In: The Computer Journal, Vol. 40, No. 5, 1997, pp. 245-258.

Transaction Identifiers in Nested Transactions: Implementation Schemes and Performance

Fernando de Ferreira Rezende

Daimler-Benz AG - Research & Technology - Dept. CIM-Research (F3P)
P.O.Box 2360 - 89013 Ulm - Germany
Phone: ++49 (0731) 505 2840 - Fax: ++49 (0731) 505 4210
E-Mail: fernando.rezende@dbag.ulm.DaimlerBenz.COM

Theo Härder and Jan Zielinski

University of Kaiserslautern - Dept. of Computer Science
P.O.Box 3049 - 67653 Kaiserslautern - Germany

Abstract *- In this paper, we address a specific topic inside the context of nested transaction implementation, namely, the assignment of identifiers to transactions. We discuss the most important information such identifiers should carry. We do this based on an analysis of the main requirements the components of a general transaction processing system pose on the identifiers. Thereafter, we present some schemes for the assignment of transaction identifiers and discuss their pros and cons with regard to the requirements presented. Finally, we compare one of our schemes to a conventional one by considering the most common operations that are performed with the identifiers. At last, we show the performance measurements we have obtained.*

Index Terms *- Nested transactions, transaction identifiers, transaction management, performance.*

1. Introduction

When executing more complex transactions, it turns out that single-level transactions do not achieve optimal flexibility and performance. As a solution, the concept of nested transactions was popularized by Moss [1], where single-level transactions are enriched by an inner control structure.[1] Such a mechanism allows for the dynamic decomposition of a transaction into a hierarchy of subtransactions, leading to several well-known advantages in a computing system such as intra-transaction parallelism, intra-transaction recovery control (modular rollbacks), explicit control structure, system modularity, distribution of implementation, etc. Despite the fact that nested transactions are considered a fundamental paradigm for complex applications with e.g. long lived transactions such as CADs, practically no commercial system makes indeed use of nested transactions. One of the main reasons for this is certainly the absence of efficient implementations.

A particular problem inside the field of nested transaction implementation is the assignment of identifiers to transactions. Among alternatives to cope with this problem, the simplest one is to manage a counter and to provide identifiers on the basis of this counter. Additionally, in order to maintain information about the internal structural organization of the transactions, data structures like trees and hash tables are employed in such an alternative.

In this paper we present enhanced encoding schemes for the assignment of identifiers in nested transactions. The distinguishing feature of our schemes is that the identifiers themselves carry the information about the internal hierarchical organization of transactions. Thus, data structures like trees and hash tables are not necessary in our schemes. This feature leads to an efficient solution since it enables our schemes to obtain optimal processing times when manipulating the identifiers, especially during the navigation through transaction hierarchies. Furthermore, our schemes

1. The ideas underlying the concept of nested transactions stem from Davies's *spheres of control* [2, 3].

perform considerably well considering memory resources; notwithstanding, they consume more memory space than the traditional ones as soon as the transaction hierarchies become too deep. However, this aspect can be considered of minor importance since the cost of memory storage is decreasing more and more as times go by and is by far less critical than the time to access information. Finally, by discussing an efficient implementation of an important aspect of nested transaction systems, namely identifier representations, this paper can be viewed as a contribution to the propagation of nested transactions in commercial systems.

This paper is organized as follows. We firstly present a general model of nested transactions (Sect. 2). Thereafter, we analyze the different requirements the components of a transaction processing system pose on the identifiers, and by this way, we build up the main features the identifiers should possess (Sect. 3). We then start presenting our schemes for assigning identifiers to transactions (Sect. 4). We compare one of our schemes to a method commonly used in many systems and we show some performance measurements which confirm the efficiency and benefits of our proposals (Sect. 5). At last, we comment on the assignment of identifiers in some systems (Sect. 6), and finally we conclude this paper (Sect. 7).

2. A Model of Nested Transactions

We basically follow Moss's model and terminology [1], where a transaction may contain any number of *subtransactions*, which again may be composed of any number of subtransactions - conceivably resulting in an arbitrarily deep hierarchy of nested transactions. The root transaction which is not enclosed in any transaction is called the *top-level transaction* (TL-transaction). Transactions having subtransactions are called *parents*, and their subtransactions are their *children*. We also speak of *ancestors* and *descendants*. The ancestor (descendant) relation is the reflexive transitive closure of the parent (child) relation. We use the term *superior* (*inferior*) for the non-reflexive version of the ancestor (descendant). The set of descendants of a transaction together with their parent/child relationships is called the transaction's *hierarchy*. In the following, unless otherwise noted, we use the term *transaction* to denote both TL-transactions and subtransactions.

The hierarchy of a TL-transaction can be represented by a so-called *transaction tree*. The nodes of the tree represent transactions and the edges illustrate the parent/child relationships between the related transactions. In the example of a transaction tree shown in Fig. 1, the root is represented by TL-transaction **A**. The children of subtransaction **C** are **D** and **F**, and the parent of **C** is **B**. The inferiors of **C** are **D**, **E**, **F**, and **G**, and the superiors are **B** and **A**. In turn, the descendants and ancestors sets of **C** additionally contain **C** itself. The hierarchy of **C** is depicted as the subtree spanned by **C**'s descendants.

3. Identifier Features

In this section, we build up the main features which the storage structure for the transaction identifiers (TRIDs, for short) should possess by considering the different requirements that the components of a general transaction processing system pose on these identifiers.

3.1 Transaction Manager

When a subtransaction commits, the transaction manager has many tasks to do, e.g., to guide the lock manager, to inform the log and recovery managers, etc. For most of its tasks, the identifier of the parent transaction must be known. Thus, it would be useful for the transaction manager if it could, from the child TRID, immediately identify its parent. Additionally, the transaction manager is also responsible for creating transactions. On the one hand, it should be able to create as many

subtransactions as necessary. On the other hand, there should not be a critical upper limit to the assignment of TRIDs. Therefore, for the transaction manager, it is important that:

Req. 1: A TRID should allow the immediate recognition of its parent.

Req. 2: TRIDs in deep as well as broad transaction hierarchies should be adequately supported.

Req. 3: The TRID storage structure should accommodate as many identifiers as necessary.

3.2 Recovery Manager

When a subtransaction commits, the recovery manager must chain the log records written for the committing subtransaction to the ones of its parent [4, 5]. The main purpose of this log chaining is to rightly guide the recovery manager in the abort process of a transaction. In the nested transaction model, when a transaction aborts, all its inferiors must be also rolled back, independently of whether they are still active or have already 'committed'. Hence, on the basis of this log chaining, the recovery manager realizes through a special *subtransaction commit log record* that at that point in the transaction log, a subtransaction of the aborting transaction has committed. By this means, the recovery manager has enough hints to start rolling back the committed subtransaction of the aborting parent transaction. Therefore, to the end of easily chaining the log records, a TRID should carry the identifiers of its superiors, allowing by this way for the recognition of a parent transaction at any level of the hierarchy. Thus, the recovery manager's requirement equals the transaction manager's first requirement stated before.

3.3 Lock Manager

The model of nested transactions we assume makes maximum parallelism in a transaction hierarchy possible, allowing for parent/child as well as sibling parallelism (as is the case in Camelot [6, 7], Clouds [8, 9], Eden [10, 11], LOCUS [12, 13], KRISYS [4, 14], etc.).[2] Hence, a distinction is made between the locks explicitly acquired by a transaction and those acquired by inferiors and then passed on to their parents at commit time - referred to as *held* and *retained* locks [15], respectively.[3] Therefore, when comparing locks' compatibilities, the lock manager handles retained and held locks. The comparison is very simple if the requested lock is compared with a *held* lock: if they are incompatible, the requested lock cannot be granted, and that is all. However, it is made more difficult in the case of comparing requested locks with *retained* locks: if they are incompatible, the lock manager must go ahead and check whether the requesting transaction is a descendant of the one retaining the lock. If so, the lock can be granted, otherwise it cannot. In turn, this check could be made efficiently if the lock manager could immediately extract this information from both TRIDs. This point builds the lock manager's requirement:

Req. 4: The check whether a transaction is an inferior of another one should be made on the basis of the identifiers themselves.

3.4 Deadlock Manager

For considering the requirements of the deadlock manager, we assume that an extension of the basic approach for deadlock detection in nested transactions is followed. The basic approach [1] allows to identify *direct-wait* and *ancestor-descendant deadlocks*. In turn, extensions of this approach [15, 16] maintain further information to detect *opening-up* (*future*) *deadlocks* as early as

2. Moss's nested transaction model [1] is based on the assumption that only leaf transactions acquire and use locks, i.e., it prohibits parent/child parallelism.
3. As can be noticed, the assumption made in this paper is that the transaction system uses locking to detect conflicts and to guarantee the serializability of transactions. For non-locking systems this requirement would not apply.

possible. Additionally, *detection arcs* [17] can be employed to allow for very efficient deadlock detections. In these extended approaches, the deadlock manager copes with different kinds of waiting relationships. All these *waits-for* relations are represented in a *waits-for-graph* [18], where cycles are looked for.

The first waits-for relation expresses that the lock requester is directly waiting for the lock holder, and hence the inclusion of an edge in the waits-for-graph representing the waiting relationship between both transactions (nodes in this graph) can be performed easily [18]. The second waits-for relation reflects the transaction hierarchy itself and means that a parent transaction waits for the commit of its children. To derive such a waiting relationship is a simple task as well [1]. The third waits-for relation represents a waiting situation between the lock requester and the highest ancestor of the lock holder (retainer) that is not an ancestor of the lock requester (i.e., the *highest non-common ancestor* between both). Finally, the *detection arcs* represent a higher-level abstraction of the other waiting relations.[4] Representing these *enhanced* waiting relationships may save a lot of useless work, since they allow for early as well as very efficient deadlock detection.

In order to derive the latter two relations, the deadlock manager must determine the highest non-common ancestor between the transactions involved in a waiting situation. However, it may be costly to find out who such a non-common ancestor is [4]. For this purpose, both hierarchies must be transitively upward traversed and compared. However, this task would be facilitated if the deadlock manager could catch this information by just comparing TRIDs. This is the deadlock manager's requirement:

Req. 5: It should be possible to identify the highest non-common ancestor between two transactions through their identifiers.

3.5 Cache Manager

From this manager's point of view, the storage structures for the identifiers should be flexible enough to store short as well as long identifiers. In fact, static structures are inappropriate or even impossible to use, if breadth and depth of the transaction hierarchy is not known in advance. Hence, we arrive at the following cache manager's requirement:

Req. 6: The TRID storage structure should be of variable length and flexible enough to optimize the memory utilization and to efficiently store short as well as long TRIDs.

4. Assigning Transaction Identifiers in Nested Transactions

In this section, we present, in an evolutionary way, some schemes for the assignment of TRIDs in nested transactions. We keep in mind the fulfillment by the schemes of those requirements previously enumerated. After presenting each main scheme, we analyze its pros and cons with regard to the requirements.

4.1 The Elementary Scheme

The most elementary scheme, normally used as an illustrative example in the literature [1], is the one presented in Fig. 1. In this approach, the TRID is represented by a variable length vector of integers. Such a vector is composed of one element at the top level and incremented by one more element at each forthcoming level. Hence, every time a subtransaction is created, it receives the complete TRID of its parent and one more element which distinguishes it from the other children of its parent.

4. The reader if referred to [17, 4] for more details on all these waiting relations.

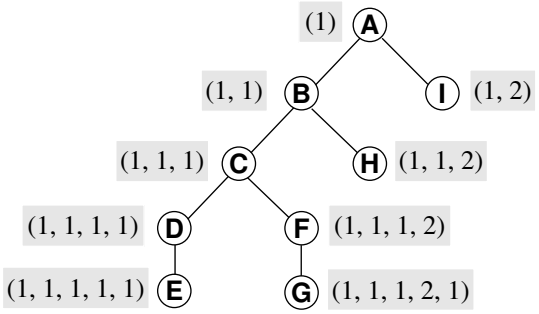

Figure 1: A transaction tree and TRIDs in the elementary scheme.

This is a nice and easily understandable scheme which even fulfils some of the requirements we have pointed out previously: a TRID allows for the recognition of its parent TRID (Req. 1); deep as well as broad transaction hierarchies are relatively well supported (Req. 2); a TRID reflects the execution history of transactions (Req. 4); and finally, the highest non-common ancestor between two transactions is recognizable from their TRIDs (Req. 5). However, the worst point of this approach is its memory overhead (Req. 3, 6). Assuming four bytes long integers, we need $4 \times N$ bytes to identify any transaction at level N. In addition, it does not matter if such a transaction is the 1st or the 2^{32}th child of its parent (at level N - 1), the same $4 \times N$ bytes are allocated to identify it. Notwithstanding, this scheme has some important properties. The other schemes we present are based on this one.

4.2 The EG Scheme

In order to be more precise, we need to scale down a factor and deal no longer with bytes, but with bits. In this section, we present a scheme where the number of representable TRIDs exponentially grows according to the number of bits allocated - the EG scheme (*exponential growth of transaction identifiers*).

Like before, in the EG scheme a TRID is going to carry the TRIDs of the superior transactions. Hence, a TRID is divided into several units, each one representing a level in the transaction hierarchy. We represent this *level unit* through an *encoding sequence*, which in turn is composed of several *encoding units*. The encoding units have a predefined length in bits and therefore can represent a predefined number of TRIDs. Every time the superior limit of an encoding unit is reached, another one is allocated, and the assignment of new TRIDs may proceed until this second encoding unit is also full; then a third one is allocated, and so forth. In turn, to keep track of how many encoding units build an encoding sequence, an *encoding unit counter* is needed. Such an encoding unit counter has also a predefined length and should precede the encoding units for readability (Fig. 2). Hence, to determine an encoding sequence one should:

(1) read the encoding unit counter stored in the first m bits, where m = length of the encoding unit counter, and then

(2) read the next (encoding unit counter + 1) $\times$ n bits, where n = encoding unit length.

How many different encoding sequences may be represented by this approach, i.e., how many TRIDs may be built at each level of a hierarchy, can be calculated by means of Equation I.

$$2^{n \times 2^m}$$

where: m = length of encoding unit counter
n = length of encoding unit

Equation I: Maximal number of encoding sequences representable by the EG scheme.

Fig. 2a) shows the body of an encoding sequence, whereas Fig. 2b) illustrates the exponential growth of TRIDs in this scheme. In our illustrations of this scheme, we have chosen 2 bits for the

length of the encoding unit counter (m) and 4 bits for the encoding units (n). However, the definition of these lengths may be arbitrarily made and adjusted according to the necessities of each particular system. In addition, one could differentiate the encoding units and state that there are two encoding unit lengths, one for TL-transactions (longer) and another one for subtransactions (shorter). For the sake of simplicity, we make no distinction in the encoding unit lengths yet (we return to this point in a moment). Finally, each pair (encoding unit counter, encoding units) represents an encoding sequence, i.e., a level in the transaction hierarchy. Fig. 3 presents examples of TRIDs in this scheme.

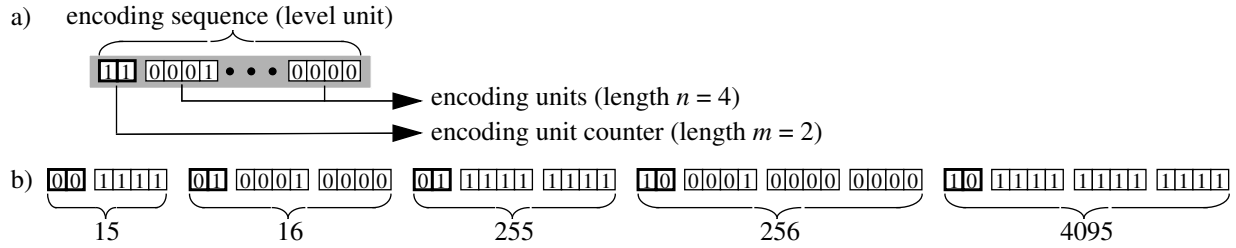

Figure 2: The EG scheme.

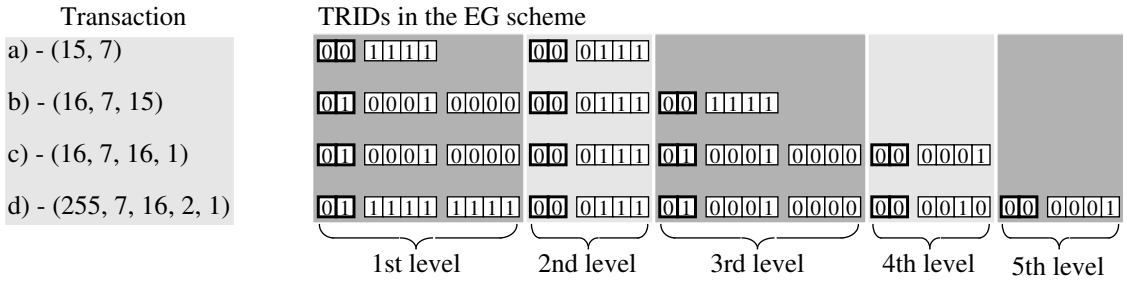

Figure 3: Examples of TRIDs in the EG scheme.

As seen, the encoding unit counter ascertains the length of an encoding sequence on each particular level. However, there can be an arbitrary number of levels in a hierarchy so that one cannot know, at the time of interpreting a TRID, when to stop reading the encoding unit counters and skipping the corresponding number of bits. Therefore, in order to know where a bit stream finishes and to be able to correctly interpret it, some kind of total length information of a TRID is necessary. This information may be stored in a predefined number of bits in the beginning of a bit stream and interpreted as necessary. Usually, one would store this information as an absolute number, simply representing the total number of bits in the bit stream. We have particularly chosen to store this information as a relative number, representing the level the transaction is located on in the hierarchy. The most important advantage of storing the number of levels is that it allows to (more) easily capture the parent TRID (Fig. 4). In addition, for the sake of homogeneity, we are going to represent this number of levels as before, i.e., as an encoding sequence composed of a pair (encoding unit counter, encoding units). Hence, the first encoding sequence of a TRID gives the number of levels in the hierarchy. The EG scheme approaches the satisfaction of all our needs:

Req. 1: It is still possible to recognize the parent of a transaction on the basis of its TRID. As shown in Fig. 4, one must read the first encoding sequence to the end of learning how many levels there are (4). Knowing that the number of levels is 4, one knows that the transaction itself is at the 4th level, and consequently that its parent TRID goes until the 3rd level. One skips the following 3 encoding sequences and has the complete parent TRID at hand. Of course, the number of levels in the beginning of the parent TRID is one less than the one of its child.

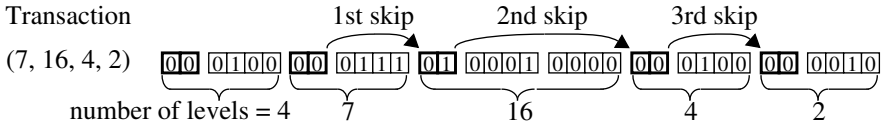

Figure 4: Capturing the parent TRID in the EG scheme.

Req. 2: Deep as well as broad transaction hierarchies are well supported. With the possibility of choosing an adequate number of bits for both encoding unit counter and encoding units, one can tune the EG scheme to the necessities of particular systems using Equation I.

Req. 3: Assuming 4 bits as the length of encoding units, one can create 2^{16} children for each transaction. This shall be sufficient for subtransactions. It may become critical for TL-transactions in systems where TRIDs are not reused. Notwithstanding, this drawback may be eliminated if a distinction in the encoding unit lengths for TL-transactions and subtransactions is made. In the next section, we detail this proceeding, which could be also used here. However, even with the possibility of tuning the lengths of encoding unit counter and encoding units, this scheme always accommodates a potentially very large but finite number of TRIDs.

Req. 4: The execution history of transactions is completely reflected in the TRIDs so that the check whether two transactions are in the same path of a transaction hierarchy can be made on the basis of their TRIDs. To accomplish that, one must only verify whether the longer TRID contains the shorter one.

Req. 5: The highest non-common ancestor between two transactions is recognizable from their TRIDs. On comparing the encoding sequences of both TRIDs until they are no longer equal, one has at hand such a non-common ancestor.

Req. 6: TRIDs have variable length, and one can precisely allocate the number of bits necessary to represent subtransactions at different levels. Hence, the memory space is efficiently used and short as well as long TRIDs are well stored.

In summary, the EG scheme fulfills all our requirements. Its main problem is that it may fail when one tries to create a TRID out of the range supported by the encoding sequences. Although one may try to overcome this problem by adjusting those figures accordingly, it may not be completely eliminated. Before presenting the next encoding scheme, we comment on how this scheme could be expanded in order to try to postpone the occurrence of this problem.

Extending the EG Scheme

Since in the EG scheme the encoding unit counter may get saturated early, we suggest here an expansion in the EG scheme with the representation of minimal extra information, which turns out to be very important when allocating encoding units. We suggest the representation of a counter for the encoding unit counter. Such a counter has also a predefined length in bits (k) and works in the same way as before, i.e., every time the superior limit of an encoding unit counter is reached, another one is allocated, and so forth (see Fig. 5). Hence, to determine an encoding sequence in this extended EG scheme, one should:

(1) read the *counter of encoding unit counter* stored in the first k bits, where k = length of the counter of encoding unit counter,

(2) read the encoding unit counter stored in the next (counter of encoding unit counter + 1) $\times$ m bits, where m = length of the encoding unit counter, and finally

(3) read the next (encoding unit counter + 1) $\times$ n bits, where n = encoding unit length.

Equation II shows how many different encoding sequences may be represented by the extended EG scheme at each level of a transaction hierarchy. With its help, we can perceive the extremely

high representation capacity of this scheme. For example, if we choose 2 bits for the counter of encoding unit counter (k = 2) and keep the same figures for m (= 2) and n (= 4) as before, we can represent 2^{1024} different transactions at each level of a hierarchy. Of course, such a gain on representation capacity means, on the other hand, more processing overhead for interpreting the bit streams and a bit more memory space for the extra counter.

$$2^{n \times 2^{m \times 2^{k}}}$$

where: k = length of counter of encoding unit counter
m = length of encoding unit counter
n = length of encoding unit

Equation II: Maximal number of encoding sequences representable by the extended EG scheme.

We have sketched in Fig. 5a) the body of an encoding sequence in the extended EG scheme. Fig. 5b), in turn, shows the minimal TRID, whereas Fig. 5c) shows the maximal one.

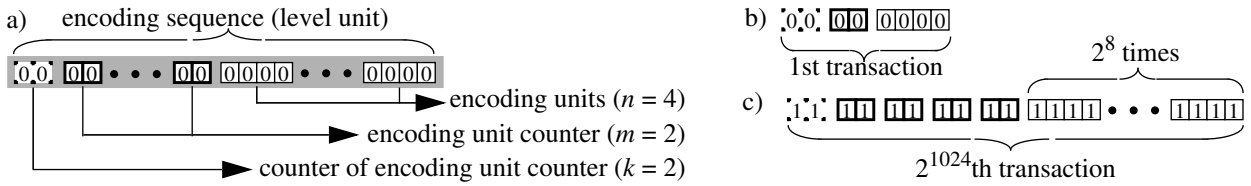

Figure 5: The extended EG scheme.

This extended EG scheme copes well with the problem we mentioned before. However, we have advocated that the size of an encoding unit shall be tuned to each system to the end of rightly accommodating the TRIDs at each level of the transaction hierarchy. Hence, it is first of all pretended and desired that in most cases the transactions should be identified by just one encoding unit. Therefore, in these cases the bits for the encoding unit counter and for its own counter are superfluous. In the following, we present another interesting scheme where we cope with both problems at the same time. We potentially allow an infinite number of TRIDs while avoiding the counters. Additionally, we still keep the good features of these schemes.

4.3 The AG Scheme

The idea underlying the AG scheme (*additive growth of transaction identifiers*) is very simple. We have a certain number of bits (also an encoding unit) for identifying the transactions at each level, which should cover the sub-TRIDs in the average case. When an encoding unit is full, i.e., when all its bits are already used, then another encoding unit is allocated and added to the previous one to proceed with the assignment of TRIDs. If it is full again, another one is allocated, and so forth. The main difference to the EG scheme is that we reserve one special representation of bits in order to signal that an encoding unit is full. Hence, when all bits of an encoding unit are set to 0 (zero, our special *full representation*), then the next forthcoming encoding unit pertains to this same level. This scheme is additive in the sense that, in order to capture a TRID at a level, all encoding units of this level must be added until a non-full representation is found, which then signals the beginning of the next level. Therefore, to determine a TRID, one should:

(1) read the value of encoding unit (say, *value*) stored in n bits, where n = encoding unit length, and

(2) check whether encoding unit is equivalent to the full representation (0). If so, then add to *value* the representation capacity of an encoding unit (refer to Equation III) and return to the previous step.

Equation III shows how many different representations can be produced per encoding unit, i.e., at a level of a transaction hierarchy, by the AG scheme. In turn, an infinite number of TRIDs may be represented, since it may potentially allocate an infinite number of encoding units.

$2^n - 1$ where: n = length of encoding unit

Equation III: Maximal number of representations per encoding unit in the AG scheme.

Fig. 6 illustrates the additive behavior of TRIDs in the AG scheme. In particular and in contrast to the EG scheme, we have chosen 8 bits as the length of encoding units (n). The idea here is the same as before: A single encoding unit should be enough to identify the transactions at each level; if it is not due to an exceptional case, another encoding unit is used. Therefore, the right tuning of the encoding unit length is a very important aspect, which influences the performance of this scheme.

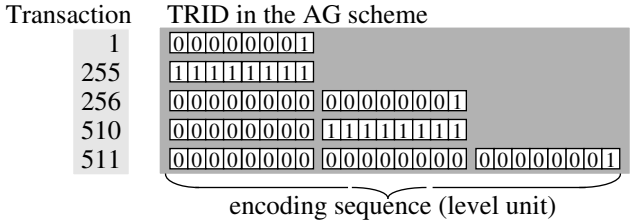

Figure 6: The additive behavior of TRIDs in the AG scheme.

As seen in the EG scheme, the first unit in a TRID is used to store its number of levels, particularly because one needs to know where a TRID finishes. We also need this same information here, and of course due to the very same reason. However, we cannot store it as the number of levels like before. In the EG scheme, the number of levels together with the encoding unit counter provide enough information to learn the length of the whole TRID. But we do not have encoding unit counters in this scheme, and the number of levels alone is not sufficient, since a single level may spread along several encoding units. Therefore, we have decided for this scheme to store this information as the total number of encoding units. In addition, this information will be stored in the same way as the encoding units. Hence, the first encoding sequence in a TRID gives its number of encoding units (Fig. 7). In the following, we analyze this scheme with respect to the requirements:

Req. 1: To recognize the parent TRID is an easy task (Fig. 7). The number of encoding units is read (= 5). Thereafter, one directly skips to where the parent TRID potentially is, i.e., two encoding units before the TRID ends. If this encoding unit is not full, then this is the parent. Otherwise, as illustrated in Fig. 7, one must skip backward until a non-full encoding unit is found.

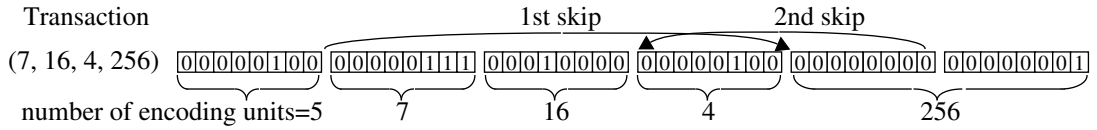

Figure 7: Capturing the parent TRID in the AG scheme.

Req. 2: This approach allows for a great flexibility in supporting deep as well as broad hierarchies. On the one hand, it potentially supports an infinite number of encoding units. On the other hand, the encoding unit length may be tuned so that series of full encoding units can be avoided.

Req. 3: If we assume 8 bits as the size of encoding units, we can create (2^8 - 1) different children for a transaction with a single encoding unit. While being sufficient for subtransactions in many systems, it is certainly not sufficient for TL-transactions. We overcome this problem here by making a distinction in the encoding unit length for TL-transactions and subtransactions (Fig. 8). We can consider, for example, that the encoding unit length for TL-transactions is 4 bytes and that the one for subtransactions is 1 byte. With this distinction, we can store (2^{32} - 1) different TL-transactions in one encoding unit (4 bytes long). When opening the transaction in the limit of the encoding unit storage capacity, a new encoding unit is allocated (more 4 bytes). In order to be able to process this information about the different encoding unit lengths, we need to store it as meta-

information. In addition, one may think of using different lengths not only for TL-transactions but also for subtransactions at different levels of the transaction hierarchy.

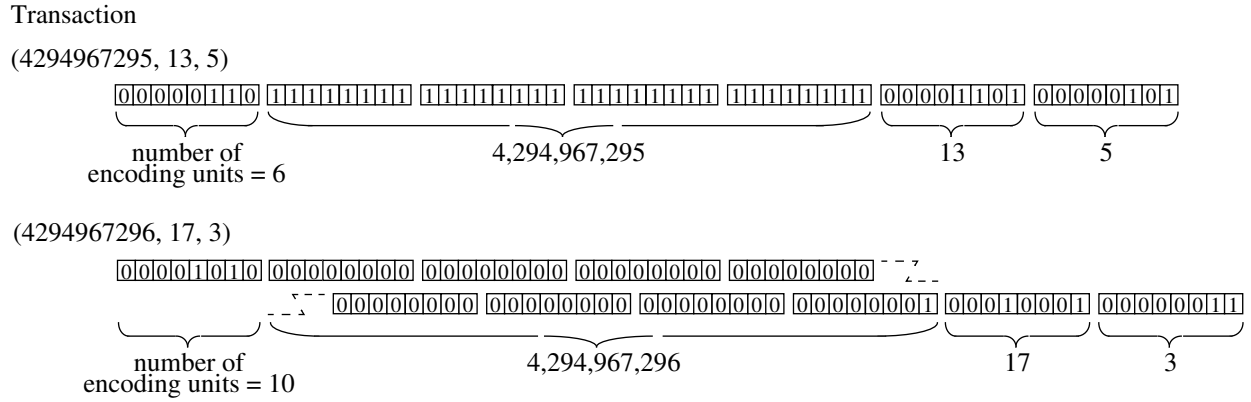

Figure 8: Considering different encoding unit lengths for TL-transactions and subtransactions.

Req. 4: As before, the execution history of transactions is reflected in the TRIDs so that the check about the inferior relationship between two transactions is made only on the basis of their TRIDs (the shorter TRID must be contained in the longer one).

Req. 5: It is possible to recognize the highest non-common ancestor between two transactions. As before, the encoding sequences of the TRIDs are compared until they are unequal.

Req. 6: This is the most important advantage of this scheme: the TRIDs are of variable length, no space is necessary to store encoding unit counters, and the encoding unit lengths can be rightly tuned in order to efficiently represent transactions at different levels.

Summing up, it may be said that the two important points of this scheme are the following: a potentially infinite number of transactions can be identified (of course, by an also infinitely large representation), and no extra bits are necessary for encoding unit counters at each level, since a special (full) representation carries this information. A crucial problem of this scheme is its additive behavior, because for the transactions which do not fall in the normal case, there may be long sequences of full encoding units. In the following, we present a final and interesting encoding scheme, whereby we combine both AG and EG schemes. The idea is to capture the best property of each particular scheme in only one scheme.

4.4 The AEG Scheme

The AEG scheme - *additive and exponential growth of transaction identifiers* - is a combination of the AG scheme with the EG scheme. In essence, we are going to apply the additive growth feature of the AG scheme to the encoding unit counter of the EG scheme. In turn, the encoding units themselves are going to work in the same way as in the EG scheme, i.e., allowing for an exponential growth of TRIDs. Therefore, to interpret an encoding sequence in the AEG scheme requires the following:

(1) read the value of encoding unit counter (say, *value*) stored in m bits, where m = encoding unit counter length,

(2) check whether encoding unit counter is equivalent to the full representation (0). If so, add to *value* the representation capacity of an encoding unit counter (Equation III) and return to the previous step.

(3) read the next $(value + 1) \times n$ bits, where n = encoding unit length.

Equation IV shows how many different encoding sequences can be represented by the AEG scheme at each level of a transaction hierarchy. In turn, like in the AG scheme, an infinite number of TRIDs may be represented, since the occurrences of encoding unit counters can increase accordingly.

$$2^{n \times i \times (2^m - 1)}$$

where: m = length of encoding unit counter
n = length of encoding unit
i = number of occurrences of encoding unit counters

Equation IV: Maximal number of encoding sequences representable by the AEG scheme.

Fig. 9 shows some examples of transactions identified in the AEG scheme (only one level in the hierarchy is shown, but like before the transactions carry the TRIDs of their parents). As can be seen, the 1st transaction is represented by one encoding unit counter and one encoding unit, like in the EG scheme. However, as soon as an encoding unit counter reaches its encoding unit allocation capacity, another one is used to allocate and manage more encoding units.

All in all, the AEG scheme is more flexible than the EG scheme because it is not subject to allocation capacity failures. In turn, it is more powerful than the AG scheme in the sense that it allows for an exponential growth of TRIDs. However, it certainly incurs more processing overhead for interpreting the TRIDs.

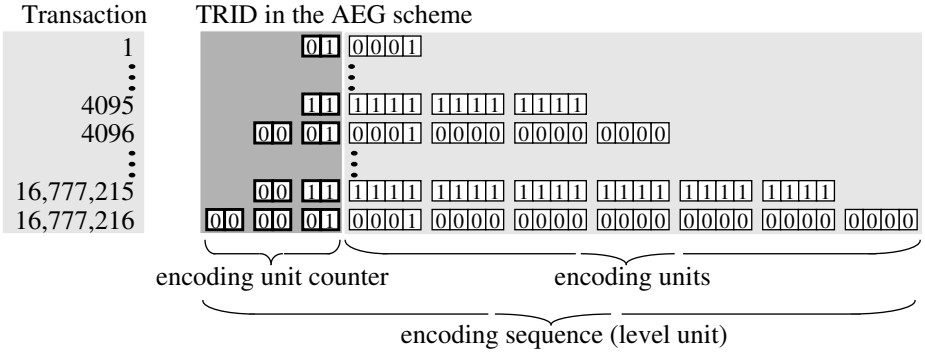

Figure 9: The AEG scheme.

5. Performance Evaluation

We have realized some performance measurements in order to confirm the validity of our enhanced schemes for the assignment of TRIDs in nested transactions, some of which we present in this section. Since an exhaustive evaluation of all our encoded schemes is impossible in the framework of this paper, we have selected the AG scheme for indicative performance results. The EG scheme limits the number of encoding sequences which makes the choice of the parameters (n, m) critical in a practical application. In contrast, the AG and AEG schemes do not embody such restrictions. The AG scheme is simpler to implement and allows faster operations whereas the AEG scheme may be often more economical in storage utilization. Here, we try to illustrate the potential gain of our encoded schemes as compared to a conventional scheme which represents the transaction structure explicitly.

In such a conventional scheme, the TRIDs do not carry the identifiers of their superiors, and hence trees and hash tables are used to find out the hierarchical relationships between transactions. First, we briefly introduce the conventional scheme and its data structures. Thereafter, we present the main results that we have obtained.

5.1 The Conventional Scheme

The conventional scheme we have chosen for comparison is the one implemented in PRIMA [19, 20]. In this scheme, the nested transaction structure is visualized by a set of m-ary trees, where the nodes are transactions and the edges are parent/child relationships. The root of such an m-ary tree corresponds to a TL-transaction. The transactions are represented by transaction control blocks (TCBs), and the edges by pointers between them.

PRIMA's m-ary transaction trees are implemented by a special type of binary trees (Fig. 10). For efficiency reasons, pointers have been added to link the children to their parent. Hence, a TCB contains four pointers (*parent*, *child*, *right sibling*, and *left sibling*) which are used to establish the nested structures. Further, a TCB stores additional information about the transaction, which describes its identification, type (e.g., access system transaction), resources, etc. For the sake of simplicity, we are not going to enter in the details of TCBs, but we just provide the necessary information to understand the nesting of transactions in this scheme.

Fig. 10 illustrates the tree structures which correspond to the transaction tree of Fig. 1. As can be seen, the parent of a transaction can be found by traversing the *parent pointer*. In turn, all children of a transaction can be reached by firstly traversing the *child pointer*, and secondly, from this pointer on one can navigate via the *right sibling pointers* to the other children of the transaction. In particular, the *left sibling pointer* is used for easily removing a transaction from the sibling chain. Hence, in order to reflect the nested structure of the transactions, the conventional scheme explicitly chains the TCBs together, and the TRIDs are uniquely assigned on the basis of a global counter.

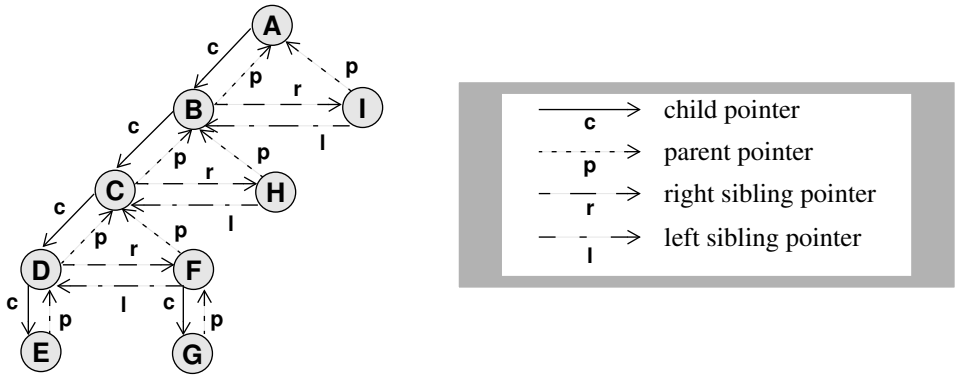

Figure 10: Transaction trees in the conventional scheme.

5.2 Algorithms and Environment

Generally speaking, there are two main differences in the implementation of both schemes. First, whereas our AG scheme works just on the basis of the TRIDs, the conventional scheme is TCB-oriented, and it navigates via the pointers previously presented (Fig. 10) in order to perform the desired functions. Second, our AG scheme mainly follows a top-down strategy in performing the functions, i.e., it starts comparing the bytes from the TL-transaction down to the leaves of the transaction hierarchy. In turn, the conventional scheme uses a bottom-up strategy in the functions, whereby it directly accesses a TCB via a hash table, and from that starting point, it recursively goes up to the superior TCBs via their parent pointers.

We have implemented two main kinds of algorithms for the AG scheme. The first kind of algorithms are of a very general use. They are so programmed that all encoding unit lengths, namely for the first (total length), TL-transaction, and subtransaction encoding sequences, are represented as meta-information (through *defines*). With their help, one can easily tune the charac-

teristics of the TRIDs to any system. However, for performance reasons, it is desirable that one changes these general algorithms according to the specific necessities of a system, so that some of the tasks and checks performed in the algorithms may be facilitated. Thus, we have a special version of these algorithms tailored to the features of our prototype system KRISYS, where the length of encoding units for TL-transactions is four bytes, the one for subtransactions in general is one byte, and the first byte in the TRIDs gives the total length information (just like the way we have presented the AG scheme - refer to Sect. 4.3). This version of the algorithms is much simpler, and therefore it has shown better performance results than the general algorithms. In the following, we present the performance results we have obtained with this specific version of the algorithms.

We have run all these algorithms on a Sun Sparc Station ELC 4/25[5], with 64 Mbytes of main memory capacity, under the operating system SunOS 4.1.4[5], and the windows system Sun-X11R5[5]. In order to get exact time measurements, we have used a kernel module written in Sparc-Assembler, which has allowed us to access the precise Hardware-µsec-Timer of the Sun Workstation. We have performed all the algorithms until the 50th level in a hypothetical transaction hierarchy, where the 1st level corresponds to the TL-transaction. In addition, we have repeated all functions thousand times in order to get good average times that are not so influenced by eventual machine overloads.

5.3 Memory Space Utilization

The first aspect we have compared is the memory space utilization in both approaches (Fig. 11). We have assumed in our comparisons that one encoding unit in the AG scheme is sufficient to store the TRIDs at each level, i.e., there is no sequence of full encoding units. Otherwise, it would be very hard, if not impossible, to draw comparisons. In the conventional scheme, the number of bytes is constant and independent of the number of levels. This scheme always allocates 20 bytes to identify a transaction: 4 bytes for each one of the four pointers plus 4 bytes for the TRID itself. The first level in the AG scheme consumes 5 bytes: 4 bytes for the TL-transaction identifier plus 1 byte for the length information. Subsequently, one more byte is needed for each forthcoming level. As can be seen in Fig. 11, before the 16th level the AG scheme uses less memory in the normal cases than the conventional one. However, after the 16th level the AG scheme consumes always more memory space than the conventional scheme.

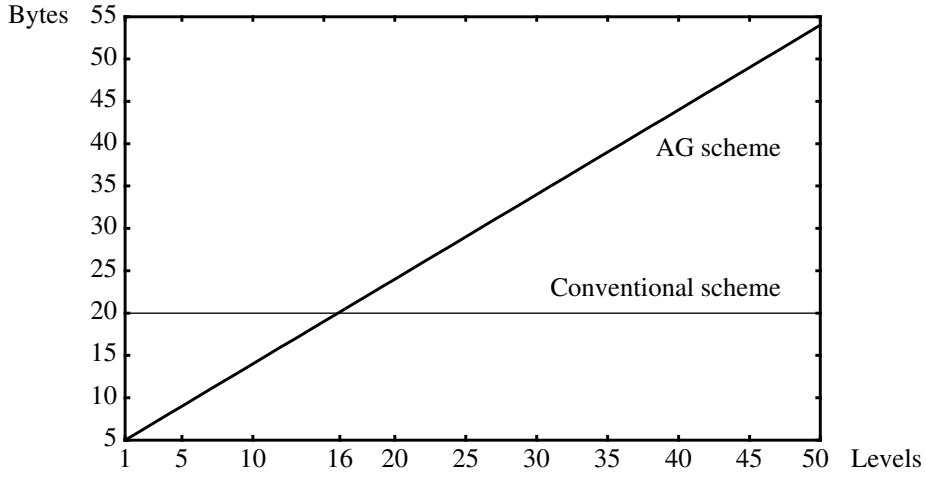

Figure 11: Memory space utilization in the normal case.

5. Registered trademark of Sun Microsystems, Inc.

5.4 Getting the Parent Transaction Identifier

The first procedure we have tested was the one for getting the parent TRID. Whereas in the conventional scheme the TCB of the transaction must be located through the hash table and thereafter traversed via the parent pointer, in the AG scheme the parent TRID is contained in the TRID itself, and thus all that must be done is to simply recompute the total length information. In both approaches, the execution time for getting the parent TRID is more or less independent of the level the transaction is on (Fig. 12).

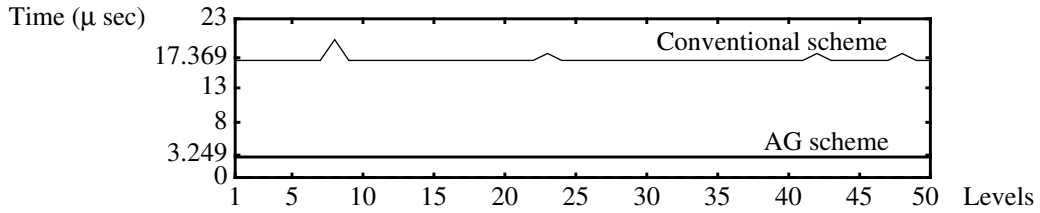

Figure 12: Getting the parent transaction identifier.

5.5 Checking the Inferior Relationship

The function for checking the inferior relationship between two transactions is performed differently in both approaches. The conventional scheme, which uses a bottom-up strategy, varies in terms of execution time according only to the difference of levels between both transactions in the hierarchy; it does not depend on which specific levels these transactions are on. In our measurements (Fig. 13), we have then varied this level difference for the conventional scheme by starting from 1 level difference between both transactions until 50 levels. In turn, the AG scheme is more or less independent of such a level difference between both transactions. On the contrary, it varies according to the specific levels the transactions are on in the hierarchy. This is so because it compares a certain number of bytes of both transactions (the shorter TRID must be inside the longer), which corresponds to the specific levels the transactions are on. Hence, for the AG scheme, we have varied this number of byte comparisons from the 1st to the 50th levels in the transaction hierarchy.

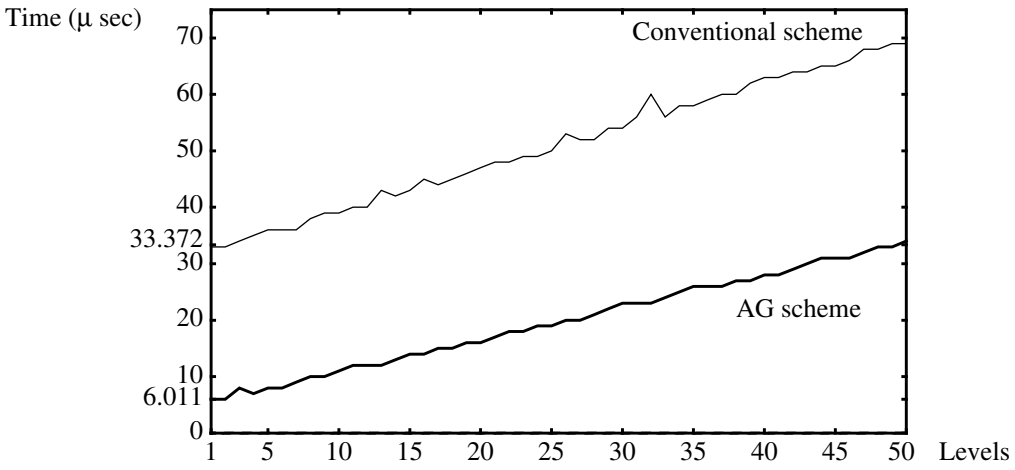

Figure 13: Checking the inferior relationship.

5.6 Getting the Highest Non-Common Ancestor

The function to get the highest non-common ancestor works completely different in both schemes. The AG scheme follows, like before, a top-down strategy in a very simple way. It starts comparing

the bytes of both transactions from the beginning until the first different byte is found; this is then the highest non-common ancestor between both transactions. Hence, its time measurements vary according to the number of bytes examined, i.e., according to the level such an ancestor is on in the hierarchy.

Fig. 14 shows the strategy of the conventional scheme for finding the highest non-common ancestor between two transactions (between **T1** and **T2**; notice that the highest non-common ancestor between **T1** and **T2** is an ancestor of **T2**, but not of **T1**). The TCB of the transaction which is deeper in the hierarchy (in the example **T2**) must be brought to the same level of the other transaction's TCB (Fig. 14a). Thereafter, the parent pointers of both TCBs are navigated upward in the hierarchy until the first common ancestor is found (Fig. 14b). During this navigation, the previous transaction in the path from **T2** to such a first common ancestor must be always remembered. On finding the first common ancestor, the most recently remembered transaction is the highest non-common ancestor between both (**T3** - Fig. 14c). Therefore, this function in the conventional scheme depends on two factors:

- **Factor A:** it depends on how long the path from the deeper transaction until it reaches the same level of the other transaction is (Fig. 14a),
- **Factor B:** from that point on, it also depends on how long the path from both TCBs until the first common ancestor is (Fig. 14b).

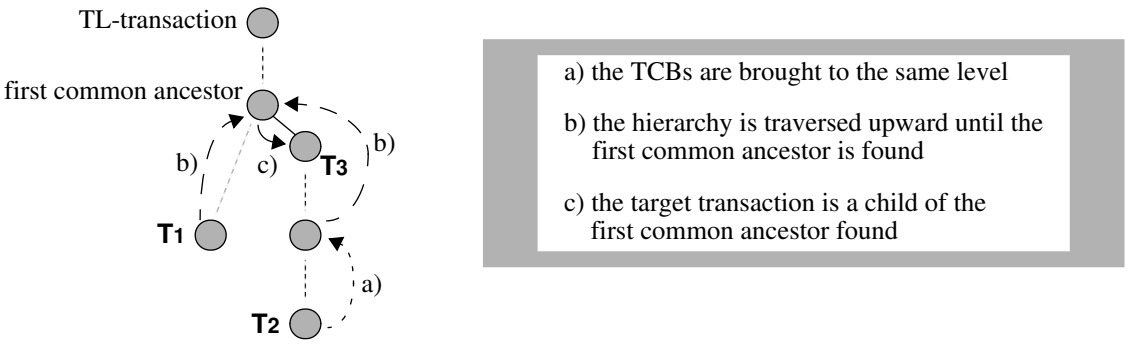

Figure 14: The conventional scheme strategy to get the highest non-common ancestor.

Due to these different influencing factors, we have realized two different measurements for this function. In our first measurement (Fig. 15), we have kept the value of **Factor A** to its minimum (= 0), i.e., both transactions are already at the same level of the hierarchy (in particular, we have put them on the 51st level, since we have varied the levels from 1 to 50 in all our measurements). We have then varied the position such a non-common ancestor is on in the hierarchy, i.e., level 1 in Fig. 15 means that such a non-common ancestor is the TL-transaction itself. In turn, level 50 means that this non-common ancestor is on the 50th level. The graph in Fig. 15 reflects this variation. Whereas the AG scheme is very fast to find a non-common ancestor when this is the TL-transaction (it must just compare the first encoding sequence of both TRIDs for realizing that), the conventional scheme takes a longer time, because it must traverse 50 TCBs upward in the tree until it reaches the TL-transaction. In turn, the conventional scheme gets better performance results as such a non-common ancestor is deeper in the hierarchy (the path it must traverse becomes shorter). On the other side, the AG scheme takes a longer time because it must compare more encoding sequences. Notice that it appears that there is a crossover point a little after the 50th level. We have not determined this crossover point because, as mentioned previously, we have performed all measurements until the 50th level. Furthermore, it is still worth speculating that the graph is probably not linear after this crossover point. If it were linear, the time from the conventional scheme would approach zero which would absolutely not be justifiable. (These same comments apply to the next graph - Fig. 16).

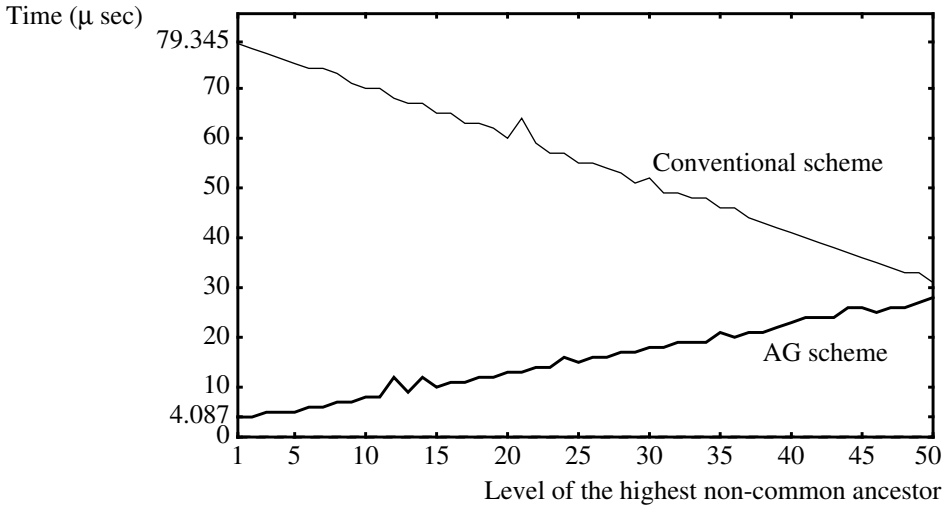

Figure 15: Getting the highest non-common ancestor: **Factor A** = 0.

In our second measurement (Fig. 16), we have kept the value of **Factor A** to its medium value (= 25), i.e., the first transaction is on the middle of the hierarchy, whereas the second is a leaf (in particular, we put the first on the 26th level and the second on the 51st level). In the same way as before, we have varied then the position such a non-common ancestor is on in the hierarchy, from the level 1 until the 25th level. Fig. 16 shows the performance results we have obtained. The AG scheme, as discussed, is independent of such variations, and therefore it has shown practically the same time measurements. The conventional scheme has shown a little bit different time measurements. On the one hand, the path of **Factor A** must be always traversed, on the other hand, the path of **Factor B** has become shorter.

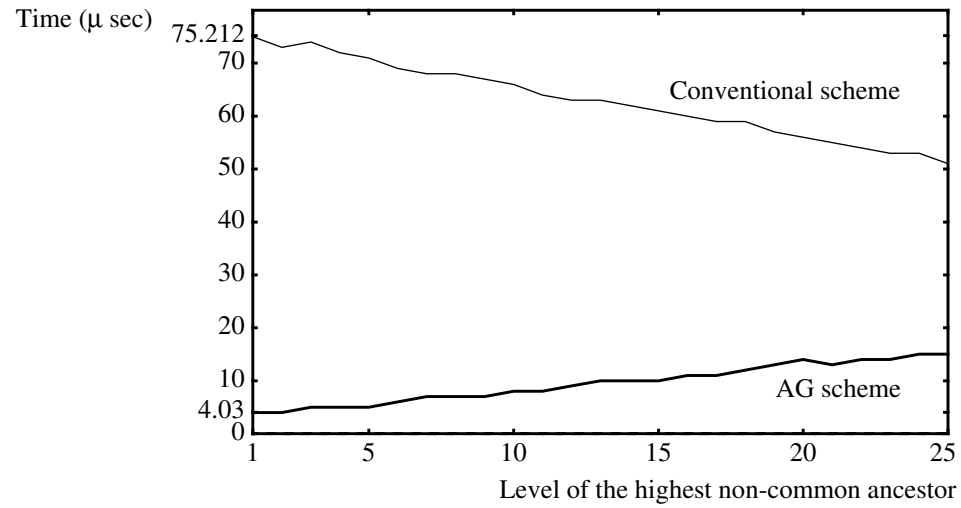

Figure 16: Getting the highest non-common ancestor: **Factor A** = 25.

5.7 Creating a Transaction Identifier

The function for creating new TRIDs works also differently in both schemes. In particular, we have measured the performance for the creation of TRIDs for subtransactions. In the conventional scheme, a subtransaction is always inserted as the most-left children of its parent. Hence, this function involves first of all the creation of a TCB to accommodate the new TRID. Thereafter, in order to reflect the new subtransaction in the trees, many pointers are updated: the parent's child pointer, the left sibling pointer of the old left-most child, and all four pointers of the new subtrans-

action (see Fig. 10). In turn, the creation of a new subtransaction TRID in the AG scheme is a very simple operation. Generally, it involves only an access to a *subtransaction counter* of the parent transaction and the allocation of the respective number of bytes. The results are shown in Fig. 17. The conventional scheme is completely independent of the level the transaction is on, whereas the AG scheme varies a little bit because it must allocate as well as copy more bytes to accommodate the TRID as the level increases.

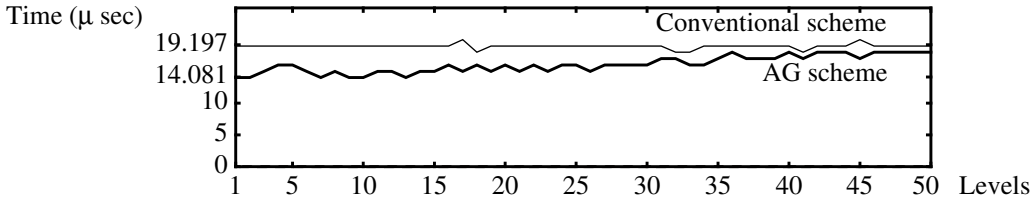

Figure 17: Creating subtransaction identifiers.

6. Related Work

Nested transactions have been implemented in many systems (Argus [21, 22, 23], Camelot [6, 7], Clouds [8, 9], Eden [10, 11], LOCUS [12, 13], KRISYS [4, 14], PRIMA [19], etc.). Unfortunately, very little has been published on the strategies employed by those systems for assigning identifiers (exceptions are [7, 20]).

As we have seen, in PRIMA [19], a conventional scheme for the assignment of TRIDs has been implemented, where trees are used to represent the transaction hierarchies. In this scheme, the subtransaction TRIDs carry the TRID of the TL-transaction, but the transaction hierarchy is not reflected in the TRIDs. For each TL-transaction, there is a tree which represents its hierarchy. In addition, for each transaction tree, there is a hash table allowing for the access to the subtransactions inside this tree. To realize any operation involving TRIDs, the hash tables are accessed and the trees are traversed.

In Camelot [6, 7], the TRIDs have a constant length, which is divided into two parts. The first part is a family (hierarchy) identifier. Each TL-transaction receives a family identifier, which is guaranteed to be unique by a local counter. A subtransaction counter guarantees the uniqueness of subtransaction identifiers inside a family. In turn, the second part of a TRID contains a hint, called *family position indicator* (FPI), which provides information about the position of a subtransaction inside the hierarchy. Such FPIs have a constant length (4 bytes) and register subtransactions until the fourth level in the hierarchies. In addition, a maximum of 256 transactions can be differentiated at each of those 4 levels. Thus, FPIs may contain imprecise information, and therefore, they can be used only as an aid in some special cases. Hence, for example, to check the inferior relationship between two transactions or to get the highest non-common ancestor, their whole hierarchies must be known and their trees must be traversed.

7. Conclusions

In this paper, we have addressed the assignment of transaction identifiers in nested transactions. By appropriately representing the identifiers, some work of the other managers can be saved. We have built up the main features the TRIDs should have, and we have done that on the basis of the different requirements the components of a transaction processing system have on those. In essence, a transaction identifier is represented by a data structure of variable length, and it carries the identifiers of the superior transactions.

We have presented schemes for assigning such TRIDs. The elementary scheme is normally used as an illustrative example in the literature [1], and it wastes memory resources. However, it has some good properties which are kept in the other schemes we have presented. The EG scheme copes well with memory resources; its sole disadvantages are that it supports only a finite number of TRIDs and it must represent extra bits for encoding unit counters. We have then extended the EG scheme in order to allow for more flexibility in the allocation of encoding units, putting its upper limit to an extreme large value with the addition of some more information in the encoding sequences. Thereafter, we have introduced the AG scheme, which copes well with the allocation of encoding units and needs no counters. The AG scheme potentially supports an infinite number of TRIDs, as long as an infinite representation is possible. At the same time, the AG scheme avoids extra bits for encoding unit counters, because it uses a special representation for signalling sequences of encoding units. In particular, with the option of choosing different encoding unit lengths for representing TRIDs on different levels of the transaction hierarchies, this scheme turns out to be very efficient in the matter of memory space utilization. At last, we have presented a combination between the AG and EG schemes, namely the AEG scheme. This scheme, like the AG scheme, is not subject to failures in the allocation of encoding units. Furthermore, it allows, like the EG scheme, an exponential growth of TRIDs.

In addition, we have shown the most important performance measurements of a comparison between the AG scheme and a conventional one. With respect to all kinds of processing, our AG scheme has shown time figures much better than the conventional scheme. However, our AG scheme consumes more memory space than the conventional scheme as the transaction hierarchies become too deep.

We hope with this paper to have covered a topic of nested transactions which has not received much attention thus far. The algorithms we have implemented, in the C language, may be gotten via *anonymous ftp* (131.246.94.94, /pub/informatik/software/rezende/TRIDs_NT). By this way, we hope to facilitate the work of the ones who might like to implement our ideas in their own transaction processing systems.

References

[1] Moss, J.E.B. (1985) *Nested Transactions: An Approach to Reliable Distributed Computing.* M.I.T. Press, USA.

[2] Davies, C.T. (1973) Recovery Semantics for a DB/DC System. In: Proc. of the ACM National Conference, Atlanta, USA.

[3] Davies, C.T. (1978) Data Processing Spheres of Control. IBM Systems Journal, **17** (2), 179-198.

[4] Rezende, F.F. (1997) *Transaction Services for Knowledge Base Management Systems - Modeling Aspects, Architectural Issues, and Realization Techniques*. infix Verlag, Germany.

[5] Rothermel, K., Mohan, C. (1989) ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In: Proc. of the 15th Int. Conf. on Very Large Data Bases, Amsterdam, The Netherlands, 337-346.

[6] Spector, A.Z., Pausch, R.F., Bruell, G. (1988) Camelot: A Flexible, Distributed Transaction Processing System. In: Proc. of the IEEE Spring Computer Conference, USA.

[7] Eppinger, J.L., Mummert, L.B., Spector, A.Z. (1991) *Camelot and Avalon - A Distributed Transaction Facility.* Morgan Kaufmann, USA.

[8] Ahamad, M., Dasgupta, P., Blanc, R.J., Wilkes, C.T. (1987) Fault Tolerant Computing in Object-Based Distributed Systems. In: Proc. of the 6th IEEE Symp. on Reliability in Distributed Software and Database Systems, USA.

[9] Dasgupta, P., Blanc, R.J., Appelbe, W. (1989) The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work. In: Proc. of the 8th IEEE Int. Conf. on Distributed Computing Systems, USA.

[10] Almes, G.T., Black, A.P., Lazowska, E.D., Noe, J.D. (1985) The Eden System: A Technical Review. IEEE Transactions on Software Engineering, **SE-11**, 43-58.

[11] Pu, C., Noe, J.D. (1988) Nested Transactions for General Objects: The Eden Implementation. Technical Report TR-85-12-03, Univ. of Washington, Washington D.C., USA.

[12] Müller, E.T., Moore, J.D., Popek, G.A. (1983) Nested Transaction Mechanism for LOCUS. In: Proc. of the 9th Symp. on Operating Systems Principles, USA, 71-89.

[13] Weinstein, M., Page, T., Livezey, B., Popek, G. (1985) Transactions and Synchronization in a Distributed Operating System. In: Proc. of the 10th Symp. on Operating Systems Principles, USA.

[14] Rezende, F.F., Härder, T. (1995) Concurrency Control in Nested Transaction with Enhanced Lock Modes for KBMSs. In: Proc. of the 6th Int. Conf. on Database and Expert Systems Applications, London, U.K.

[15] Härder, T., Rothermel, K. (1993) Concurrency Control Issues in Nested Transactions. VLDB J., **2** (1).

[16] Rukoz, M. (1991) Hierarchical Deadlock Detection for Nested Transactions. Distributed Computing, **4**.

[17] Rezende, F.F., Härder, T., Gloeckner, A., Lutze, J. (1997) Detection Arcs for Deadlock Management in Nested Transactions and their Performance. In: Proc. of the 15th British National Conference on Databases, London, U.K., 54-68.

[18] Holt, R.C. (1972) Some Deadlock Properties in Computer Systems. ACM Computing Surveys, **4** (3).

[19] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A. (1987) PRIMA - A DBMS Prototype Supporting Engineering Applications. In: Proc. of the Int. Conf. on Very Large Data Bases, 433-442.

[20] Härder, T., Profit, M., Schöning, H. (1992) Supporting Parallelism in Engineering Databases by Nested Transactions. SFB 124 Report No. 34/92, Univ. of Kaiserslautern, Kaiserslautern, Germany.

[21] Liskov, B. (1985) The ARGUS Language and System. In: Paul, M., Siegert, H.J. (eds.), *Distributed Systems - Methods and Tools for Specification: An Advanced Course*, LNCS 190, Springer, 343-430.

[22] Liskov, B. (1988) Distributed Programming in ARGUS. Communications of the ACM, **31**.

[23] Liskov, B., Curtis, D., Johnson, P., Scheifler, R. (1987) Implementation of ARGUS. ACM Operating Systems Review, **21** (5).