In: Proc. of the 16th Brazilian Computer Society Conference and 23rd Integrated Seminar on Software and Hardware (SBC-SEMISH'96), Recife, Brazil, Aug. 1996. pp. 119-130.

Implementing Identifiers for Nested Transactions

Fernando de Ferreira Rezende, Theo Härder, and Jan Zielinski
Department of Computer Science - University of Kaiserslautern
P.O.Box 3049 - 67653 Kaiserslautern - Germany
Phone: +49 (0631) 205 3274 - Fax: +49 (0631) 205 3558
E-Mail: {rezende|haerder}@informatik.uni-kl.de, zille@inf.ufrgs.br

Abstract *- We address a specific topic inside the context of nested transaction concepts, namely, the assignment of identifiers to transactions. We discuss the most important information such identifiers should carry, based on an analysis of the main requirements the components of a general transaction system have on them. Thereafter, we present schemes for the assignment of such identifiers, and discuss their pros and cons w.r.t. those requirements. Finally, we compare one of our schemes to a conventional one, considering the most common operations that are performed with the identifiers, and show some of the performance measurements that we have obtained. Particularly w.r.t. processing time, our scheme has proven to be generally much faster.*

1. Introduction

An important, performance influencing aspect in the implementation of nested transactions (NTs, for short) is the choice of an appropriate strategy to the assignment of transaction identifiers. On one hand, a wrong choice may consume too much memory space in every situation. On the other hand, it may be too costly in terms of processing time when executing operations with the identifiers. Nested transactions have been implemented in many systems (Camelot [16], Clouds [1, 3], Eden [2, 13], LOCUS [12, 17], KRISYS [14], PRIMA [5, 6], etc.). Unfortunately, very little has been published on the strategies employed by those systems for assigning identifiers (exceptions are [4, 7]).

In this paper, we propose 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 traditionally used to maintain such information are not necessary in our schemes. This feature enables our schemes to obtain optimal processing times when manipulating the identifiers, especially during the navigation through transaction hierarchies. Further, our schemes perform reasonably well considering memory resources, however, as all has its price, they consume more memory space than the traditional ones as the transaction hierarchies become too deep.

This paper is organized as follows. In Sect. 2, we present a model of NTs. Thereafter, we build together the main features the identifiers should possess in Sect. 3. We then present our schemes for assigning identifiers in Sect. 4. Following, in Sect. 5 we compare one of our schemes to a traditional one and show some significant performance measurements. At last, we present our conclusions in Sect. 6.

2. A Model of Nested Transactions

We basically follow Moss's model and terminology [11], 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 NTs. 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.

3. Identifier Features

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

3.1 Transaction Manager

When a subtransaction commits, the transaction manager (TM) has many tasks to do, e.g., to guide and to inform the lock and recovery managers, etc. In almost all these tasks, the identifier of the parent transaction must be known. Hence, it would be useful for the TM if it could, from the child TRID, immediately identify its parent. Additionally, the TM is also responsible for creating transactions. On 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. In summary, for the TM, it is important that:
(1) A TRID should allow the immediate recognition of its parent.
(2) Deep as well as broad transaction hierarchies should be well supported.
(3) The TRID storage structure should accommodate as many identifiers as necessary.

3.2 Recovery Manager

When a subtransaction commits, the recovery manager (RM) must chain the log records written for the committing subtransaction to the ones of its parent. The main purpose of this log chain is to rightly guide the RM in the abort process of a transaction. In the NT 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 chain, the RM recognizes 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 the recognition of a parent transaction at any level of the hierarchy. Thus, the RM's requirement equals the first TM's requirement stated above.

3.3 Lock Manager

We assume a model of NTs which enables maximum parallelism in a transaction hierarchy, allowing for parent/child as well as sibling parallelism. Hence, a distinction must be made between the locks explicitly acquired by a transaction and those acquired by inferiors and then passed on to their parents at commit time (lock inheritance). This distinction is usually made by stating that a transaction acquiring a lock on an object *holds* it. In turn, a transaction inheriting the locks of a committing subtransaction *retains* them. If a transaction holds a lock, it has the right to access the locked object. However, a retained lock is only a place holder and indicates that transactions outside the hierarchy of the retainer cannot acquire the lock, but that descendants potentially can. Therefore, when comparing locks' compatibilities, the lock manager (LM) 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 LM 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 LM could immediately extract this information from both TRIDs. This point builds the LM's requirement:

(4) The check whether a transaction is an inferior of another one should be made on the basis of the own identifiers.

3.4 Deadlock Manager

We assume that an extension of the basic approach for deadlock detection in NTs is followed. The basic approach [11] allows to identify *direct-wait* and *ancestor-descendant deadlocks*. In turn, extensions [8, 15] maintain further information to detect *opening-up* (*future*) *deadlocks* as early as possible. In those, the deadlock manager (DM) copes with three different kinds of waiting relations. All these *waits-for* relations are represented in a *waits-for graph* [9], where cycles are looked for. The first waits-for relation expresses that the lock requester is directly waiting for the lock holder. The second waits-for relation reflects the transaction hierarchy itself and means that a parent transaction waits for the commit of its children. 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). Representing this third waiting relation may save a lot of useless work, since it allows for early deadlock detection. However, it may be costly to find out which such a non-common ancestor is. For this purpose, both hierarchies must be transitively upward traversed and compared. However, this task would be facilitated if the DM could catch this information by just comparing TRIDs, hence:

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

3.5 Buffer Manager

From the buffer manager's (BM) point of view, the storage structures for the identifiers should be flexible enough to store short as well as long identifiers. As a matter of fact, static structures are inappropriate, or even impossible to use, if breadth and depth of the transaction hierarchy are not known in advance. Hence, the BM's requirement:

(6) The TRIDs' storage structure should be of variable length and flexible enough to optimize the memory utilization and to efficiently store both short and long TRIDs.

4. Assigning Transaction Identifiers in Nested Transactions

4.1 The Elementary Scheme

The most elementary scheme, normally used as an illustrative example in the literature, 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.

This is a nice and easily understandable scheme which even fulfils some of the requirements we have pointed out previously: A TRID allows the recognition of its parent TRID; deep as well as broad transaction hierarchies are relatively well supported; a TRID reflects the execution history of transactions; and finally, the highest non-common ancestor between two transactions are recognizable from their TRIDs. However, the worst point of this approach is that its memory overhead makes its implementation worthless. Assuming four bytes long integers, we would need $4 \times N$ bytes to identify any transaction at level N. In addition, it would 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 would be allocated to identify it. In summary, although wasting memory space, this scheme has some important properties, so that the other schemes we present are based on this one.

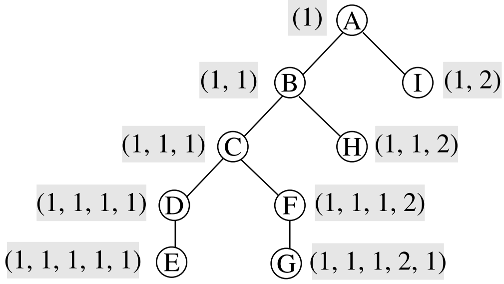

Fig. 1: A transaction tree and TRIDs in the elementary scheme.

4.2 Exponential Growth of Transaction Identifiers - 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, accordingly to the number of bits allocated.

Like previously, 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* (ES, for short), which in turn is composed of several *encoding units* (EUs). The EUs have a predefined length in bits, and therefore can represent a predefined number of TRIDs. Every time the superior limit of an EU is reached, another one is allocated, and the assignment of new TRIDs may proceed, until this second EU is also full, when a third is then allocated, and so forth. In turn, to keep track of how many EUs build an ES, an *encoding unit counter* (EUC) is needed. Such an EUC has also a predefined length, and should precede the EUs for readability (Fig. 2). Hence, to determine an ES, one should:

- read the EUC, stored in the first m bits, where m = length of the EUC, and then
- read the next (EUC + 1) $\times$ n bits, where n = EU length.

How many different ESs 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.

$$2^{n \times 2^m} \quad \text{where: } m = \text{length of EUC}, \; n = \text{length of EU}$$

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

Fig. 2 shows the body of an ES. In our illustrations of this scheme, we have chosen 2 bits for the length of the EUC (m), and 4 bits for the EUs (n). However, the definition of these lengths may be arbitrarily made and adjusted accordingly to the necessities of each particular system. In addition, one could differentiate the EUs and state that there are two EU lengths, one for TL-transactions (longer), and another one for subtransactions (shorter). For the sake of simplicity, we make no distinction in the EU lengths yet (we return to this point in the next section). Finally, each pair (EUC, EUs) represents an ES, i.e., a level in the transaction hierarchy. Fig. 3 presents examples of TRIDs in this scheme.

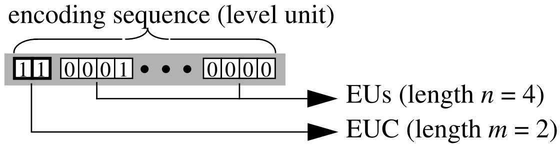

Fig. 2: The EG scheme.

As seen, the EUC ascertains the length of an ES at 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 EUCs and skipping the corresponding number of bits. Therefore, in order to know where a bit stream finishes and 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 in the hierarchy. The most important advantage of storing the number of levels is 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 ES composed of a pair (EUC, EUs). Hence, the first ES of a TRID gives the number of levels in the hierarchy. The EG scheme approaches even more the satisfaction of all our needs:

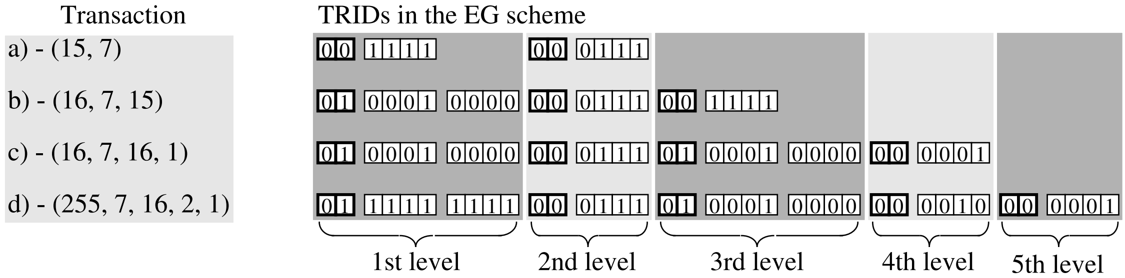

Fig. 3: Examples of TRIDs in the EG scheme.

Requirement 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 ES 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 ESs 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.

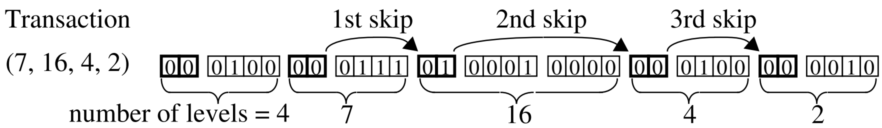

Fig. 4: Capturing the parent TRID in the EG scheme.

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

Requirement 3: Assuming 4 bits as the length of EUs, one may 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 EU 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 EUC and EUs, this scheme always accommodates a potentially very large, but finite number of TRIDs.

Requirement 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.

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

Requirement 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 ESs. 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 one could expand this scheme in order to try to postpone this problem's occurrence.

Extending the EG Scheme

Since in the EG scheme the EUC 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 EUs. We suggest the representation of a counter for the EUC. 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 EUC is reached, another one is allocated, and so forth (refer to Fig. 5). Hence, to determine an ES in this extended EG scheme, one should:

- read the *counter of EUC*, stored in the first k bits, where k = length of the counter of EUC,
- read the EUC, stored in the next (counter of EUC + 1) $\times$ m bits, where m = length of the EUC, and finally
- read the next (EUC + 1) $\times$ n bits, where n = EU length.

Equation II gives how many different ESs may be represented by the extended EG scheme at each level of a transaction hierarchy. By its means, we can perceive the extremely high representation capacity of this scheme. For example, if we choose 2 bits for the counter of EUC ($k = 2$), 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.

$$2^{n \times 2^{m \times 2^{k}}} \quad \text{where: } k = \text{length of counter of EUC},\; m = \text{length of EUC},\; n = \text{length of EU}$$

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

We have sketched in Fig. 5 a) the body of an ES in the extended EG scheme. Fig. 5 b), in turn, shows the minimal TRID, whereas Fig. 5 c) the maximal one. This extended EG scheme copes well with the problem we mentioned before. However, we have advocated that the size of an EU 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 EU. Therefore, in these cases the bits for the EUC 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.

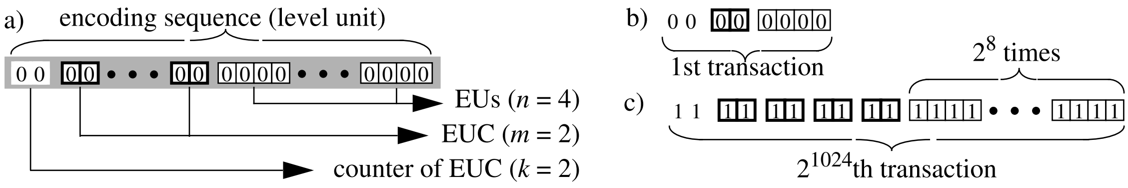

Fig. 5: The extended EG scheme.

4.3 Additive Growth of Transaction Identifiers - The AG Scheme

The idea underlying the AG scheme is very simple. We have a certain number of bits (also an EU) for identifying the transactions at each level, which should cover the sub-TRIDs in the normal, average case. When an EU is full, i.e., all its bits were already used, then another EU is allocated and added to the previous one to proceed with the assignment of TRIDs. Being it again full, another one is allocated, and so forth. The main difference to the EG scheme is that we reserve one representation of bits to the end of signaling us that an EU is full. Hence, when all bits of an EU are set to 0 (zero, our special *full representation*), then the next forthcoming EU pertains to this same level. This scheme is additive in the sense that, in order to capture a TRID at a level, all EUs 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 EU (say, *value*), stored in n bits, where n = EU length, and
(2) check whether EU is equivalent to the full representation (0). If so, then add to *value* the representation capacity of an EU (refer to Equation III), and return to step 1.

Equation III gives how many different representations can be produced per EU, 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 EUs.

$$2^n - 1 \qquad \text{where: } n = \text{length of EU}$$

Equation III: Maximal number of representations supported per EU in the AG scheme.

Fig. 6 shows examples of TRIDs in the AG scheme. In particular and differently from the EG scheme, we have chosen 8 bits as the length of EUs (n). The idea here is as before, a single EU should be enough to identify the transactions at each level, being it not due to an exception case, another EU is used. Therefore, the right tuning of the EU length is a very important aspect, which influences the performance of the whole mechanism.

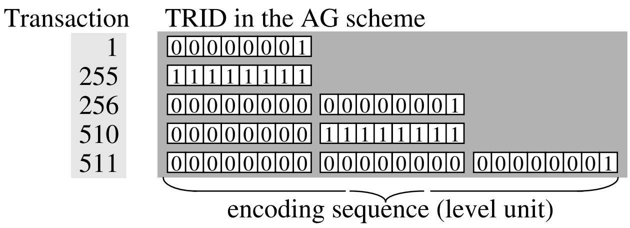

Fig. 6: Examples of TRIDs in the AG scheme.

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 EUC provide enough information to learn the length of the whole TRID. But we do not have EUCs in this scheme, and the number of levels alone is not sufficient, since a single level may spread along several EUs. Therefore, we have chosen for this scheme to store this information as the total number of EUs. In addition, this information will be stored in the same way as the EUs. Hence, the first ES in a TRID gives its number of EUs (Fig. 7). In the following, we analyze this scheme w.r.t. our requirements:

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

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

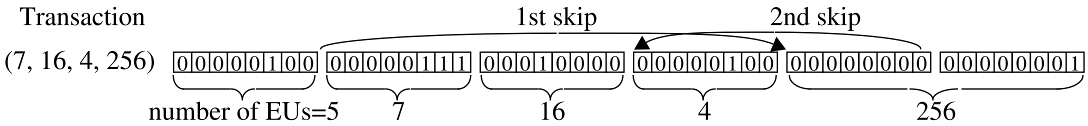

Fig. 7: Capturing the parent TRID in the AG scheme.

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

Requirement 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).

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

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

Summarizing, two important points of this scheme are: A potentially infinite number of transactions can be identified (of course, by an also infinitely large representation), and no extra bits are necessary for EUCs at each level, since a special (full) representation carries this information. A critical 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 EUs. In the following, we present a final and interesting encoding scheme, where we combine both schemes (AG and EG) together. The idea is to capture the best property of each particular scheme in only one scheme.

4.4 Combining the AG and EG Schemes Together - The AEG Scheme

In this scheme, we are going to apply the additive growth feature of the AG scheme to the EUC of the EG scheme. In turn, the EUs 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 ES in the AEG scheme requires the following:

(1) read the value of EUC (say, *value*), stored in m bits, where m = EUC length,
(2) check whether EUC is equivalent to the full representation (0). If so, add to *value* the representation capacity of an EUC (Equation III), and return to step 1.
(3) read the next (*value* + 1) $\times$ n bits, where n = EU length.

Equation IV gives how many different ESs 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 EUCs can increase accordingly.

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

where: m = length of EUC
n = length of EU
i = number of occurrences of EUCs

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

Fig. 8 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 EUC and one EU, like in the EG scheme. However, as soon as an EUC reaches its EU allocation capacity, another one is used to allocate and manage more EUs.

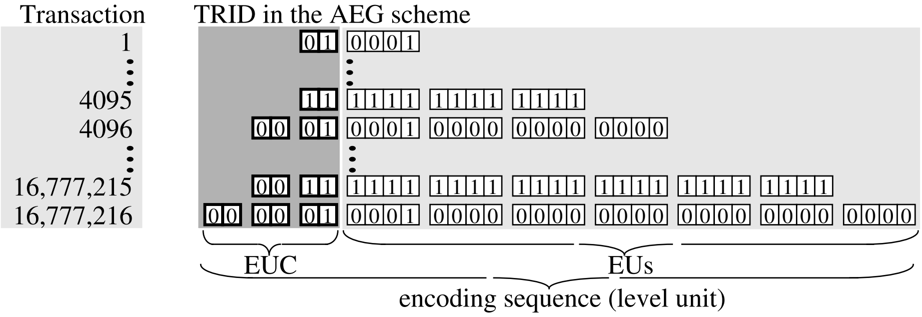

Fig. 8: The AEG scheme.

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 an exponential growth of TRIDs. However, it certainly incurs more processing overhead for interpreting the TRIDs.

5. Performance Evaluation

5.1 The Conventional Scheme

The conventional scheme we have chosen for comparison is the one implemented in PRIMA [6, 5, 7]. In this scheme, the NT 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. 9). A TCB contains four pointers which are used to establish the nested structures. The parent of a transaction may be found by traversing the *parent pointer*. In turn, all children of a transaction can be reached by traversing the *child pointer*, and from this pointer on one can navigate via the *right sibling pointers* to the other children. 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 given by means of a counter.

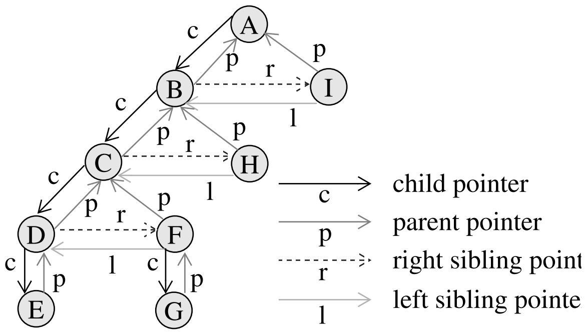

Fig. 9: Transaction trees in a conventional scheme.

5.2 The Algorithms and the Environment

The algorithms we have implemented, in C, may be gotten via *anonymous ftp* under *ftp.uni-kl.de* (131.246.94.94, /pub/informatik/software/rezende/TRIDs_NT). These are of a general use, and so programmed that all EU lengths are represented as meta-information (defines). By this means, one can easily tune the TRIDs to any system. However, for performance reasons, it is desirable that one changes the algorithms accordingly, so that some of the tasks and checks performed may be facilitated. In particular, we have a version of these algorithms tailored to the features of KRISYS [10], where the length of EUs for TL-transactions is four bytes, the one for subtransactions is one byte, and one first byte for the total length information. This version of the algorithms is much simpler. Here, we present the performance results we have obtained with this version. We have run the algorithms in a Sun Sparc Station ELC 4/25[1], with 64 Mbytes of main memory, under SunOS 4.1.4[1], windows system Sun-X11R5[1]. In order to get precise time measurements, we have implemented a kernel module in Sparc-Assembler, which has allowed us to access the Hardware-µsec-Timer of the Sun-Workstation[1]. We have performed the algorithms until the 50th level in a hypothetical transaction hierarchy, where the 1st level corresponds to the TL-transaction. Further, we have repeated all functions thousand times to the end of getting averages, and so not being disturbed by eventual machine overloads.

5.3 The Performance Results

5.3.1 Memory Space Utilization

The first aspect we have compared is the memory space utilization in both approaches (Fig. 10). We have assumed in our comparisons that one EU in the AG scheme is sufficient to store the TRIDs at each level. Otherwise, it would be very hard, if not impossible, to draw comparisons. In the conventional scheme, the number of bytes is constant, independently from 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. In turn, the first level in the AG scheme consumes 5 bytes: 4 bytes for the TL-TRID plus 1 byte for the length information. Subsequently, one more byte is needed for each forthcoming level. As can be seen in Fig. 10, 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 more memory space than the conventional scheme.

5.3.2 Getting the Parent TRID

Whereas in the conventional scheme the TCB of the transaction, after being directly accessed through a hash table, must be 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 (Fig. 11).

5.3.3 Checking the Inferior Relationship

The function for checking the inferior relationship is performed differently in both approaches. The conventional scheme, which uses a bottom-up strategy, varies in terms of execution time accordingly only to the difference of levels between the transactions in the hierarchy, and it is independent from which specific levels these transactions are. In our measurements (Fig. 12), we have then varied this level difference for the conventional scheme, starting from 1 until 50 levels. In turn, the AG scheme is more or less independent from such a level difference. On the contrary, it varies accordingly to the specific levels the transactions are. 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 transactions' levels. Hence, for the AG scheme, we have varied this number

1. Registered trademark of Sun Microsystems, Inc.

of byte comparisons from the 1st to the 50th levels in the hierarchy.

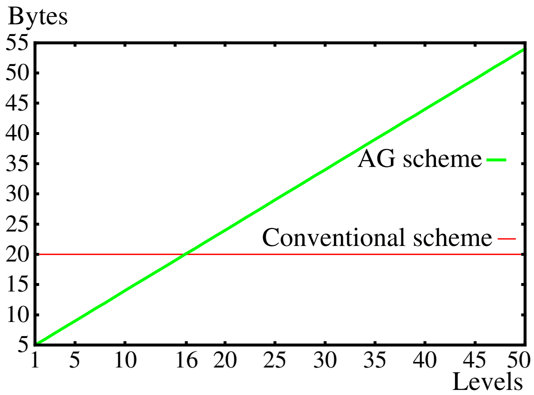

Fig. 10: Memory utilization in the normal case.

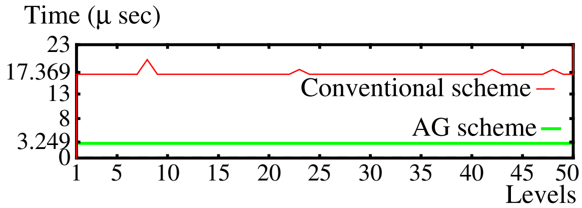

Fig. 11: Getting the parent TRID.

5.3.4 Getting the Highest Non-Common Ancestor

This function 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. Hence, its time measurements vary accordingly to the number of bytes examined, i.e., accordingly to the level such an ancestor is. In turn, in the conventional scheme the TCB of the transaction which is deeper in the hierarchy must be brought to the same level of the other transaction's TCB. Thereafter, the parent pointers of both TCBs are navigated upwards in the hierarchy until the first common ancestor is found. During this navigation, the previous transaction in the path must be remembered. On finding the first common ancestor, the most recently remembered transaction is the highest non-common ancestor between both.

Due to different influencing factors, we have realized different measurements for this operation. All of them have shown almost the same behavior. In the one shown in Fig. 13, we have kept both transactions at the same level of the hierarchy. We have varied then the position such a non-common ancestor is, i.e., level 1 means that it is the TL-transaction itself. In turn, level 50 means that this ancestor is at the 50th level. 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 ES of both TRIDs for realizing that), the conventional scheme takes longer, because it must upward traverse 50 TCBs in the tree until it reaches the TL-transaction. In turn, it gets better performance results as such an ancestor is deeper in the hierarchy (the path it must traverse becomes shorter). On the other side, the AG scheme takes longer because it must compare more ESs.

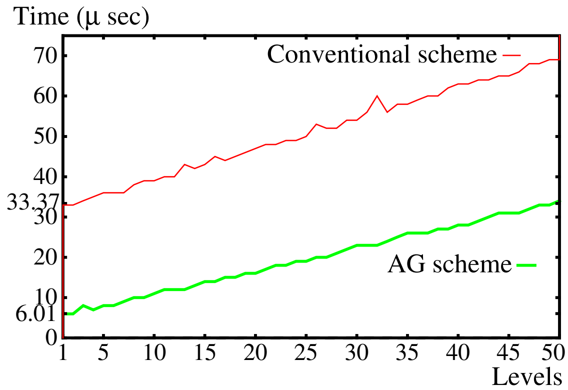

Fig. 12: Checking the inferior relationship.

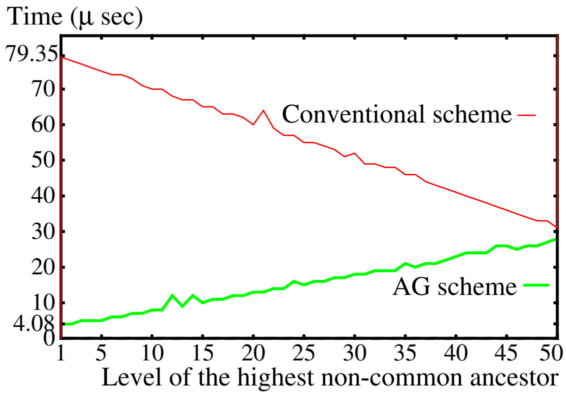

Fig. 13: Getting the highest non-common ancestor.

6. Conclusions

We have addressed the assignment of TRIDs in NTs. Essentially, a TRID should be of variable length, and carry the superior TRIDs. The elementary scheme is hardly worth of implementation. The EG scheme supports a finite number of TRIDs, and must represent extra bits for EUCs. The extended EG scheme allows for more flexibility in the allocation of EUs, putting its upper limit to a very large value with the addition of some more information in the ESs. The AG scheme copes well with the allocation of EUs and needs no counters, and it potentially supports an infinite number of TRIDs, as long as an infinite representation is possible. In addition, it avoids extra bits for EUCs, because it uses a full representation for signaling sequences of EUs. At last, the AEG scheme is not subject to failures in the allocation of EUs, and allows an exponential growth of TRIDs.

We have also realized many performance measurements, the most important of them we have shown here: 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 one. As all has its price, our AG scheme consumes more memory than the conventional one as soon as the transaction hierarchies become too deep. Finally, we hope to have covered a topic of NTs which has not received much attention thus far, although influencing the performance of any systems employing NTs. With the algorithms we have put available via anonymous ftp, we hope to facilitate the work of the ones who might like to implement our ideas in their own systems.

References

[1] Ahamad, M., Dasgupta, P., Blanc, R.J., Wilkes, C.T.: Fault Tolerant Computing in Object-Based Distributed Systems. In: *Proc. IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1987.

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

[3] Dasgupta, P., Blanc, R.J., Appelbe, W.: The Clouds Distributed Operating System. In: *Proc. Int. Conf. on Distributed Computing Systems*, USA, 1989.

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

[5] Härder, T.: *The PRIMA Project - Design and Implementation of a Non-Standard Database System*. SFB 124 Report 26/88, University of Kaiserslautern, Germany, 1988.

[6] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications. In: *Proc. VLDB*, 1987.

[7] Härder, T., Profit, M., Schöning, H.: *Supporting Parallelism in Engineering Databases by Nested Transactions*. SFB 124 Report 34/92, University of Kaiserslautern, Germany, 1992.

[8] Härder, T., Rothermel, K.: Concurrency Control Issues in Nested Transactions. *Journal of VLDB* **2** (1), Jan. 1993.

[9] Holt, R.C.: Some Deadlock Properties in Computer Systems. *ACM Computing Surveys* **4** (3), 1972. pp. 179-196.

[10] Mattos, N.M.: *An Approach to Knowledge Base Management.* Springer, 1991.

[11] Moss J.E.B.: *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, USA, 1985.

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

[13] Pu, C., Noe, J.D.: *Nested Transactions for General Objects: The Eden Implementation*. Technical Report TR-85-12-03, University of Washington, USA, Dec. 1985.

[14] Rezende, F.F., Härder, T.: Concurrency Control in Nested Transactions with Enhanced Lock Modes for KBMSs. In: *Proc. 6th DEXA*, London, UK, 1995.

[15] Rukoz, M.: Hierarchical Deadlock Detection for Nested Transactions. *Distributed Computing* **4**, 1991.

[16] Spector, A.Z., Pausch, R.F., Bruell, G.: Camelot: A Flexible, Distributed Transaction Processing System. In: *Proc. IEEE Spring Computer Conference*, USA, 1988.

[17] Weinstein, M. et al.: Transactions and Synchronization in a Distributed Operating System. In: *Proc. Symp. on Operating Systems Principles*, Dec. 1985. pp. 115-126.