Open Nested Transactions: Semantics and Support

J. Eliot B. Moss
Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003–9264, USA
Email: moss@cs.umass.edu

***Abstract*— We describe semantics for serializable (safe) open nested transactions. Given these semantics, we then suggest hardware necessary to support them directly. We further consider some useful, but not serializable, applications for open nesting, and *their* hardware implications. We focus primarily on *linear nesting*, which we previously argued to be more amenable to hardware support than the general case.**

I. Motivation

There is increasing interest in adding some notion of transactions to general purpose programming languages such as Java (see [1] for example). Transactions promise to make it easier for ordinary programmers to write correct and efficient concurrent applications (at least of certain kinds), by solving a number of problems exhibited by locking, such as deadlock, priority inversion, and concurrency bottlenecks.

We focus here on support for *nesting* of transactions, which we believe is essential for realizing the full potential of transactions. Applications and libraries will call other libraries, and thus language transactions will occur nested inside other language transactions. Simply aggregating these into big transactions leads to loss of concurrency, more transaction conflicts, and more re-executions of failed transactions. Further, a strict transaction model is also rigid, leading programmers to code outside the model. For the sake of correctness, programmers should use transactions wherever possible, i.e., transactions should be the default. Hence, it may be wiser to support reasoned loopholes rather than a strict model.

Here we describe *open nesting*, a transaction model offering higher concurrency that non-nested and simply nested (closed nested) transactions. The database community developed open nesting some time ago, but it is less familiar to language designers and hardware architects, and undoubtedly some of the concepts need a degree of "translation" and adjustment as we attempt to migrate them into the environment of programming language transactions—hence our motivation for recapitulating the model here. We first describe a safe, serializable, model, which serves primarily to offer higher concurrency. However, by "abusing" the model, one can achieve useful excursions from the strictures of serializability, which we also explore to some extent.

II. The Open Nesting Model

Here we offer a necessarily condensed exposition, avoiding formalism. We do this not only for the sake of conciseness; we also hope that the informal presentation builds better intuition and conceptual understanding.

Transactions: A transaction is a dynamic execution of a sequence of *operations*, which should appear to execute instantaneously with respect to other concurrent transactions. If a transaction fails, it is as if it never ran (no partial executions). A failed transaction may be retried, depending on the nature of the failure, in order to achieve exactly-once execution wherever possible. These transactions offer *Atomicity*, *Consistency*, and *Isolation*, the first three of the ACID properties; in this setting we forgo Durability. Gray and Reuter's text on transaction processing [2] remains an excellent background work on transaction concepts and their implementation in information systems (databases, etc.). Before moving on, we stress the point of view of a transaction execution as a *sequence of operations*. As database folks might say: "The log is the truth—the state is only a cache."

At the lowest level of abstraction, a transaction's operations examine (read) and modify (write) state, and thus a transaction can be summarized in terms of the set of locations and values it has read and written, in terms of bytes or words. However, it remains helpful to realize that this is just a summary of the effect of a transaction's operations at this lowest level of abstraction—though it is the level at which Transactional Memory (hardware support) will tend to operate.

Conflict: We say two operations *conflict* if they do not commute, i.e., swapping them causes either or both of them to return a different result or to lead to a different state (possibly returning a different result in the future). Two transactions conflict if they issue operations that conflict. Transaction systems impose *concurrency control* to prevent conflicting transaction executions.

Transaction abort and commit: If a transaction completes successfully, we say it *commits*. Concurrency conflicts, and other events (errors and exceptions of various kinds), can lead to the need to *abort* an in-progress transaction, i.e., to remove any of its tentative effects, and perhaps to retry it. In general, to abort a transaction one must *undo* its sequence of operations, by executing inverses of those operations (undos) in reverse order. In the special case of the level of abstraction of reads and writes, it is adequate merely to restore the state of any modified location to what it was before the transaction modified it. However, it is helpful to view this as just an optimization of undoing in reverse order. "Read" and "Write" are perfectly good operations, too, and can be logged and undone just fine. But as we will see later, framing things at higher levels of abstraction can offer increased concurrency.

Closed Nesting: In closed nesting, a transaction may execute *child* transactions (subtransactions). Following one of our early descriptions [3], we assume that a parent does not execute while any of its children do, but in general a transaction may have multiple concurrent children (see also [4], [5]). Using the log model, if a child commits, we append its log to its parent's log. If a child aborts, we undo its actions and discard its log. Importantly, a child abort does *not* abort its parent, though the parent may be notified of the abort and take alternative action of its choosing, including aborting itself. One interesting property of the closed nesting model is that a child's operations are deemed never to conflict with operations of its parent (or any ancestor). However, conflict of operations between a child and a non-ancestor (sibling, descendant of sibling, or a transaction that is or is part of a different top-level transaction) *does* induce transaction conflicts. The *aggregation* of child effects into parent logs demonstrates the aggregation we described before, and also shows

how closed nesting is semantically "*flat*", operating at a single level of abstraction.

Linear Nesting: In linear nesting we restrict a transaction to have at most one child at a time. Thus the tree of running descendants of any transaction is a *line*, not a general tree structure. In linear nesting one can identify a transaction solely by a top-level transaction id (the original ancestor of the transaction) plus a nesting level. This greatly simplifies determining ancestor-descendant relationships in hardware, and also permits some useful further optimizations, as we recently described [6].

Open Nesting: Open nesting is, not surprisingly, similar to closed nesting. However, in the open nesting case the parent and child execute at *different levels of abstraction*. We explain this with a running example. Suppose you are to implement a highly concurrent version of the Java HashMap library class, but with transactional semantics. For concreteness, we consider a closed (chained) hash table. This consists of an array of linked lists, one list for each hash bucket.

Consider first how this would work without nesting, in a model that operates at the level of reads and writes of memory words. For either the get or put operations, you first compute the hashCode of the object presented as a key. This touches the key object, and perhaps updates a header word of that object, used to cache the hash code. You then use the hash code to determine a bucket, and pick up the linked list for the key's bucket and begin examining the objects in that list. You will immediately conflict with any concurrent transaction that has updated the pointer to the head of the list (but not yet committed), and, if you get that far, with any transaction that has updated the list farther along. You also examine other key objects as you traverse the list looking for a match to your key object. You can easily conflict with transactions that created or updated those objects. Finally, in the case of put, you must update the list, creating further conflicts.

Of course one way people approach transactions is to say "Oh, I'll just do each hash table operation as its own (top-level) transaction. Then all these conflicts go away, except for low-level conflicts between actually concurrent table operations." But this strategy does not allow the table operations to be composed effectively into bigger transactions, and such composition is sometimes necessary for correct semantics. For example, an algorithm might manage a collection of objects by keeping them on various lists, with the requirement that each object always be on or another of the lists. To *move* an object from one list to another cannot properly be done as two *separate* transactions, since another transaction could slip in the middle and observe that the object being moved is currently on *no* list.

(Closed) nested transactions allow you to aggregate multiple low level transactions into a bigger low level transaction. Closed nested transactions might be adequate for the "move this object from one list to another" case, since the list deletion and insertion are relatively short and closer together in time. But complex operations can have many effects in many places, and can run for a significant period of time. Hence, conflicts can be a real problem. However, many of the conflicts at the level of memory words are what we call *false conflicts*: conflicts that occur at a low level of abstraction, but which are not essential to the *semantics* of the operations. For example, when looking for a particular key in a HashMap, other keys we happen to touch along the way do not matter, and it is unfortunate that observing them may lead to unnecessary conflicts.

Returning to the HashMap example, how would it work with open nesting? The lower level (child) transactions work with memory reads and writes as their operations. The high level semantics of the table have to do with presence and absence of keys (and values) within a set of mappings. If two transactions try to access and update the same key, that is a fundamental conflict, at the higher level of abstraction, regardless of implementation. But the accident of two *different* keys lying in the same hash bucket is not fundamental, but a false conflict when viewed from the higher level.

Just like non-nested and closed nested transactions, open nesting must be able to commit and abort transactions, and to indicate when conflicts occur. The critical notion here is that when an open nested transaction commits, we accomplish a *change in level of abstraction*. In the HashMap case we switch from memory reads and writes to a view of a HashMap as a set of key-value mapping pairs. The essential operations are inserting a pair, observing a pair, and deleting a pair, and none of these commute on the same pair except two observations of it. Thus, these operations are what we need to log at the higher level. So, as a HashMap.get transaction commits, we discard the lower level read-/write-set log of the child and add (at least logically) an *observed* record to the parent's log. We proceed similarly for insertion and deletion. Please ignore for the moment exactly how this might be *implemented* and try to focus on the concept.

What if the parent aborts? When we switched levels of abstraction as the child committed, we also switched the level at which we must abort the work done by the child. To abort a completed insertion, we must delete the mapping, by invoking a delete operation, *not* by just slamming back in the pre-insertion state of memory words. Intervening insertions and deletions, though they are for other keys, nevertheless can have rearranged the hash bucket or the whole table. So, which bucket pointers need to be adjusted can be quite different. One way of looking at this is that the commit of the open nested transaction *really* (finally) commits the child's state changes, as seen at the lower level of abstraction. The only way to reverse the effect of the child action is to execute another action, in this case a delete of the pair the child inserted. Thus, from the standpoint of the lower level, the upper level undo is *forward progress*.

It should clear that when the child commits and gives up its read- and write-sets, operations that conflict at the low level will be able to proceed, and we achieve higher concurrency. However, for proper *high level* semantics, we must detect and prevent high level conflicts, i.e., conflicts that arise from the inherent *semantics* of the higher level actions. That is, just as the lower level has its concurrency control, detecting and responding to conflicts at the level of memory reads and writes, the higher level needs its own concurrency control in terms of what is in and out of the set of mapping pairs in the HashMap. Because semantics is very data-type dependent, we cannot expect simple and uniform semantics for conflicts (or for undos) at the higher level. Rather, they must be given by the programmer.

What might higher level (abstract) concurrency control look like for our HashMap? One strategy is to associate with each mapping pair a set of reading transactions and a writing transaction. Thus we implement read and write locks at the level of mapping pairs, in software. Each reading operation would need to acquire a read lock, each writing operation a write lock, etc. There are additional subtleties. For example, some operations in HashMap are on the *whole map*, e.g., returning the whole key set, or all pairs. These operations prevent any additions or deletions to the set, because they effectively make assertions about what is *not* in the set as well as what *is*. This is a subtle point, demonstrating the abstractness of these locks: if we probe a HashMap for a key not in the map, we must also "lock" this fact, since the caller detected absence of the key and may be relying on it.

When a higher level transaction commits (or aborts) we must release these locks. Thus, we need to support on-commit actions as

well as on-abort undos (which invoke an inverse and *then* release the lock). In summary, just as an open nested commit requires a change in level of abstraction with respect to undos (logs), it demands a change in point of view with respect to concurrency and visibility (locks), a switch from lower level conflict management (perhaps in hardware) to more abstract management (in software).

Open Nested Insertion Dissected: Now we can describe insertion into a HashMap more completely. It first computes the hash code of the key object. The hashCode operation is itself a good candidate for open nesting, to avoid false conflicts with other manipulations of the key object (including its creation). Since hashCode is supposed to be idempotent (always return the same value for the same object), it does not abstractly conflict with any other operation on the key object. The put (insert) operation then proceeds to determine if the key in question is locked. If it *is* locked then it waits (or, depending on policy, causes an abort, of itself or of conflicting transactions). If the key is not locked, the transaction acquires a lock for it. This lock is specific to the particular HashMap, and hence records (somehow) the HashMap, the key, and the locking transaction. These lock table manipulations are part of the ongoing open nested transaction. The transaction records the lock in a list of things it needs to deal with on completion (either commit or abort). The transaction also records, in its parent's list of things to do on abort, the undo of this operation, namely a deletion of the key in question. In the midst of all this the transaction also does its basic work of inserting the pair into the HashMap.

If the open nested transaction commits, it drops its read- and write-sets. Thus its updates to the HashMap, to the lock table, and to the parent transaction's commit action list and abort action list are installed and become visible, at the level of memory reads and writes. If a conflicting get or put comes along, the conflicter will see this when it searches the lock table, and will trigger concurrency control action. If the open nested action aborts, we discard its low level writes, to the basic data structure, to the lock table, and to the commit, abort, and completion action lists. If the *parent* aborts, we run the abort actions, as open nested actions themselves, and then drop any locks.

Multiple Levels of Nesting: What if the HashMap operations are themselves part of a larger action? We can use closed nesting to aggregate actions. This is appropriate if there is little concurrency to be gained from adding a level of abstraction. However, we can also add a level of abstraction by using open nesting again. In this case, when the higher level open nested action commits, we discard the lower level operation undo log (abort action list) and perform any commit actions (such as releasing locks). The higher level open nested action is guaranteeing correctness, according to *its* even more abstract viewpoint. Some years ago we proved the correctness, in terms of equivalence of abstract state and results, of composing open nesting in this way [7] (see [8], [9], [10] for a sampling of related work by others).

III. Example: Phone Directory

We now offer an example that uses two levels of transaction nesting (closed on top of open) and a concurrent update scenario to illustrate further our proposed style of use for open nesting (i.e., abstract serializability: serializability in terms of the higher level (abstract) state). We assume a computerized telephone directory, organized by name, department, and phone number. The basic data structure consists of phone directory entry objects; we call the class PDE for short. To support various lookups, the overall directory (which we call PD for short) has two indexes (maps): a SortedMap from person name (String) to PDE, and a SortedMap from department name (String) to SortedMap giving an alphabetical list of PDE's for people in the given department. We use SortedMap's rather than HashMap's to support alphabetical listing of ranges of names, etc., as well as equality searches. We assume the SortedMap structures are implemented using open nesting for all of their operations. Operations on a PD use closed nesting if they need to examine or update multiple structures in the PD. For example, adding or deleting a PDE requires updating two maps (the by-name map and one department's map); changing a PDE's department requires updating two department maps; etc. Some operations, such as looking up a single name, do not require both levels of nesting.

Concurrency Control: The SortedMap we envision allows range lookups: at the very least individual names (range of size 1) and all names (full range), but likely also including all persons whose last name starts with a given prefix, etc. Thus an appropriate locking model is locking ranges of names. Two range locks conflict if their locking *mode* (read, write) conflicts and their ranges *overlap*. (This is a simple and highly implementable restriction of *predicate locking* [11], which in general is NP-complete.) Note that an application built on top of the PD might hold a lock a long time, e.g., while displaying a user's PDE for possible editing in an interactive window, to avoid having the information change while the display window is open. (One can use a timeout on data entry to prevent the lock from being held indefinitely, or somehow allow a conflicting updater to force the interactive transaction to abort.) Since some lookups and updates proceed through the by-name map, and others through the department map, we associate the range lock table with the PD as a whole, and use lowest level concurrency control (read- and write-sets) on the SortedMap's.

We observe that a range lock covers not only those PDE's in the PD at the current time, but the *infinite* number of PDE's that lie in the range and are *not* in the table. That is, one is locking PDE absence from the table as well as PDE presence. This gives one confidence that a range retrieval gets all the entries in the PD at the current time. Note that this effect cannot be achieved simply by locking PDE's individually!

Scenario: Suppose we have two concurrent transactions on a PD, one adding an entry for "Moss" in "Computer Science" and one deleting an entry for "Mosher" in "Computer Engineering". These person and department names are close enough alphabetically that they may well lie in the same internal leaf object in the by-name SortedMap data structure, which, for the sake of argument, we implement with a B-tree. We further observe that the PD updates may be part of much larger and complex transactions, likely distributed, for provisioning and deprovisioning of staff in personnel processes. Thus, even after the PD operations are committed (from their point of view), they may be part of a larger nested transaction that may take some time to complete.

Suppose the insertion and deletion start about the same time, with the insertion coming first. We'll call the transactions I and D (see Figure 1). They are closed nested transactions, each consisting of several steps. First, each acquires a range lock on the (small) range consisting of the person's name. We assume that acquiring the range lock is an open nested action (I_1 and D_1), on the range lock table. We do it this way so that the locking table will not be a bottleneck. D next tries to delete its name from the by-name SortedMap (open nested transaction D_2). It gets to the leaf bucket, examining index buckets and name String's along the way, and finds the relevant entry. Rather than physically removing the entry right away, it adds to D's commit list an operation that will do the physical deletion later, as

part of committing D. The reasoning here is that one should not physically remove data until it is finally committed, since otherwise, when there is an abort and one would need to re-insert the removed object, there might not be space. This is a general principle in designing transactional data structures, understood by the database implementation community.

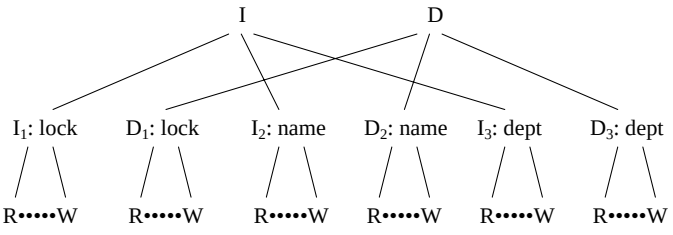

Fig. 1. Interleaved Concurrent Insertion and Deletion

Meanwhile, I_1 inserts its range lock in the lock table, possibly needing to wait (briefly) or retry if its lock acquisition overlaps with D's. I then proceeds to search the by-name SortedMap (open nested transaction I_2). It finds the same bucket and looks for the right place to insert the new key. It attempts to shuffle higher entries over by one and to insert its new entry. It thus does writes to the leaf bucket's words, and likely conflicts with the deletion (D_2). However, as soon as D_2 commits, I_2 can proceed and enter its information. It enters in I's abort action list an operation to delete the new entry (exactly like D_2's operation added to D's commit list). Similar to D_2 and I_2, open nested transactions D_3 and I_3 manipulate the department maps. Here they will not conflict, since the PDE's are for different departments.

The figure is drawn to illustrate significant points. It is intended to be read with time passing from left to right. Looking at the lowest level, we see that the reads and writes of each open nested action (I_i, D_i) are clustered together. While in reality these reads and writes may be somewhat mixed together, serializability of the individual transactions I_i and D_i requires that it be *as if* they were clustered together. The figure shows one possible ordering for the I_i and D_i. Thus, from the lowest level perspective, the order of the transactions is I_1, D_1, I_2, D_2, I_3, D_3. However, at the higher level of abstraction, since I and D are operating on different keys, the I operations commute with the D operations, so the order given is *equivalent to* (i.e., *equal* in the abstract to) I_1, I_2, I_3, D_1, D_2, D_3. Thus, I and D are abstractly serializable: even though they are tangled together at the level of reads and writes, they are ACI, Atomic, Consistent, and Isolated, in the abstract.

When I and D are ready to commit, they have the following actions on their respective commit, abort, and completion lists:

- I's commit list: empty
- I's abort list: remove its PDE from the by-name map and its department map
- I's completion list: remove its range lock
- D's commit list: remove its PDE from the by-name map and its department map
- D's abort list: empty
- D's completion list: remove its range lock

Here we present the lists with their actions in the order in which they were added. If I aborts, the system will (before actually abandoning I) execute its abort list actions in reverse order (undoing things in the opposite order from which they were done, i.e., last to first) and then execute I's completion actions in reverse order (it is not clear this order matters as much). If I commits, the system executes the commits actions in the given order, and then the completion actions in the given order. The same applies to D. It is important to understand that I (or D) is considered to abort if it itself aborts (e.g., because of concurrency control conflicts or some kind of failure) or if any ancestor aborts (before I (D) commits). On the other hand, I (D) does not commit until an *open* ancestor commits (with top-level ancestors understood to be open).

We summarize below the protocol for managing a transaction's lists when the transaction completes. Note again that we consider top-level transactions to be *open*.

- *On open commit:* perform commit actions, then completion actions; ignore the abort list
- *On closed commit:* append each list to the parent's corresponding list
- *On abort:* perform abort actions in reverse order, then completion actions in reverse order; ignore the commit list

Please refer to Section V for corresponding protocols for managing read- and write-sets (lowest level actions).

An Abort Scenario: Suppose for some reason the I transaction aborts after I_2 and before I_3. It is important that the undo be a separate operation at the abstract level. If another user's transaction added or removed a different key from the same bucket, then just backing out I_2's writes will be wrong, since the bucket's state has further evolved since I_2 ran. Likewise, D_2's commit operation, to really remove its key, must also be a separate abstract operation, since intervening insertions and deletions of other keys may move the slot containing the key D_2 wishes to delete. Likewise it is important to maintain abstract concurrency control (abstract locks). Otherwise, in the case of I aborting after I_2, the inserted key might be observed by another transaction—but then I_2 almost immediately goes away. In this case I is not properly isolated or atomic with respect to other transactions.

Some Special Cases: In the phone directory example we presented a fairly general case, a collection with multiple indexes, with individual operations (insert, lookup, remove individual entries) and collection operations (lookup all keys in a given range (not covered in detail)). We now consider a couple of special cases, which begin to show how the general model can sometimes be streamlined.

Special Case: Interning. The Java String class maintains a table of unique copies of particular strings, via its intern operation. This table offers only the one operation, which conflicts only with interning another String consisting of the same characters. However, a conflicter can simply be re-run. Further, there is no need to remove an interned string if the intern operation later aborts. For highest concurrency one should implement intern as an open nested action. It needs no abstract lock, and no (abstract) commit, abort, or completion actions.

Special Case: Creating Strings. Following on from the intern example, for intern not to encounter conflicts when examining string contents (it compares strings using String.equals, etc.), String constructors should be open nested, committing their memory writes immediately. Thereafter, a String is immutable, so there will be no conflicts. Like interning, constructing strings will have no commit, abort, or complete actions. Realizing that String constructors should be open nested leads us to:

Special Case: Java Object Allocation. Java object allocation, up through the point of filling in the object header with its type information, etc., and zeroing the object contents, should also be open nested. This will insure that transaction aborts can never make visible an object that does not have a proper type and safely initialized fields. (This does not prevent the escape and visibility of an object whose constructor has aborted, leaving the object's fields null, but at least we won't violate Java type safety.)

IV. What Transactions Don't Do

In the interest of "full disclosure" it is important to be clear about some things that transactions do *not* do, and for which one needs other mechanisms:

- Transactions do not arrange communication between threads. In fact, the I of the ACID properties stands for Isolation, *non*-communication! To obtain serializability, one can *connect together* the transactions of threads that communicate, so that the transactions all commit or all abort (or rollback to a point prior to the communication, etc.). Luchangco and Marathe describe a design to accomplish this [12]. In some cases one may wish to relax the restriction that communicating transactions be connected for purposes of commit and abort (see Section VI).
- Transaction processing does not impose any particular order on transactions issued by concurrent threads, nor does it inherently reveal their order. With transactions one obtains atomicity (the A of ACID), and by implication Consistency (the C). Similarly, with stylized use of Java locking (roughly speaking, applying synchronized to each operation that needs to be atomic) one obtains the same properties. But the Java memory model also imposes *ordering* between synchronized operations issued by the same thread and by different threads on the same object. This cuts both ways: transactions may allow more concurrency, by *not* imposing ordering; Java locking imposes ordering, which is a kind of communication (i.e., it allows reasoning about ordering of events).
- Transactions do not themselves introduce concurrency. For example, one might want a certain loop to execute with its iterations logically concurrent (to obtain concurrency). Executing each iteration logically as a transaction will insure that iterations that happen to conflict execute atomically. One may further want the iterations' transactions to commit in the loop's original order—or perhaps not, if one knows the order doesn't matter.

V. Supporting Open Nesting

We now take up the question of how to *support* open nesting. First, it is reasonably clear that some things need to be specified by the programmer. In particular, the programmer will need to indicate, for each operation, its inverse. The programmer will also need to specify which pairs of operation invocations conflict. If the programmer supplies this information at a fairly high level (e.g., tables of inverses and conflicts), then the language implementation may be able to generate much of the detailed code required, especially if we provide suitable generic libraries for managing lock tables, and so forth. The language implementation can also manage the commit action, abort action (undo), and completion action lists. There is no real need, and probably no benefit, from supporting these parts of open nesting implementation in hardware.

What can hardware do? First of all, hardware can provide the lowest level transaction semantics: read- and write-sets on memory words. Of course this can also be done in software, as a number of recent software transactional memory implementations demonstrate (e.g., [13], [1], [14]). But how do such hardware transactions interact with nesting? We have sketched elsewhere hardware to support linear nesting for closed nested transactions [6]; there we also suggested one specific behavior for that hardware when an open nested transaction commits. However, that was just one possible position among many. We continue to assume that aborts of open subtransactions drop the read- and write-sets, and commits install the subtransaction writes to top level. But that still leaves a number of options of what to do when an open nested transaction commits. Here are some of those options:

1) Reads and writes aggregate into the parent, as with closed nesting.
2) Drop subtransaction read *and* written locations from all read- and write-sets.
3) Drop subtransaction written locations from all read- and write-sets, but leave ancestors' reads (updated with values written by the subtransaction values).
4) Leave ancestor read- and write-sets alone (updated with values written by the subtransaction).
5) Retain subtransaction written locations as read-set entries in the parent, and drop subtransaction read-set entries.
6) Retain subtransaction read and written locations as read-set entries in the parent.

These options vary in the degree of concurrency they allow, and in the strength of the guarantee they provide to the subtransaction's invoker. For example, the first option has the least concurrency and the strongest guarantee. However, because it offers no increase in concurrency over closed nesting, it would not be a particularly useful choice. The second option offers the highest concurrency, by dropping the most items from read- and write-sets (which are used to determine conflicts). However, it might be surprising that some objects, recently touched by the parent, are now "magically" available for any other transaction to modify at will.

Let's pick apart the situation a little more, using the HashMap example. The state of key and value objects is quite distinct from the state of the HashMap itself. The internal structures of the HashMap, its array of buckets, the bucket linked lists, etc., are independent of the key and value objects, and private to the HashMap. Thus, releasing all read- and write-sets on the HashMap is not a problem, provided all uses of the HashMap follow the open nesting protocol, with its high level concurrency control, etc. In this case, there is one possible difficulty, namely the need for the HashMap operations to call hashCode on the key objects. Computing the hash code might involve reading, or even updating, a variety of words in the key object (and other objects), so arbitrarily dropping those words *from all ancestors* could do strange things if the ancestors have been manipulating the objects. It seems best to leave ancestor words alone, which favors the fourth option on our list. With this option, any data structures accessed only by open nested actions (such as the internals of the HashMap) are effectively entirely committed, while we interfere least with ancestor read- and write-sets, and thus with ancestor concurrency control expectations.

A possible difficulty with hashCode is that user-implemented hash operations could still encounter conflicts. For example, if the transaction that created the key object and initialized it has not yet committed, but it allows a reference to the key object to escape (e.g., via an open nested transaction that writes the reference into some shared data structure), then another transaction could conflict when trying to read the object's data to compute its hash code. This suggests that object initialization should be an open nested transaction, so that object state cannot "disappear", even if the creating transaction aborts. There are safety concerns here previously noted: we cannot have object headers turning into zeroes (or worse, if initialization does object zeroing, whatever arbitrary contents were in those words before!). Likewise, in order to prevent unwanted concurrency conflicts, programmers may need to exercise care in implementing hashCode. For example, they can provide an extra field to hold their user-computed hash value, and hashCode can

consult this first. If we use a separate open nested transaction to initialize the cached hash code, and commit it, then our HashMap operations will only read this value, which will never be updated again. Thus we will have no future conflicts on hashCode.

Revised Proposal: In sum, considering the overall programming model, we suggest that open nesting support the fourth option above, namely that when an open nested action commits, we drop its read- and write-sets, leaving ancestor read- and write-set items alone, *except* for updating their values to correspond to what the committing transaction wrote. This enhances concurrency while preventing ancestor concurrency control "surprises". This proposal differs from the one we previously offered [6], which discards an ancestor item if the open nested transaction writes the same location.

Style of Concurrency Control: Thus far we have presented concurrency control in terms of *locking* (read- and write-sets are effectively fine-grained short-term locks on individual memory words or cache lines), sometimes called *pessimistic concurrency control*. There are also *optimistic* approaches, which proceed without necessarily checking for concurrency conflicts until a transaction attempts to commit (Kung and Robinson's work [15] is foundational, and this line of research and its application continues [16], [17], [18], [19]). There is no fundamental reason why one cannot apply optimistic concurrency control to transactional memory, open transactions, and even multi-layering of open nesting. However, for safety it is important that systems be designed so that a transaction will never see and attempt to operate from an *inconsistent* state. We do not try to determine here what might be the specific implications of this design constraint.

VI. Bending the Rules

As with anything requiring correct programming, programmers can goof when writing a class such as HashMap, and end up offering broken semantics by failing to use open nesting correctly. However, with good library and language support, allowing descriptions of inverses and conflicts in a more declarative way rather than requiring that they be programmed "by hand", we can substantially increase the reliability of open nesting.

For its part, the hardware mechanism fundamentally enhances concurrency, through "early" release of read- and write-set items. We can bend the rules of safe open nesting to take advantage of that concurrency in useful ways. Put another way, sometimes we do not need or desire serializability. Here are some examples of applying open nesting to achieve useful non-serializable behavior:

Logging/Tracing: When entering information in logs, or collecting traces, one will often desire the information to be recorded no matter what, i.e., even if the containing transaction aborts. In the abort case, one might want to add another log/trace record indicating the abort. This kind of recording is easy to do with an open nested action whose abort action just appends the abort notice. It remains helpful to use a transaction when writing a log/trace record, so that if bits of the record get added to a buffer piecemeal via a series of OutputStream method calls, the record will not be mixed together with output from other log/trace records being written at the same time. But once the individual log record is done, the log/trace stream is available to record other events, from the same thread or from a different one. This is an example of a permanent effect.

Communication: Open nesting can be used to support essentially any kind of communication, because it can immediately and permanently commit changes to shared communication buffers. Of course one might still encounter conflicts on the objects that were communicated. If we desire the system to roll back when a receiving transaction aborts, then we need to introduce some additional mechanism to link the abort and commit of different transactions. Helpful work in exploring these ideas includes the ACTA model [10] and Pedregal's dissertation [20].

Work Queues: A special case of communication is shared work "queues" (not necessarily FIFO). Here, one transaction can post a unit of work, committing it to the work queue. It uses an open transaction to do this on the basis that it knows the unit of work is necessary/helpful (or at least not harmful), no matter what. An idle thread can pick up a work item, again committing immediately the removal of the item from the queue. However, if the recipient aborts, it can arrange its abort list to re-insert the work unit into the queue, to make sure the work unit is handled eventually.

VII. More Applications of Nesting

While it should be clear that open nesting will require a certain amount of run-time system support, we ask now the converse question: *Can open nesting help in implementing modern run-time systems?* We first observe that transactional memory, and open nesting as well, might be of considerable use inside operating systems, to overcome concurrency bottlenecks and increase software reliability. The trick in using inside an operating system is that the operating system may be involved in providing some of the support mechanisms, e.g., managing overflows of hardware transactional caches, etc. Thus, the operating system itself may not have as rich a transaction model at its disposal. However, operating system layering, i.e., carefully coding those components that provide the richer transaction model, and then using that model more freely elsewhere, may address that issue.

Within the purview of programming language run-time systems we briefly consider these issues: exceptions, memory management (allocation and garbage collection), checkpointing, conditional atomic actions, and sequential rollback.

Exceptions: As we have noted elsewhere [6], Harris and Fraser's design for transactions in Java [1] took the approach that exceptions thrown during a transaction constitute *normal completion*, as far as the transaction is concerned. Their approach has the virtue of least disturbance to original Java semantics. However, we believe that in most cases when things go wrong, programmers want partial work cleaned up. Usually, a thrown exception indicates that things have gone wrong. Thus we suggest that throwing an exception should generally abort its containing transaction (unless caught within the transaction). This avoids the programmer having to try to back things out "manually", using try-finally blocks. It is well-known that exceptional paths in programs are more likely to be wrong, partly because they are harder to think through, partly because they are frequently an after-thought, and partly because they are hard to test (it is difficult to generate all the necessary scenarios to test them). Transaction abort seems well-suited to cleaning up exceptional cases.

While the idea seems good, there are a couple of technical difficulties in carrying it out. First, it is most likely the default that if a transaction aborts, we retry it. This is sensible in the case of concurrency conflicts, for example, and possibly in some exceptional cases. Consider, however, the case of a web purchasing transaction where in the middle of various steps (updating inventory, generating packaging and shipping orders, etc.) we discover that the credit card is invalid. We rightly wish to back out of the steps being taken, but we should not automatically retry the credit card. We need to distinguish a category of exceptions that will indicate that the transaction should not be retried. This is easily done for Java (and

several other languages) by indicating that exceptions in a particular part of the Exception class hierarchy will not be retried.

A more serious obstacle is that transactions are themselves objects, and need to be created and filled in, and not have their contents zeroed out when the transaction aborts. That is, they are constructed by an aborting transaction, but their state must be visible to exception handlers outside the transaction. One way to accomplish this is to have the run-time support for the throw statement make *copies* of objects reachable from the exception object being thrown, and to make those copies using an open nested transaction. If the exception object refers to objects that existed before the aborting transaction began, it might not copy those. Their state will be rolled back (if it was updated), but it may help to refer to them (e.g., information about the credit card that the transaction found to be invalid). The whole matter bears further study, but it does appear that open nesting can assist in achieving helpful semantics for exceptions.

Memory Management: There are two cooperating parts to programming language memory management: allocation and reclamation. We have already suggested that when an object is allocated and initialized (header filled in and body zeroed), that work should be an open nested action, so that the rest of the system will never see an object that is not type-safe. In highly concurrent situations one might even split allocation into two open nested transactions: the first one finds an available block of storage and reserves it (increments a bump pointer, removes from a free list, etc.); the second one does the low level initialization of the new object. In a world of explicit storage management (malloc/free), the undo of allocation might free the allocated space (return it to a free list). This is a reasonable default policy for that storage management regime, but it is well to keep in mind that open nesting can also allow escape of a reference to the new object. Such published references needs to be backed out in case of abort, or the default behavior overridden.

In the case of implicit storage management, i.e., garbage collection (GC), the "undo" of allocation is garbage collection. However, it might be appropriate for an aborting transaction to zero out reference fields of objects whose construction is aborted. This will reduce GC work and perhaps allow more objects to be reclaimed (i.e., it reduces GC over-estimation of liveness of other objects).

What about using open nesting to assist concurrent GC? A concurrent GC runs at the same time as the application. Most concurrent GCs make changes to application objects while they run. For example, they set "marked" bits in object headers, or, in copying GCs, even update references to point to new copies of the target objects. It would seem that open nesting is well suited to this task, since a short open nested transaction can permanently update the object headers, references, etc. However, transactional memory introduces an interesting phenomenon, as illustrated by the following scenario. Suppose field f of some object starts out referring to object x. An executing, but not yet committed, transaction T updates f to refer to object y. A concurrent copying collector wishes to redirect object y to a new copy, y'. An open nested action can do this, though there is a potential conflict between the GC and T. A more subtle case is when the GC wishes to redirect x to refer to x'. Field f does not contain the value x when the GC looks at it, and then T aborts, then f will have the value x restored.

One strategy we can pursue for supporting concurrent GC is rather different: it is to offer a *reflective view* of other threads' state, a view that reveals all the possible values a location might end up with depending on the pattern of transaction commits and aborts. That is, the GC needs to be able to view the transactional memory contents. More than that, it needs to update them, e.g., to change the record rollback value of x to x'. In a sense the GC is a level of abstraction *below* the ordinary computation, and this makes sense, since ordinary computation is on type-safe objects, while memory management's job is (in part) to manufacture the view of type-safe objects from flat sequences of memory locations. Rather than requiring special hardware or operations, the GC might request that a thread be frozen and its transactional state information be dumped to memory. The GC could then operate on and update that state (which is accessible only to run-time system components). When the thread restarts, we reload its transactional state (but with new values, e.g., x' instead of x).

Another approach to implementing concurrent GC is for the GC not to manipulate words where it conflicts with an ongoing transaction. The GC might wait, might go do other work for a while, or might eventually force the ongoing transaction to abort (to guarantee GC progress). In any case, hardware transactional memory support will simplify implementing concurrent GC correctly. In fact, our own early hardware proposal [21] grew in part from our experience designing lock-free concurrent GC [22]. It is quite helpful if one can perform atomic read-modify-write operations on at least two separate memory words.

Checkpointing: It is often helpful to be able to roll back a transaction to any of various points, and restart forward progress from there, rather than having to discard all of a transaction's work so far. This readily accomplished using (closed) nesting: just initiate and enter a new child transaction at each checkpoint position desired. In the face of conflict, one aborts as many of these transactions as necessary to roll back to a place before the conflict. Of course any committed open nested transactions need to be undone using their undos, etc. Checkpointing is also related to the next two applications, conditional atomic actions and sequential rollback.

Conditional Atomic Actions: Monitor-like constructs, including stylized use of Java synchronized objects, often include operations that conditionally wait, but otherwise execute atomically. A typical case is a communication queue, such as a circular buffer. An insert operation needs to wait if the limited buffer space is currently full, and a remove operation needs to wait if the buffer is empty. Open nesting makes it easy to implement buffer insert and remove with minimal conflicts, but waiting does not fit transaction models. Harris and Fraser [1] included in their design a form of *conditional* atomic action, what we might call a conditional *transaction*, that conditionalizes execution of a statement *S* on the truth of a predicate expression *p*. One way to think of how this works is that we use a transaction; call it T. T first evaluates *p*. If *p* is true, T proceeds to execute *S* and then T commits. If, in the first step, T finds *p* to be false, it aborts, and retries from the beginning. As stated, this would result in busy waiting. Harris and Fraser observed, though, that it makes no sense to retry T before some other conflicting transaction commits. Thus, we should perhaps suspend T right after it discovers *p* to be false, but give T a very low priority so that any conflicter will win the conflict and cause T to abort, which is the signal for it to retry. If there is no conflict for some time, we can abort T and the thread scheduler can use a back off algorithm to allow T to check, less and less frequently, whether its awaited condition is true.

Since a conditional atomic action can be embedded in a larger transaction, the beginning of the test of the predicate *p* is like a checkpoint.

Sequential Rollback: Suppose a transaction T performed two open nested transactions one after another, *A* and *B*. Suppose it also did some non-nested work before, in between, and after them; we call that work *a*, *ab*, and *b*, the overall order being *a*, *A*, *ab*, *B*, *b*. Note

that *a*, *ab*, and *b* add to T's read- and write-sets, while *A* and *B* add to its commit, abort, and completion action lists. The protocol we gave earlier was to undo *B* and then *A*, running their abort actions in reverse order, then their completion actions in reverse order. Lastly we drop T's read- and write-sets. While this should usually work, we can imagine situations where it would be better to undo in this order: *b*, *B*, *ab*, *A*, *a*. We can accomplish these semantics by inserting a checkpoint in T immediately after each open nested action. Thus, T directly incorporates *a* and *A*, then its child T_1 directly incorporates *ab* and *B*. Finally, a child T_{11} of T_1 incorporates *b*. If we abort at this point, T_{11} will abort, restoring memory word state to right after *B*. Then we will undo *B* and after that undo *ab*'s word state changes, which gives us the state right after *A*. We then undo *A* and after that undo the writes of *a*.[1] Under what conditions we need this sequential roll back of memory level state is something we need to study further. We are also not sure whether it would tend to impose any significant performance penalty if done as a matter of course.

VIII. Summary

We have offered an exposition of the semantics of open nesting and how it can support serializable (safe) execution with higher concurrency than non-nested or closed nested execution. We have advanced the description of how to support open nesting in the language run-time system and hardware. We also described additional applications of nesting, and some useful, though not serializable, applications of open nesting.

IX. Concerning Related Work

The topic of nested transactional memory is quite new and relatively unexplored. If a reviewer is aware of other work on the topic, we will be pleased to hear of it! We hope the works we have cited along the way provide adequate useful background and reference to what has come before—we indeed stand on the shoulders of giants.

X. Acknowledgments

This material is based upon work supported by the National Science Foundation under grant number CCR-0085792 and by Intel Corporation. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the sponsors. We further acknowledge ongoing collaboration on nested transaction memory models with Bradley Kuszmaul, Charles Leiserson, Gideon Stupp, and James Sukha, particularly the notion of linear nesting.

References

[1] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proceedings of the 2003 ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, ACM SIGPLAN. Anaheim, CA: ACM Press, Oct. 2003, pp. 388–402.

[2] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, ser. Data Management Systems. Morgan Kaufmann, 1993.

[3] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*. Cambridge, MA: M.I.T. Press, 1985.

[4] ——, "Nested transactions: An approach to reliable distributed computing," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, Apr. 1981, also published as MIT Laboratory for Computer Science Technical Report 260.

[5] ——, "Nested transactions: An approach to reliable distributed computing," in *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*. Pittsburgh, PA: IEEE, July 1982, pp. 33–39.

[6] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: Model and preliminary architecture sketches," presented at the 2005 Workshop on Synchronization and Concurrency in Object Oriented Languages (SCOOL '05), held at OOPSLA, October 2005, San Diego, CA (no proceedings). Submitted for publication. Draft available at http:www.cs.umass.edu/~moss/scool2005.pdf.

[7] J. E. B. Moss, N. D. Griffeth, and M. H. Graham, "Abstraction in recovery management," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. Washington, D.C.: *ACM SIGMOD Record 15*, 2 (June 1986), May 1986, pp. 72–83.

[8] C. Beeri, P. A. Bernstein, and N. Goodman, "A model for concurrency in nested transactions systems," *Journal of the ACM*, vol. 36, no. 2, pp. 230–269, Apr. 1989.

[9] G. Weikum and H.-J. Schek, "Concepts and applications of multilevel transactions and open nested transactions," in *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992, pp. 515–553.

[10] P. Chrysanthis and K. Ramamritham, "Synthesis of extended transaction models using ACTA," *ACM Transactions on Database Systems*, vol. 19, no. 3, pp. 450–491, 1994.

[11] K. P. Eswaren, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notion of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976.

[12] V. Luchangco and V. J. Marathe, "Transaction synchronizers," presented at the 2005 Workshop on Synchronization and Concurrency in Object Oriented Languages (SCOOL '05), held at OOPSLA, October 2005, San Diego, CA (no proceedings).

[13] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204–213.

[14] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 2003, pp. 92–101.

[15] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, June 1981.

[16] M. Herlihy, "Apologizing versus asking permission: Optimistic concurrency control for abstract data types," *ACM Transactions on Database Systems*, vol. 15, no. 1, pp. 96–124, 1990.

[17] M. Rinard, "Effective fine-grained synchronization for automatically parallelized programs using optimistic synchronization primitives," *ACM Transactions on Computer Systems*, vol. 17, no. 4, pp. 337–371, Nov. 1999.

[18] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, "Efficient optimistic concurrency control using loosely synchronized clocks," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 23–34, June 1995.

[19] S. Jagannathan and J. Vitek, "Optimistic concurrency semantics for transactions in coordination languages," in *Coordination Models and Languages*, ser. Lecture Notes in Computer Science, vol. 2949, 2004, pp. 183–198.

[20] C. Pedregal-Martin, "Transaction recovery in databases and beyond," Ph.D. dissertation, University of Massachusetts, Amherst, MA, Sept. 2002.

[21] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the International Symposium on Computer Architecture*, 1993, pp. 289–300.

[22] M. P. Herlihy and J. E. B. Moss, "Lock-free garbage collection on multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 3, pp. 304–311, May 1992.

[1] We first heard of this approach from Charles Leiserson.