Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 63 (2006) 186–201
Science of Computer Programming
www.elsevier.com/locate/scico

Nested transactional memory: Model and architecture sketches

J. Eliot B. Moss[a,*], Antony L. Hosking[b]

[a] *Department of Computer Science, University of Massachusetts, Amherst, MA 01003-9264, USA*
[b] *Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA*

Received 28 November 2005; received in revised form 14 May 2006; accepted 18 May 2006
Available online 14 August 2006

Abstract

We offer a reference model for nested transactions at the level of memory accesses, and sketch possible hardware architecture designs that implement that model. We describe both closed and open nesting. The model is abstract in that it does not relate to hardware, such as caches, but describes memory as seen by each transaction, memory access conflicts, and the effects of commits and aborts. The hardware sketches describe approaches to implementing the model using bounded size caches in a processor with overflows to memory. In addition to a model that will support concurrency within a transaction, we describe a simpler model that we call *linear nesting*. Linear nesting supports only a single thread of execution in a transaction nest, but may be easier to implement. While we hope that the model is a good target to which to compile transactions from source languages, the mapping from source constructs to nested transactional memory is beyond the scope of the paper.

Keywords: Nested transactions; Transactional memory; Closed nesting; Open nesting

1. Motivation

Recently there has been increased interest in incorporating some notion of atomic actions in widely used modern programming languages, particularly Java (as done by Harris and Fraser [7] for example). It is not our purpose here to sing the praises of atomic actions; we take for granted that they are desirable. However, for atomic actions to be adopted as an approach for widespread use, several problems remain to be solved. Here we begin to address one of those problems—scale—in two manifestations.

A number of previous authors have observed limitations of early hardware transactional memory designs such as that of Herlihy and Moss [8], namely that they do not support transactions of unbounded size. In the part of this paper in which we offer architectural sketches, we offer one approach to this issue as a contribution to the ongoing discussion in the community as to how to solve this problem (see Sections 3.1 and 3.2).

Our primary focus, however, is on a different scaling problem, namely the need for *nesting* of transactions in large concurrent systems. Here are the primary reasons that nesting is essential:

* Corresponding author.
E-mail addresses: moss@cs.umass.edu (J.E.B. Moss), hosking@cs.purdue.edu (A.L. Hosking).
URLs: http://www.cs.umass.edu/~moss (J.E.B. Moss), http://www.cs.purdue.edu/homes/hosking/ (A.L. Hosking).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.05.010

(1) Libraries will contain atomic actions, and user code or other libraries will need to group these within larger atomic actions. Thus, one must be able to *compose* code that uses actions. This can be accomplished in a simple way, as exemplified in the design of Harris and Fraser [7], by just grouping all nested actions within their enclosing top-level action. While this "works", it does not address the other reasons for nesting.
(2) Large monolithic actions increase the volume of work that must be done if a transaction needs to be rolled back. Closed nesting [11–13] (discussed further below) at least partially addresses this concern. It allows each new portion of work to be attempted, and possibly to fail, without necessarily aborting work already accomplished. It can also be implemented so as to allow a program to control its response to the abort of a portion of work, e.g., to attempt the work in a different way. (This can be particularly helpful in distributed systems: if one resource is unreachable or too busy, one can sometimes redirect to a suitable alternate.)

    Similarly, clients of opaque library code that suffer performance penalties because of failing nested transactions in library code may need some way to program around those performance bottlenecks.

    In summary, while grouping subtransactions into large monolithic transactions is a correct response to supporting execution of nested transactions, the performance and functionality implications of subtransaction failure may be important enough that programmers will expect to be able to program with subtransactions and respond to their failure.
(3) Large monolithic transactions limit concurrency. Compared with locking, using transactions over the same objects improves concurrency in two ways. One is that transactions can proceed concurrently without conflict if they access different data, e.g., different fields of the same object or different objects. The other is that transactions distinguish between reading and writing, and concurrent reads do not conflict. Locks demand mutually exclusive access even for reads. *Open nesting* is intended to improve concurrency further. It can also be used (carefully) as a kind of "escape hatch" from the strictures of transactional semantics, if absolutely needed.

1.1. Closed nesting

Space precludes an extended discussion of nested transactions, but we offer a quick summary here, which also serves to introduce terminology. We use the model of Moss [13]. A transaction is either *top-level*, which is similar to a traditional non-nested transaction, or is nested within a *parent* transaction. This nesting gives rise to trees of transactions, on which we use family relationship terminology, such as parent, child, ancestor, descendant, sibling, etc. We also use the term *subtransaction* for a child. In this model, only transactions with no current children can access data; such accesses are either reads or writes. Transactions accumulate read and write sets, which determine conflicts as well as what data need to be updated upon commit. Logically, the (globally committed) data are not updated until a top-level transaction commits. When a transaction reads a value, it sees the value in its own read or write set (if there is one), otherwise the value seen by its parent. A top-level transaction will see the latest (globally committed) value.

A desired operation by a given transaction conflicts with an operation on the same datum by another transaction if: at least one of the accesses is a write, and neither transaction is an ancestor of the other.

When a nested transaction commits, its read and write sets are unioned with those of its parent. When a top-level transaction commits, its writes become permanent. When a transaction aborts, its read and write sets (and associated tentatively written values) are simply discarded.

It is possible to show that these rules lead to a semantics of nested transactions in which a legal execution is equivalent to a serial execution of the committed transactions (only), in some order, i.e., serializability.

1.2. Open nesting

Open nesting[1] allows further increases in concurrency, by releasing concrete resources earlier and applying conflict detection (and roll back) at a higher level of abstraction. For example, transactions that increment and decrement a shared integer would normally conflict, since they write a shared datum. But, since increment and decrement commute as abstract operations (assuming one is not observing the actual value involved), they can be correctly implemented

[1] Moss et al. [15], for example, offer a description and justification of open nesting (multi-level transaction) concurrency control and recovery, and a proof of its correctness.

with open nesting. An increment (say) would do read, add one, write. The open nested action would be over and the updated field would not be part of the parent transaction's read or write set. However, if the parent later aborted, we would need to run a compensating decrement to remove the logical effect of the committed open nested action.

Other applications of open nesting include: highly concurrent indexes and collection data structures (B-trees, hash tables, etc.), memory allocation and garbage collection primitives (including concurrent garbage collectors), scheduler queue operations, and coping with "external" activities such as input/output.

The only difference between open and closed nesting in terms of a read/write set execution model concerns what happens when a transaction commits. When an open transaction commits, we discard its read set, and commit its writes *at "top level"*. We also remove the written data elements from the read and write sets of all other transactions. (Given the conflict rules, these can only be ancestors of the committing transaction.)

We assume that additional mechanisms handle recording of any necessary compensating actions, and applying them in case an ancestor transaction aborts. In practice, some of an open nested action's writes would update a list of compensating actions held by the action's parent. The details are more of a language and run-time system design issue than one of this model. Likewise, we assume additional mechanisms deal with expressing and checking abstract conflicts. These will be similar to the lock tables of transaction managers. The locks may be on somewhat abstract entities as opposed to language or memory objects, and one might choose to offer richer schemes of locking modes. Again, some of an open nested action's reads would be checking this abstract lock table, and some of its writes would be updating the table as necessary. These details properly belong at the language and language implementation level.

It lies beyond the scope of this paper to consider how to state or prove some kind of serializability theorem about open nesting. (It is far from obvious how to claim equivalent execution, for example.)

2. Nested action execution model

The descriptions above were intentionally brief and admittedly vague, intended only to give a rough sense of what closed and open nesting actually mean. We now offer a detailed model for executing nested transactions, closed and open. Its purpose is partly to *define* the semantics of nesting (as understood at the level of memory reads and writes), and partly to be a standard against which to judge the correctness of proposed hardware designs. We take correctness of this model as a given—but as already pointed out, it stands in need of further formal justification itself, such as a suitable serializability theorem.

2.1. System states

We describe execution in terms of *system states* and transitions from one system state to another. A system state includes all the "globally committed" memory state, plus memory state (read and write sets) for each transaction currently live in the system.

Transactions: First, we assign each transaction a unique number, a positive integer distinct from any other transaction currently in the system. Transactions start *live*, and may later *commit* or *abort*, at which time they are no longer represented in the system state. It turns out to be convenient to use transaction number 0, but it is associated with the globally committed state, not with any actual transaction. We use T to designate the set of legal transaction numbers, including 0.

Every non-zero transaction t has a unique *parent transaction*, *parent*(t), and the parent relationship is acyclic. *Top-level transactions* are those whose parent is 0. The *children* of transaction t are those transactions whose parent is t. Thus, transactions are organized into a single tree with transaction 0 as the root. A transaction with no (live) children is said to be *running* and is allowed to (attempt to) issue memory reads and writes.

Memory: The system's memory state is a total map from memory locations L to memory values V. (One can also think of it as an array indexed by L, etc.) We could extend this model to handle extension and contraction of the available virtual memory addresses (e.g., as performed by `mmap` and `munmap`). There are interesting questions as to the units to use for V, since it may be natural to do most processing in terms of cache lines, but cache lines are generally rather larger than the smallest addressable unit and thus using them for V can induce false access conflicts.

System state: A system state is a subset of $\Sigma = T \times L \times Boolean \times V$ such that no transaction has more than one entry for a given location, and transaction 0 has an entry for every location. The *entries* include a value drawn from V and a boolean *written* flag. The written flag indicates whether the transaction wrote, or only read, the value.

More formally, $S \subset \Sigma$ is a system state if:

$$\forall l \in L : \exists v : (0, l, true, v) \in S$$

and

$$(t, l, w, v) \in S \wedge (t, l, w', v') \in S \Rightarrow w = w' \wedge v = v'.$$

An *initial system state* has entries only for transaction 0, i.e., it consists only of globally committed memory state.

2.2. Additional useful definitions

Here are a few extra definitions we find useful.

$$ancestors(t) = \text{ if } t = 0 \text{ then } \{\} \text{ else } parent(t) \cup ancestors(parent(t)).$$

We likewise define *descendants* (formal version omitted). The definitions of *parent*, *ancestors*, *children*, and *descendants* make sense only for states in which the transaction is live, and, for *children* and *descendants*, may vary from state to state. We make the state explicit as needed for precision.

We define the *ReadSet*, *WriteSet*, and *LocSet* of transaction t in state $S \in \Sigma$:

$$ReadSet(S, t) = \{l \mid (t, l, false, v) \in S\}$$
$$WriteSet(S, t) = \{l \mid (t, l, true, v) \in S\}$$
$$LocSet(S, t) = ReadSet(S, t) \cup WriteSet(S, t).$$

We define the value of location l for transaction t in state S:

$$ValueOf(S, t, l) = \text{ if } \exists (t, l, w, v) \in S \text{ then } v \text{ else } ValueOf(S, parent(S, t), l).$$

Note that *ValueOf* is well-defined because transaction 0 defines a value for every location.

2.3. Allowed reads and writes

In state S, transaction t can read location l, notated $CanRead(S, t, l)$, if for each $(t', l, w', v') \in S$ at least one of the following is true:

(1) $t' = t$ (myself), or
(2) $t' \in ancestors(S, t)$ (an ancestor's value), or
(3) $w' = false$ (the other action is a read).

Similarly, $CanWrite(S, t, l)$ if for each $(t', l, w', v') \in S$ one of the following is true:

(1) $t' = t$ (myself), or
(2) $t' \in ancestors(S, t)$ (an ancestor's value).

2.4. Effect of reads and writes

Suppose $CanRead(S, t, l)$. Then if t reads l in state S, we obtain a new state S'. If $l \in LocSet(S, t)$, then $S' = S$. Otherwise $S' = S \cup \{(t, l, false, ValueOf(S, t, l))\}$. That is, if t already has a value for l, then S is unchanged; otherwise, S' consists of S plus an entry showing that t has read $ValueOf(S, t, l)$ for location l.

We make a minor modification to the effect of a read performed by an *open* nested action: it sets *written* to be true if any ancestor (i.e., any transaction) has written the value. That is, rather than *false* above, an open nested action has:

$$\exists (t', l, true, v') \in S$$

Our purpose is to insure that when the open nested action commits, it globally commits the latest value.[2]

[2] Fine points such as these are exactly where we need a serializability characterization and theorem, to verify that we have them right.

Suppose $CanWrite(S, t, l)$. Then if t writes value v to l in state S, we obtain a new state $S' = (S - old) \cup \{(t, l, true, v)\}$, where $old = \{(t', l', w', v') \in S \mid t' = t \wedge l' = l\}$. In words, a write deletes any previous value for location l and transaction t, replacing it with the value written and a written flag that is *true*.

Comment: If $CanRead(S, t, l)$ (resp. $CanWrite(S, t, l)$) is false, but the next action t wants to perform is the read (write), then an implementation might delay the action, in hope that a conflicting transaction will commit. This policy can deadlock. Alternatives are to abort either the requesting transaction or all conflicting transactions.

2.5. Semantics of commit and abort

Suppose t is *open nested* and commits in S, which is allowed only if t has no children in S. The new state $S' = (S - old_t - old_w) \cup new_0$, where:

$$old_t = \{(t', l', w', v') \in S \mid t = t'\}$$
$$old_w = \{(t', l', w', v') \in S \mid l' \in WriteSet(S, t)\}$$
$$new_0 = \{(0, l, false, v) \mid (t, l, true, v) \in S\}.$$

In words, when an open nested action commits, it updates the globally committed state for words that t wrote, and forces drops from *all* transaction states of every word *written* by t. The first effect is the global commit effect of an open nested transaction. The second effect achieves the "breaking" of dependences induced by open nested action commits.

Comment: We observe a fine point here. The exact definition of what constitutes a "word", i.e., the smallest separately writable unit of memory state, matters. Using a larger unit (a wide cache line) with closed nested actions will lead to false conflicts, which hurts performance but not consistency. Using a large unit with *open* nested actions actually changes behavior and can cause more values to commit globally than otherwise would. Thus, in the presence of open nesting, each minimally writable unit ("word") needs its own *written* bit, even if the words are grouped into large units in a cache.

When an open or closed nested transaction *aborts*, which is allowed only when t has no children, the new state simply drops all entries associated with the aborting transaction t:

$$S' = S - \{(t', l', w', v') \in S \mid t = t'\}.$$

When a closed nested transaction t commits, which is allowed only when t has no children, its parent "inherits" its state:

$$S' = (S - old) \cup new$$

where:

$$old = \{(t', l', w', v') \in S \mid t' = t \vee (t' = parent(t) \wedge l' \in LocSet(t))\}$$
$$new = \{(parent(t), l, l \in (WriteSet(t) \cup WriteSet(parent(t))), v) \mid (t, l, w, v) \in S\}.$$

Here the expression $l \in (WriteSet(t) \cup WriteSet(parent(t)))$ indicates the conditions under which the *written* flag will be true after the commit, namely if either t or its parent wrote the location.

2.6. Discussion of the semantics

The locking and commit rules of this design correspond to those of Moss [13] for closed nesting. To our knowledge, open nesting has never been described at this level, so we feel further discussion is called for. If we consider open nesting as described by Moss et al. [15] (hereafter MGG), it includes two things that glue together effects and meaning at different levels of abstraction. Fundamental to the MGG model, the series of steps in an open nested action corresponds to some *abstract* action at the next higher level of abstraction. For example, a low level read, modify, and write may implement the abstract action *"increment"*. If the open nested action commits, the higher level of abstraction undertakes an obligation: it must undo the effects of the increment if the higher level action aborts. In this case, it would undo the increment by running a decrement. If the whole action aborts before the open nested action commits, then we just discard the read and write sets of the nested action, etc.

How would this model of action undo look in our memory-level model? As previously suggested, the higher level action will keep a (private) list of undo actions it needs to perform if it will abort. The open nested action on a given counter c adds a note to that undo list, recording the need for a decrement of c should the higher level action abort. The actual steps of adding this note to the private undo list can be part of the open nested action.[3]

The second way in which the two levels are glued together is that abstract operations require abstract concurrency control. This will have to be explicit and is clearly more an issue of programming language and run-time system design. Abstract locking code will examine a locking data structure, within the open nested action. If the abstract operation can proceed, the open nested action simply proceeds. If it cannot proceed, the action might abort or express a need to wait (a feature requiring further exploration, which we do not consider here). As with updating the undo list, the open nested action updates the abstract locking data structure. In the case of our increments and decrements, there are no concurrency conflicts: the operations freely commute under all circumstances, so there is no need to lock.

In all cases, that is for updates to the affected objects themselves, for additions to the undo list, and for inspections and updates of abstract locking data structures, when an open nested action commits it is appropriate to commit its memory updates and to release its read sets. It is as if it were a globally committed top-level transaction. The model we give here does exactly that.

What the MGG model does not tell us very well is the most appropriate way to handle cases where a parent and open nested child access some of the same memory words, particularly if these words can be accessed by other transactions. Such accesses are in some sense "unstructured": they cross levels of abstraction and may tend to "break" abstraction. The model here essentially removes dependence edges between past and future actions on updated memory words when an open nested action commits. This seems the best thing to do, since one can always effectively induce more dependences by coding suitable abstract locks.[4]

3. H/W sketch 1: Using cache

We first observe that the definition of the nested transaction execution model says nothing about caches. It concerns itself with the *view* of memory state that each transaction observes, and the *effects* on memory state that the transaction has (if it commits). Logically, caching is orthogonal to transaction semantics. However, since transaction state (as opposed to memory state) is inherently ephemeral, and since it arises from the issuing of reads and writes by running transactions, it is natural to expect that transaction state (read and write sets) will generally be cached. Further, "globally committed" state needs no mention of transaction ids, which matches well with the notion of main memory containing that state: it needs no additional tags, etc., associated with transactions. Caches naturally have address tags in addition to the data they contain, so it is natural to think of extending the tags to include transaction ids (rather like address space ids in some virtual memory translation schemes). Since we use transaction 0 to represent committed data, it is easy for a cache with transaction tags to hold such data. The overflow of transaction state to main memory is less convenient, and requires a table probed by some combination of hardware, firmware, or software.

3.1. Hardware and memory tables

Our first hardware sketch is simple, though perhaps somewhat unrealistic. It consists of three hardware tables (caches), plus overflow data structures. The first table is a fully associative cache of transaction state entries, which we call the *Transaction Data Cache*. Each entry includes these fields, illustrated in Fig. 1:

Transaction id: The id of the transaction to which this memory word belongs. A 0 in this field indicates a globally committed value.
Address: The address of the memory word being cached.
Data: The actual data value.

[3] Here we are hoping that no concurrent action tries to add something to the list. If access conflicts on the list were common enough to warrant, the parent action could create a new list for each open nested action it invokes, and thus avoid the conflicts.

[4] A little while after the workshop in which we originally presented this work [14], we became convinced otherwise. If an ancestor holds a word in a read or write set, and a nested open action writes that word, if the nested action commits it should update the value in all ancestors that hold that word, without changing which sets hold the word. This will produce less surprise and appears to lead to better semantics at the programming language level.

Valid bit: Indicates whether the other fields are valid. An invalid entry is available for reuse.

Dirty bit: When set, indicates that this entry is not recorded in the main memory overflow table. For transaction 0, this indicates that the cached value may differ from that stored in memory.

Written bit: Indicates whether the value has been (logically) written by the transaction.

Ancestor written bit: This caches the fact that some (non-0) ancestor of this transaction holds a written copy of this word.

Obtained-from transaction id: Indicates the transaction from which we read this value (the youngest ancestor with an entry for it).

Writing transaction id: If some descendant (would have) read this copy of the word, and has updated it, this field records the descendant transaction's id.

Reader count: The number of live descendants that have read this copy of this word.

The first six fields are straightforward; the use of the remaining four fields will become more clear as we go. The Writing transaction and Reader count fields will never be used at the same time, so we can use one bit in the field to indicate when it holds Writing transaction versus when it holds Reader count.

Fig. 1 illustrates multiple entries for a single word (address 100). The transaction 0 entry holds the latest fully committed value. There is an ancestor-to-descendant chain consisting of transactions 14, 16, 23, and 25. Transaction 25 has two subtransactions, 26 and 28. Transactions 16 and 25 have written values, which their children/descendants have read. All these entries are Valid, and we assume no overflow so the Dirty bits are 1. The Written and Ancestor written values should be obvious, as well as the obtained-from transaction id (From). The parent of a writer indicates the writing child (W *id*); other transactions give a reader count (R *cnt*).

Tid	Addr	Data	V	D	W	Anc W	From	Writing Txn / Read Count	
0	100	532	1	1	0	0	0		0
14	100	532	1	1	0	0	0	W	16
16	100	178	1	1	1	0	14	R	1
23	100	178	1	1	0	1	16	W	25
25	100	393	1	1	1	1	23	R	2
26	100	393	1	1	0	1	25	R	0
28	100	393	1	1	0	1	25	R	0

Fig. 1. Transaction data cache, sketch 1.

The second table consists of child-parent pairs for live transactions. We call it the *Transaction Parent Cache* since it provides the means to discover the id of a transaction's parent. In addition to the child and parent transaction ids, it provides a *Valid* bit for each entry, a *Dirty* bit to indicate that the entry is not (yet) represented in memory tables, an *Overflowed* bit that indicates that the transaction has overflow entries in memory, and an *Open* bit to indicate whether the transaction is open or closed nested (conventionally *closed* for top-level transactions and transaction 0). Fig. 2 illustrates some parent cache entries, for the transactions in Fig. 1. We assume no overflows and that the parent entries have not been written to memory tables.

Parent	Child	Valid	Dirty	Ov	Open
0	14	1	1	0	0
14	16	1	1	0	0
16	23	1	1	0	1
23	25	1	1	0	0
25	26	1	1	0	0
25	28	1	1	0	0

Fig. 2. Transaction parent cache, sketch 1.

The last table summarizes overflows from the Transaction Data Cache (first table) to memory, so we call it the *Overflow Summary Table*. There are many ways it could be organized. For present purposes, we consider it to be a bit vector indicating which hash codes have overflow entries, where a hash code is some hash of the address of a word. The key point is that if the bit corresponding to the hash code for a given address is 0, then there are no overflow entries for that address. If the bit is 1, then some address with that hash code has an overflow entry (see Fig. 3, which illustrates no particular situation). At additional cost we could maintain a vector of these bits for each transaction as well, and cache some number of those vectors to reduce probes to in-memory tables. There are many strategies and we are not here advocating any particular one; Rajwar et al. [18] have explored some specific schemes. Note that the novelty of our proposal lies in its handling of nesting, not in that it handles overflows.

0	0	0	0	0	1	1	0	1	0	0	1	0	0	0	1

Fig. 3. Overflow summary table, sketch 1: one bit per hash set.

We envision that Transaction Data Cache overflows go into a hash table in main memory, accessed by firmware or fast trap handlers. The entries in main memory are chained by transaction id (so that commit and abort can find efficiently all overflows pertaining to the completing transaction) and by address (to make it easy to find other entries for the same address).

The Transaction Parent Cache might overflow as well (we prefer not to place an arbitrary bound on the number of transactions). If a (live) transaction has no entry in the parent cache, then its entry must be in an overflow table in main memory. We further support the notion of a transaction being flushed from the caches. In that case, *all* of its data entries are in the main memory overflow table.

Optionally, we may prefer to use short transaction ids in the caches while allowing a larger space of transaction ids. This is similar to using address space ids for currently running processes, while allowing a larger space of process ids. In that case, in addition to the tables above, we need a mapping from short ids to long ids, to support probing the main memory structures, which should use long ids. Using short ids saves bits in the hardware, with the added complexity of occasionally requiring the extra mapping step. From here on we assume we are using short ids in the caches.

3.2. Using the tables

It seems easiest to describe how to use the tables by sketching how each interesting operation affects them.

Creating a transaction: We obtain a free Transaction Parent Cache entry and an unused (short) transaction id. Any clean parent cache entry can be dropped. To free a dirty one, we must force the entry to the memory table, cleaning it. We can then drop it.

We can imagine several ways to obtain a currently unused short transaction id. One is to keep a bit vector of used/free short ids. A priority encoder can then produce a free one, or indicate that they are all in use. Alternatively, one can generate ids (e.g., randomly or sequentially) and probe to see if the id is in use. If no short ids are available, or probing fails to reveal a free one within some threshold number of probes, then we evict (flush) a transaction from the cache.

To do that we first force to memory all dirty parent cache entries that mention the transaction as child or parent, and then drop them. Then we force to the data cache overflow table every dirty Transaction Data Cache entry that mentions the transaction, i.e., the transaction's own entries and those that mention it as a writer or obtained-from transaction. We then drop those entries from the data cache. Finally we drop the short id from the short-to-long id map. We can then use the short id for the new transaction.

Note that when we write a dirty parent cache entry to memory, either or both transactions may not yet have long ids assigned. We assign them from a counter of sufficient length to prevent wraparound (or otherwise prevent duplicates), and update the short-to-long id map. We further note that loading a Transaction Data Cache entry or Transaction Parent Cache entry from memory may require allocating one or more short ids.

Once we have the necessary transaction ids and a free parent cache entry, we enter the (child, parent) pair into the parent cache, setting the entry valid and dirty.

Reading a word: If the cache contains an entry for the issuing transaction and address, we simply return the data of the entry. Otherwise, we have a *data cache miss*. In the case of such a miss, we check the Overflow Summary Table, and if

it is possible for the entry to be in the overflow area, we probe there. If we find no entry for the requesting transaction, we determine its parent and probe the parent, first in the cache, and then in memory (if necessary). Assuming there is no conflict, and there is room in the cache for any new entries required, here are the cases that arise:

- *We find no entry at all:* We obtain the value from (globally committed) main memory and create two entries for the data read, one for transaction 0 and one for the reading transaction. The transaction 0 entry is marked valid and not dirty. Its Written and Ancestor written flags don't matter (but it is probably simpler if they are set to 0). Its Obtained-from id field does not matter. We set its Reader count to 1. The reading transaction's entry we set to valid and dirty. We set Written and Ancestor written to 0, Reader count to 0, and Obtained-from to (transaction) 0.
- *We find an entry for some ancestor:* If the entry is in the memory overflow table, it is probably easiest to pull it into the cache, though one could access/update it in memory. We create a new entry in the cache for the current transaction, using the data of the entry we found. We set our new entry valid and dirty. We set Ancestor written to 1 if either Written or Ancestor written are 1 in the ancestor. We set Written 1 if we just set Ancestor written 1 and this is an open nested action. We set Reader count to 0. (See **Writing a word**, below, for further discussion of the case where written becomes set.) Otherwise, we increment the Reader count of the ancestor's entry. In any case, we set Obtained-from to be the ancestor transaction.
- *We find an entry for this transaction:* We simply return the data.

Note: Whenever we change an ancestor's Transaction Data Cache entry, we set Dirty to 1 in that entry (to show that it may differ from any overflow table copy).

Freeing Transaction Data Cache entries: If we need a data cache entry and none are available, we can drop any clean entry. We can clean an entry by writing to overflow memory and insuring: (a) that the entry's hash code's bit is set in the Overflow Summary Table; and (b) the transaction's Overflow bit is 1 in the Transaction Parent Cache. As with creating a transaction, writing an entry to memory may involving acquiring a long transaction id.

Determining conflicts with reads: When we are reading, if we find no (non-0) ancestor entry, then there is no conflict. If we find an ancestor entry that contains a Reader count, there is also no conflict. If we find an ancestor entry with a Writing transaction id, we conflict with that writing transaction. (It cannot be us, or we would have found the entry, and since we probe from younger to older ancestors, the writer cannot be our ancestor. Hence it conflicts.)

Writing a word: If the cache contains a Written entry for the issuing transaction and address, we simply update that entry. Otherwise, we have more work to do. These are the cases, assuming there are free entries as needed and no conflict:

- *We find no entry at all:* As for reading, we obtain the value from (globally committed) main memory and create two entries for the data, one for transaction 0 and one for the writing transaction. We set up the transaction 0 entry as in the case of reading, with the exception that instead of a Reader count, we indicate the current transaction as the Writing transaction. The writing transaction's entry we set to valid and dirty. We set Written to 1 and Ancestor written to 0, Reader count to 0, and Obtained-from to (transaction) 0. We set the data field to have the new value.
- *We find an entry for some ancestor:* We create a new entry in the cache for the current transaction, setting its data field to the value written, and marking the entry valid and dirty. We set Ancestor written to 1 if either Written or Ancestor written are 1 in the ancestor. We set Written to 1. We set Reader count to 0. We set the ancestor's entry's Writing transaction field to be the current transaction, and the current transaction's Obtained-from field to the ancestor.
- *We find an unwritten entry for this transaction:* We set Written and Dirty to 1 and update the data field. We further find the youngest ancestor with an entry for the same address, and set its Writing transaction entry to be the current transaction.

Determining conflicts with writes: In these situations, there is no conflict: (1) we have written the word already; (2) we have an unwritten entry and *all* our ancestors have Reader count 1 or record a Writing transaction (we are the only reader and one or more ancestors are writers); (3) we have no entry, our youngest ancestor (if any) has a Reader count of 0, and any further ancestors have a Reader count of 1 or record a Writing transaction.

In these situations there is a conflict: (1) we have an unwritten entry and some ancestor has a Reader count greater than 1; (2) we have no entry and our youngest ancestor has a Reader count greater than 0 or records a Writing transaction.

Aborting a transaction: For each unwritten entry of the transaction, decrement the Obtained-from transaction's Reader count for this entry. For each written entry of the transaction, set the Obtained-from transaction's Reader count to 0. Invalidate all entries of the aborting transaction. If the transaction overflowed, process its overflowed entries, updating Reader counts. If the parent cache entry is not dirty, then remove it from the memory tables. Drop the transaction from the parent cache.

Note that if Obtained-from transaction entries have been dropped from the cache, we need to restore them or update them in the overflow table.

Committing a top-level transaction: For each unwritten entry, proceed as for aborting (transaction 0 will already have an equivalent entry). For each written entry, restore the transaction 0 entry (if necessary), and update its value. Mark the entry dirty and written, and set its Reader count to 0. If the transaction has overflow entries, process them then delete them. Delete the transaction's parent cache entry, including memory copy (if any).

Committing a closed nested transaction: For each entry, if the parent has an entry, then *merge* the child and parent's entries. This means that the parent entry will have: Valid 1; Dirty 1; Written 1 iff parent, child, or both have Written 1; the child's data value. The parent entry maintains its Ancestor written value and Obtained-from transaction id. If the child was Written, set the parent's Reader count to 0. If the child was not Written, decrement the parent's Reader count.

If there is no parent entry, then just change the Transaction id to that of the parent.

Committing an open nested transaction: Proceed as for committing top-level transactions, with this difference for written entries: Drop all other (non-0) transaction's entries for the written entries. This must include any entries that have overflowed.

3.3. Discussion

This scheme is undoubtedly complex. A key source of complexity is the tracking of readers, writers, and obtained-from transactions, whose purpose is to speed up conflict checks and commit and abort processing. We must also provide for overflows, recording dirtiness, etc. We did not expect a design as simple as that of Herlihy and Moss [8], for example, but feel the need for something simpler than this.

3.4. Set associative and victim caches

The preceding assumes a more or less fully associative cache. If we (more realistically) use a set associative structure for the transaction data cache, plus possibly a victim cache (with a restriction not to hold more than one member of the same set), then we can simplify some of the processing. In particular, we can obtain all cache entries for the same address all at once, some in the set associative cache read-out buffer and one from probing the victim cache. We can handle the common cases of conflict and no-conflict in hardware and kick the rest out to firmware (as well as the overflow case). The logic may be simplified if we impose an ordering on the items in a set, such that ancestors will be in one direction and descendants in the other. Working a victim cache entry in is less pleasant.

4. Sketch 2: Tagging ancestors

Here is an approach that reduces the complexity of Sketch 1's transaction data cache. Sketch 2's transaction data cache is similar to Sketch 1's, but drops the Ancestor written bit, Obtained-from transaction id field, Writing transaction id field, and Reader count. In their place it has an *Ancestor bit*, which indicates that this entry is for an ancestor of the current transaction (or the current transaction itself). The notion of current transaction was implicit in Sketch 1. We implied that a processor acts on behalf of a current transaction at any given time. In Sketch 2 we make this more explicit: a change in current transaction will necessitate updating Ancestor bits.

For now, we assume associative access by transaction id, as well as access by address.

4.1. Using the tables

As before, we describe how each interesting operation affects the tables. Many steps are similar; we focus on the differences.

Creating a transaction: We obtain a free parent cache entry and an unused (short) transaction id, and fill them in, as in Sketch 1.

Reading a word: If the cache contains an entry for the issuing transaction and address, we simply return the data of the entry. If the Overflow Summary Table indicates possible overflow entries, we trap to firmware for memory table probing. The read has a conflict if there is any entry for the same address where a non-ancestor has written the word. This is now easy to check: address match, Ancestor bit 0, Written bit 1. As before we probe from youngest to oldest ancestor until we find an entry, and we fill in a new entry with the current transaction id, address, and data. We set Valid to 1, Dirty to 1, Written to 0, and Ancestor to 1. As before, if the current transaction is open nested, and any ancestor has Written 1 (easily checked with an associative access), we set Written 1. Unlike Sketch 1, there is no need to create a transaction 0 entry if there is not one. (Sketch 1 uses it to record readers and writers for conflict detection.)

Writing a word: If the cache contains a written entry for the issuing transaction and address, we simply update that entry. If the Overflow Summary Table indicates a possible overflow entry, trap to firmware. The write conflicts if there is any entry for the same word for a non-ancestor, easy to check with associative matching on this address. We probe from youngest to oldest ancestor, and fill in the new entry as when reading, but set Written to 1.

Aborting a transaction: We simply invalidate all Transaction Data Cache entries for the transaction, trapping to firmware if the transaction has overflowed. We clean up the Transaction Parent Cache as in Sketch 1.

Committing a top-level transaction: Drop unwritten entries of this transaction, and merge written ones with the transaction 0 entry. A merge in Sketch 2 has these effects: set parent Valid 1, Dirty 1, Ancestor 1, Written to logical "or" of Written of the merged entries, Data to the committing entry's data. Note: a transaction 0 entry will always have Ancestor 1 and Written 0.

Committing a closed nested transaction: For each entry of the committing transaction, if the parent has an entry, merge entries. If the parent has no entry, just change the transaction id to that of the parent. Note that all Ancestor bits for the parent remain 1.

Committing an open nested transaction: Proceed as for committing a top-level transaction, but drop all (non-0) transaction entries for *Written* data.

Switching transactions: When switching from one transaction to another, other than creating and entering a subtransaction, or completing (committing or aborting) a subtransaction and returning to its parent, we need to correct Ancestor bits. Suppose we are switching from transaction t_1 to t_2. Let t be their youngest common ancestor (which may be transaction 0). For t_1 and each ancestor of t_1 younger than t, we set the Ancestor bits of all its entries to 0. For t_2 and each of its ancestors younger than t, we set the Ancestor bits to 1. Given associative hardware, this will take time proportional to the tree distance between t_1 and t_2.

4.2. Illustration

We consider the transaction situation of Fig. 1 and show how it would appear in this cache; see Fig. 4. We assume that transaction 28 is the current transaction. Thus, in this case, only transaction 26 is not current.

4.3. Discussion

This seems much simpler than Sketch 1. There are still some complexities when committing nested actions, since the detailed steps depend on whether there is a parent entry, etc. The essential difficulty is that it appears that we need to iterate over the read and write sets. What we need is a way to commit a transaction that updates all affected entries in parallel (associatively). This leads to Sketch 3.

Tid	Addr	Data	V	D	W	Anc
0	100	532	1	1	0	1
14	100	532	1	1	0	1
16	100	178	1	1	1	1
23	100	178	1	1	0	1
25	100	393	1	1	1	1
26	100	393	1	1	0	0
28	100	393	1	1	0	1

Fig. 4. Transaction data cache, sketch 2.

5. Sketch 3: Fast commit

We augment the Transaction Data Cache of Sketch 2 with some fields to speed commit. The primary new field is *Watched transaction*, which indicates a transaction we will watch. If it commits, then we take special action. If it aborts, we clear the Watched transaction field. We also add back the *Obtained-from transaction* field.

Reading a word: The main effect we want here is for a closed nested transaction read to be inherited by its parent on commit, but only if the parent has no entry. We will use the Obtained-from field to achieve this effect. For a read by an open nested action that results in an unwritten entry, we set the Obtained-from field to 0.

Writing a word: We set the Obtained-from field as for a read. We also set the Watched transaction field of the youngest ancestor, i.e., the one from which we obtained the value. The value we put in the field is the id of the current transaction. If the transaction is open nested, we set the Watched field of *all* ancestors.

Committing a transaction: Whenever we commit, we will broadcast across the cache the committing transaction's id and its parent's id. Various cache entries take action according to how they match this information. We first consider entries of the committing transaction.

For open nested actions, we drop unwritten entries and mutate written entries to transaction 0. For closed nested actions, we mutate unwritten entries to the parent, unless the parent matches the Obtained-from field, in which case we drop the entry. For written entries we always mutate them to the parent.

Here is how other entries are affected. Ancestors of an open action that wrote will match in their Watched transaction field, and will drop their entries. The parent of a closed nested action will have the committing child's id in its Watched transaction field, and will itself match the parent id being broadcast, and will drop its entry. A non-parent ancestor of a committing nested action will match in the Watched transaction id, but the broadcast parent id will not match its own id. Its response is to set its Watched transaction id to the broadcast parent id.

Aborting a transaction: In addition to invalidating the transaction's own entries, we set to 0 any Watched transaction field that matches the aborting transaction. In the case of an aborted open nested action, we need to reconstruct the previous Watched transaction entries. This might be rather painful. If we want the abort case to run faster, then as the open nested writes occur, we could flush ancestor entries to main memory (perhaps via a buffer designed for the purpose). If the open nested action commits, we'll just drop the entries, but if it aborts, we logically restore them.

Switching transactions: We need no additional logic.

5.1. Illustration

Again, in Fig. 5, we show the cache for the situation of Fig. 1.

5.2. Discussion: Set associativity

This scheme seems definitely more workable. The biggest question is the heavy use of associative logic, which tends to be bulky, slow, and power hungry. Can we make common operations faster? Assuming that commit and abort are rare compared to read and write, we offer a modification that exploits the statistics. We keep parallel associative logic for transaction commit and abort. However, it need not complete as fast as a read or write; it might take multiple

Tid	Addr	Data	V	D	W	Watch	From
0	100	532	1	1	0	16	0
14	100	532	1	1	0	16	0
16	100	178	1	1	1	0	0
23	100	178	1	1	0	25	16
25	100	393	1	1	1	0	23
26	100	393	1	1	0	0	25
28	100	393	1	1	0	0	25

Fig. 5. Transaction data cache, sketch 3.

cycles. For reading and writing, we organize the cache set associatively, perhaps with a victim cache, so that all entries we need to examine and manipulate can be read into buffers at once. We can then use a small amount of associative logic attached to those buffers to do the entry manipulations needed for reading and writing.

6. Linear nesting

Suppose we restrict the system a bit and allow only one descendant of any given top-level transaction to be running at once. That is, we disallow concurrency *within* a transaction, permitting only a single leaf transaction for each top-level transaction. We still allow multiple *top-level* transactions, each with one running subtransaction.

Given this restriction, the live tree under each top-level transaction is linear, consisting of a single branch. This admits an interesting optimization: if an ancestor holds a value for a given address (read or written), a descendant reading that address does not need to add the address to its read set. The reason is that conflicts can only be with (subtransactions of) other top-level transactions, and the parent's entry is good enough for detecting the conflict. Furthermore, if all the transactions involved are closed nested, then the conflict will not resolve at least until the ancestor completes—it will be to the end of the top-level transaction if the value was written and none of the transactions aborts. In the case of open nesting, an intervening commit will remove the conflict. In any case, there is no need for the subtransaction to acquire a copy of the word unless it writes the word.

We can use linear nesting and the read optimization to simplify the cache logic. We encode transaction ids as a top-level transaction id (*Tid*) plus a nesting level (*Nest*). Top-level transactions have a Nest value of 1, their children have Nest value 2, etc. We reserve Nest value 0 for transaction 0 entries. Note that we no longer need a Transaction Parent Cache, since parenthood is encoded directly in the Nest values.

In addition to Tid, Nest, Address, Data, Valid, Dirty, and Written fields, our linear nesting cache design includes an additional field, the *Nested Write Stack* (NWS), which is an array indicating which higher nesting levels hold a write for this address. The NWS has limited size, so when a given word has too many nested values, we force overflowing ancestor entries to memory. We observe that cache entries flushed to memory for a given Tid and Address can be organized as a stack, simplifying the handling of overflowed entries.

Conflicts: Conflicting entries are simply those whose Tid is different and whose mode (read/write) conflicts with the action we want to perform. (We ignore transaction 0 entries in conflict detection.)

Reading a word: Assuming there is no conflict, we desire the value held by the transaction with the same Tid and the largest value for Nest. Such a value will be in the entry having the same Tid and Address, and an empty NWS (i.e., it has no descendant entries and thus is topmost). Because of the read optimization, we do *not* need to make a copy of the value read, unless no ancestor holds the value. In that case, we obtain the value from transaction 0, and create an entry with our Tid and Nest, the appropriate Address and Data, and set it Valid, Dirty, and not Written. We set the NWS to be empty.

Writing a word: Again assuming no conflict, we need there to be a value for the current transaction. If there is one, we simply update it. Otherwise, we create an entry, filling in its fields as follows. Tid, Nest, Address, and Data get the obvious appropriate values, and we set it Valid, Dirty, and Written. We push the writing Nest value on each ancestor's NWS. If the oldest ancestor's NWS overflows, we write it to memory.

If there was no write entry for an ancestor (detected by the transaction 0 entry having an empty NWS), we update the transaction 0 entry as follows: change the Tid to that of the writing transaction, and push the Nest value of the

writing transaction on the NWS. It is all right if there is an ancestor that has only read the address in question. Our purpose is to simplify dropping the transaction 0 entry if another value commits to top level. At most one top-level transaction can be writing at a time, so the Tid value will be unique, and the Nest value of 0 continues to mark the entry as being fully committed.

Aborting a transaction: We invalidate all entries for the Tid and Nest level. If the Nest value matches the top of an entry's NWS, that entry pops its NWS, discarding the NWS entry for the aborted transaction. In the case of an entry with Nest value 0 (which is really a transaction 0 value), we reset the Tid to 0.

Committing a transaction: If the committing transaction is top level (Nest 1) or open, we invalidate all the transaction's read set entries, and set its write entries to Tid 0 and Nest 0 (i.e., committing them to top level).

If the committing transaction is closed and not top-level, its entries, both read and write, decrement their Nest value.

If the committing transaction is closed, its ancestors (including Nest 0) react as follows. If an entry's NWS top value equals the committing transaction's Nest value, then it is an ancestor entry. The ancestor decrements its top NWS value. If the resulting value equals the ancestor's own Nest value (i.e., if the ancestor is the parent of the committing transaction), it invalidates itself. If the decremented top value equals the next value in the NWS, the ancestor pops its NWS.

If the committing transaction is open, its ancestors invalidate their values for this address. The entries to invalidate are those that have the committing transaction's Nest level at the top of their NWS.

6.1. Illustration

In Fig. 6, we show the linear nesting cache for the situation of Fig. 1. Note the renumbering of the transactions (old numbers listed under "Old" at the far right) and the omission of transaction 26 (which was concurrent with 28 and thus one or the other is not possible in linear nesting). Except for the old reader (14), the other transactions that only read this address do not have entries. In the illustration we assume that Nested Write Stacks hold 3 entries.

							NWS			
Tid	Nest	Addr	Data	V	D	W	0	1	2	Old
14	0	100	532	1	1	0	4	2	-	0
14	1	100	532	1	1	0	4	2	-	14
14	2	100	178	1	1	1	4	-	-	16
14	4	100	393	1	1	1	-	-	-	25

Fig. 6. Transaction data cache, linear nesting.

Assuming transactions 14.4 and 14.3 are closed and 14.2 is open, Figs. 7–9 show the situation after each commit.

							NWS			
Tid	Nest	Addr	Data	V	D	W	0	1	2	Old
14	0	100	532	1	1	0	3	2	-	0
14	1	100	532	1	1	0	3	2	-	14
14	2	100	178	1	1	1	3	-	-	16
14	3	100	393	1	1	1	-	-	-	23

Fig. 7. Linear nesting cache after 14.4 commits.

7. Related work

Transaction models and their implementations were initially developed for database and distributed systems, where they have received extensive treatment [4]. Our models are inspired by these fundamental works, especially with respect to formalization of serializability for nested transactions [2], concepts and applications of open nesting [21],

							NWS			
Tid	Nest	Addr	Data	V	D	W	0	1	2	Old
14	0	100	532	1	1	0	2	-	-	0
14	1	100	532	1	1	0	2	-	-	14
14	2	100	393	1	1	1	-	-	-	16

Fig. 8. Linear nesting cache after 14.3 commits.

							NWS			
Tid	Nest	Addr	Data	V	D	W	0	1	2	Old
0	0	100	393	1	1	0	-	-	-	0

Fig. 9. Linear nesting cache after 14.2 commits.

and synthesis of other extended transaction models [3]. There has been significant recent interest in formalizing transaction semantics within programming languages [20,10].

Various forms of transactional memory have been proposed and implemented since the hardware/software hybrid scheme of Herlihy and Moss [8]. Software-based approaches [7,22,9,19] can suffer from poor performance in cases of high data contention with large-scale concurrency.

There have been several recent hardware-based proposals. Some proposals take existing programs coded to use mutual-exclusion locks and execute them lock-free for improved concurrency while enforcing transactional behavior, while hardware overflow causes explicit acquisition of the mutual-exclusion lock [16,17]. Other approaches execute transactions speculatively in the cache but force non-speculative execution on overflow [5,6]. Unbounded Transactional Memory (UTM) attempts to solve the problem of transaction boundedness but suffers from performance degradation in the normal case [1]. Virtualized Transactional Memory (VTM) maintains the performance advantage of hardware transactions, while also shielding programmers from hardware resource limits [18]. None of these hardware schemes offer comprehensive support for nested or open transactions.

8. Conclusion

Our intent has been to describe memory level (read/write) execution semantics for closed and open nested transactions and to explore some implementation strategies. While the resulting designs have more associative logic than current caches, perhaps we are approaching sketches worth the effort of more detailed hardware design and performance/cost estimation. In any case, we have marked out at least one potentially interesting design point, namely linear nesting.

Much remains to be done: detailed hardware design and evaluation; design of language constructs and run-time support that map onto this model; a suitable serializability theorem justifying the model; incorporation with multiprocessor bus protocols (such as cache snooping, etc.); and development of and comparison with alternatives. Such alternatives include software-only designs, extending current software transactional memory (STM) systems with nesting, and the possibility of supporting nesting with essentially non-nested hardware support.

If linear nesting is deemed to be at the edge of achievable hardware complexity, then we face the interesting question of what we really lose by restricting ourselves to linear nesting. Is it possible to finesse concurrent subtransactions in software using linear nesting hardware? For that matter, can we finesse nesting on non-nesting hardware? Presumably in either case we would design a simpler cache that, in combination with software, could model the richer semantics. It is not clear what “hooks” one needs to pull that off.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grant number CCR-0085792. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. The model presented here was developed as part of an

ongoing collaboration with Bradley Kuszmaul, Charles Leiserson, Gideon Stupp, and James Sukha. We particularly acknowledge the value of our discussions with them on the semantics of open nesting and the concept of linear nesting.

References

[1] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, S. Lie, Unbounded transactional memory, in: Proceedings of the International Symposium on High Performance Computer Architecture, IEEE Computer Society, 2005, pp. 316–327.
[2] C. Beeri, P.A. Bernstein, N. Goodman, A model for concurrency in nested transactions systems, J. ACM 36 (2) (1989) 230–269.
[3] P. Chrysanthis, K. Ramamritham, Synthesis of extended transaction models using ACTA, ACM T. Database Syst. 19 (3) (1994) 450–491.
[4] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.
[5] L. Hammond, B.D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, K. Olukotun, Programming with transactional coherence and consistency (TCC), in: Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 39, November 2004, pp. 1–13.
[6] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun, Transactional memory coherence and consistency, in: Proceedings of the International Symposium on Computer Architecture, vol. 32, December 2004, pp. 102–113.
[7] T. Harris, K. Fraser, Language support for lightweight transactions, in: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, vol. 38, November 2003, pp. 388–402.
[8] M. Herlihy, 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.
[9] M. Herlihy, V. Luchangco, M. Moir, 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.
[10] S. Jagannathan, J. Vitek, Optimistic concurrency semantics for transactions in coordination languages, in: Coordination Models and Languages, in: Lecture Notes in Computer Science, vol. 2949, 2004, pp. 183–198.
[11] J.E.B. Moss, Nested Transactions: An Approach to Reliable Distributed Computing, Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, Apr. 1981; also published as MIT Laboratory for Computer Science Technical Report 260.
[12] J.E.B. Moss, Nested transactions: An approach to reliable distributed computing, in: Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems, IEEE, Pittsburgh, PA, 1982, pp. 33–39.
[13] J.E.B. Moss, Nested Transactions: An Approach to Reliable Distributed Computing, M.I.T. Press, Cambridge, MA, 1985.
[14] J.E.B. Moss, A.L. Hosking, Nested transactional memory: Model and preliminary architecture sketches, in: the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages, SCOOL (no formal proceedings), October 2005.
[15] J.E.B. Moss, N.D. Griffeth, M.H. Graham, Abstraction in recovery management, in: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, May 1986, pp. 72–83. ACM SIGMOD Record 15, 2 (June 1986).
[16] R. Rajwar, J.R. Goodman, Speculative lock elision: Enabling highly concurrent multithreaded execution, in: Proceedings of the International Symposium on Microarchitecture, ACM/IEEE, 2001, pp. 294–305.
[17] R. Rajwar, J.R. Goodman, Transactional lock-free execution of lock-based programs, in: Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 2002, pp. 5–17. ACM SIGPLAN Notices 37, 10 (October 2002).
[18] R. Rajwar, M. Herlihy, K.K. Lai, Virtualizing transactional memory, in: Proceedings of the International Symposium on Computer Architecture, 2005, pp. 494–505.
[19] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, 1995, pp. 204–213.
[20] J. Vitek, S. Jagannathan, A. Welc, A.L. Hosking, A semantic framework for designer transactions, in: D.E. Schmidt (Ed.), Proceedings of the European Symposium on Programming, in: Lecture Notes in Computer Science, vol. 2986, 2004, pp. 249–263.
[21] G. Weikum, H.-J. Schek, Concepts and Applications of Multilevel Transactions and Open Nested Transactions, Morgan Kaufmann, 1992, pp. 515–553.
[22] A. Welc, S. Jagannathan, A.L. Hosking, Transactional monitors for concurrent objects, in: M. Odersky (Ed.), Proceedings of the European Conference on Object-Oriented Programming, in: Lecture Notes in Computer Science, vol. 3086, Springer-Verlag, 2004, pp. 519–542.