation of the theory of correct concurrent executions in composite systems, and a simple correctness criterion: *stack conflict consistency*. The paper's first contribution is to eliminate a number of limitations of existing approaches. In particular, stack conflict consistency is strictly more general with respect to the amount of parallelism possible, when compared with existing models.

Firstly, increased parallelism is achieved by allowing the parallel execution of conflicting operations. Although Weikum's theoretical criterion MLSR allows this, too, stack conflict consistency is implementation-oriented, so it has to be compared with LLSR. LLSR doesn't allow the parallel execution of conflicting operations [Wei91].

Secondly, intra-transaction parallelism is allowed for ordered operations. For this, we have introduced a weak intra-transaction order, and have sketched its application in the parallelization of numerical algorithms. Moreover, the assumptions underlying the new theory are weaker than those underlying either of the models mentioned above.

Future work and open questions include the extension of this work to middleware environments, more precise comparison to multilevel serialisability, development of a better understanding of the circumstances under which the strong ordering can be converted into a weak ordering while preserving correctness, and a more comprehensive study of the application of these ideas outwith database systems, for example to parallel numerical algorithms.

Acknowledgements

We would like to thank Catriel Beeri and Gerhard Weikum for their helpful comments to earlier drafts of this paper.

References

[AGMS87] R. Alonso, H. García-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. *Bulletin of the IEEE Technical Committee on Data Engineering*, 1987.

[AVA+94] G. Alonso, R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H.-J. Schek, and G. Weikum. Unifying concurrency control and recovery of transactions. *Information Systems*, 19(1):101–115, January 1994. An extended abstract of this paper appeared in the *Proceedings of the 4th International Conference on Extending Database Technology*, EDBT'94. March, 1994, pages 123-130.

[BBG89] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman. A model for concurrency in nested transactions systems. *Journal of the Association for Computing Machinery*, 36(2):230–269, April 1989.

[BGRS91] Yuri Breitbart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Abraham Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, September 1991.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[Bra89] Steven Brawer. *Introduction to Parallel Programming.* Academic Press, 1989.

[BRS96] Stephen Blott, Lukas Relly, and Hans-Jörg Schek. An open abstract-object storage system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 330–340, Montreal, Canada, June 1996.

[BSW88] C. Beeri, H-J. Schek, and G. Weikum. Multilevel transaction management: Theoretical art or practical need? In *Proceedings of the International Conference on Extending Database Technology*, number 303 in Lecture Notes in Computer Science, pages 134–154, Venice, Italy, March 1988. Springer-Verlag, New York.

[DE89] W. Du and A. Elmagarmid. Quasi-serializability: A correctness criterion for global concurrency in InterBase. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1989.

[DPS83] P. Dadam, P. Pistor, and Hans-J. Schek. A Predicate Oriented Locking Approach for Integrated Information Systems. In *Proceedings of the IFIP Congress 1983*, Paris, September 1983.

[DSW94] A. Deacon, H.J. Schek, and G. Weikum. Semantics-based multilevel transaction management in federated systems. In *Proceedings of the 10th International Conference of Data Engineering, Houston, Texas, USA*, February 1994.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Data Management Systems. Morgan Kaufmann Publishers, San Mateo, California, USA, 1992.

[GMB+81] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the System R database manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[GR91] D. Georgakopoulos and M. Rusinkiewicz. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the International Conference on Data Engineering*, Kobe, Japan, 1991.

[Has96] H. Hasse. Einheitliche Theorie für korrekte parallele und fehlertolerante Ausführung von Datenbanktransaktionen. PhD. Thesis 11569, ETH Zürich, Department Informatik, 1996. In German.

[KS96] Helmut Kaufmann and Hans-Jörg Schek. Extending TP-Monitors for intra-transaction parallelism. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, USA, December 1996.

[Lom92] David Lomet. MLR: A recovery method for multi-level systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 183–194, 1992.

[MGG85] J. Elliot B. Moss, Nancy D. Griffith, and Marc H. Graham. Abstraction in concurrency control and recovery management. COINS Technical Report 85-51, Department of Computer and Information Science, University of Massachusetts (Amherst), Massachusetts, 01003, USA, December 1985.

[MGG86] J. Elliot B. Moss, Nancy D. Griffith, and Marc H. Graham. Abstraction in recovery management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 72–83, 1986.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[NWM+94] M. C. Norrie, M. Wunderli, R. Montau, U. Leonhardt, W. Schaad, and H-J. Schek. Coordination approaches for CIM. In *Proceedings of the European Workshop on Integrated Manufacturing Systems Engineering*, pages 223–232, Grenoble, France, December 1994.

[PLC81] C. Pu, A. Leff, and S.F. Chen. Heterogeneous and autonomous transaction processing. *IEEE Computer*, 24(12), December 1981.

[Pu88] C. Pu. Superdatabases for composition of hetrogeneous databases. In *Proceedings of the International Conference on Data Engineering*, 1988.

[Raz92] Y. Raz. The principle of commitment ordering – or – guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 292–312, Vancouver, Canada, August 1992.

[RNS96] Michael Rys, Moira Norrie, and Hans-Jörg Schek. Intra-transaction parallelism in the mapping of an object model to a relational multiprocessor system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Bombay, India, 1996.

[SSW95] Werner Schaad, Hans-Jörg Schek, and Gerhard Weikum. Implementation and performance of multi-level transaction management in a multdatabase environment. In *Proceedings of the Workshop Research Issues in Data Engineering–Distributed Object Management (RIDE-DOM95)*, Taipei, Taiwan, March 1995.

[SWY93] H.-J. Schek, G. Weikum, and H. Ye. Towards a unified theory of concurrency control and recovery. In *Proceedings of the ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 300–311, June 1993.

[VYBS95] Radek Vingralek, Haiyan Ye, Yuri Breitbart, and Hans-Jörg Schek. Unified transaction model for semantically rich operations. In *Proceedings of the International Conference on Database Theory (ICDT)*, Prague, Czech Republic, January 1995.

[Wei91] Gerhard Weikum. Principles and realisation strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[WH93] Gerhard Weikum and Christof Hasse. Multilevel transaction management for complex objects: Implementation, performance, parallelism. *The VLDB Journal*, 2(4), 1993.

[WS92] Gerhard Weikum and Hans-Jörg Schek. Concepts and applications of multilevel transactions and open nested transactions. Chapter 13 in [Elm92], 1992.

[WV90] A. Wolski and J. Veijalainen. 2PC agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase. In *Proceedings PARBASE-90*, February 1990.

[ZC91] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, 1991.