HParSTM: A Hierarchy-based STM Protocol for Supporting Nested Parallelism

Ranjeet Kumar and K. Vidyasankar*

Department of Computer Science, Memorial University of Newfoundland, Canada A1B 3X5
{ranjeet.kumar, vidya}@mun.ca

Abstract. Software transactional memory (STM) has emerged as a promising technique for facilitating concurrent programming to exploit the potential of multicore machines in a programmer-friendly manner. One of the striking features of STM is its ability to support composability of transactions through nesting, i.e., smaller subtransactions can be glued together to form larger transactions. So far, a number of protocols have been proposed for supporting non-nested transactions in STM. The work related to nested transactions support either serial execution of subtransactions (i.e. no nested parallelism) or allow only child-level parallelism in which the parent is not allowed to execute its operations while it has active children subtransactions.
In this paper, we present an STM protocol (HParSTM) for closed nested transactions. Some of the key features of this protocol are (1) parallelism between the sibling subtransactions as well as between the parent and the child transactions, (2) satisfying opacity property, (3) aborting a (sub)transaction only when it conflicts with some other live (sub)transaction (progressiveness property), (4) use of visible reads, (5) flexibility for read only nested subtransactions, (6) purely optimistic approach, and (7) hierarchical design.

Keywords: Software transactional memory, Transaction, Atomic operation, Closed nested transactions, Nested parallelism, Shared object, Consistent global state, Lock, Opacity, Progressiveness.

1 Introduction

Software transactional memory. To complement the rapid growth in the development of multicore machines, a paradigm shift in programming practice, from sequential programming to concurrent programming, is being observed. To this end, software transactional memory (STM) aims at providing a mechanism for handling low-level concurrency control for accessing shared objects in a multi-threaded environment in such a way that the programmers can write programs without having to worry about the underlying concurrency management. Using STM system, a programmer delimits the regions of a program that must execute atomically. Each atomic region constitutes a transaction.

Opacity and progressiveness. The notion of opacity, introduced and formalized by Guerraoui et. al. [11], is a widely accepted correctness criterion for the concurrent execution of transactions in STM systems [1, 9, 11]. Opacity requires that all transactions (including the aborted ones) access consistent states. Another property desirable in STM systems is *progressiveness* [1]. An STM system is *progressive* if it forcefully aborts a transaction only when there is a time τ at which conflicts with another concurrent transaction that is not committed or aborted by time τ; we say that two transactions conflict if they access some common shared object, and at least one writes the object [11].

Transaction tree and closed nested transactions. One of the unique features of STM is the composability of the transactions. In other words, a set of smaller transactions can be combined to form a larger transaction through nesting. The different types of nesting are (a) flat nesting (b) closed nesting, and (c) open nesting. In this paper, we only deal with closed nested transaction. Nested transactions are created when an atomic region is created inside an outer atomic region. The execution of nested subtransactions can be conceptually represented by a dynamic tree of active subtransactions, called *transaction tree* [8], in which the transactions are related by parent-child relationship. The transaction at the topmost level, that has no parent, is referred to as root transaction. When a root transaction commits, all the changes made to the globally shared objects by the root transaction become visible to all other transactions executing concurrently in the system. The commit of a closed nested subtransaction is only local to its parent, i.e., instead of making actual changes to the globally shared data, it merges its read and write sets with those of its parent upon committing. The abort of a closed nested subtransaction does not affect the state of its ancestor(s).

Contribution of the paper. Most of the work in STM so far has been carried out for normal (non-nested) transactions. Those that consider nested transactions support only *serial execution* of nested subtransactions [6, 7]. Recent works [2, 3, 4, 5, 10] go further to support parallelism at the child-level, under the restriction that the parent transaction does not execute while it has active children. So far, no pa-

* This research is supported in part by the Natural Sciences and Engineering Research Council of Canada individual discovery grant 3182.

per (to the best of our knowledge) allows parent transaction to execute concurrently with its children transactions.

In this paper, we present an STM protocol (HParSTM), for closed nested transactions, that offers full parallelism at all levels i.e. parallelism between sibling transactions as well as between the parent and the children transactions. The benefit of parent/child parallelism is that it can reduce the branching in the transaction tree by distributing tasks, independent ones in particular, at the parent and child levels. Moreover, all the nodes in the transaction tree can now participate in active computation. This way, we obtain a smaller transaction tree, and thereby reduce the depth of nesting needed.

We build upon the protocol proposed in [1] (described in Section 3.1). The choice of this protocol as a base for our algorithm is motivated by its following features: (1) simplicity of idea, (2) satisfying opacity and progressiveness properties, and (3) formal proof of correctness.

Roadmap. The paper is made up of 8 sections. Section 2 describes the base system model, and discusses the problem specification. We first offer an insight into the design of the Protocol (HParSTM) by providing an overview in Section 3, followed by the discussion of the complexity involved in handling the contention management in Section 4. In Section 5, we present the actual protocol (pseudocode and description). In Section 6, we briefly discuss the correctness of the protocol. In Section 7, we discuss the related research works, and Section 8 concludes the paper.

2 System Model

Our system is similar to the one used in [1]. The base computational model consists of processes, base objects, locks and atomic registers. There are n asynchronous sequential processes (i.e. threads) denoted $p_1, ..., p_n$ that cooperate through base read/write atomic registers and locks. The local as well as global copies of shared objects (e.g., the base object x) are protected by individual locks. Each process is made of up a sequence of transactions along with some code to manage these transactions, and has its own memory. The processes issue transactions one at a time.

A transaction is a sequence of read and write operations that can examine (read) and modify (write), respectively, the state of the base objects. It consists of a sequence of events that are an *operation invocation*, an *operation response*, a *subtransaction invocation* and *response*, a *commit invocation*, a *commit response*, and an *abort event*. An operation is considered *terminated* if its response event has occurred. Similarly, a transaction is considered *completed* if its commit response or abort event has occurred.

Further, each transaction satisfies the following constraints: (1) a transaction performs one operation at a time, and (2) a composite transaction must wait until all of its (active) children have completed before entering the validation for its commit.

The set of transactions is denoted by T. The set of objects is denoted by X and the set of possible values associated with them is V. A local copy of an object x (or a set s) associated with a transaction t is denoted by $t.x$ ($t.s$) to avoid ambiguity. Likewise, a field b associated with an object a is represented as $a.b$. Further, we sometimes use the notation $t.foo()$, instead of $foo_t()$, for referring to a method $foo()$ associated with a transaction t.

3 Design Overview of HParSTM

As stated earlier, the protocol presented in this paper builds upon the base protocol proposed in [1] for non-nested transactions. We provide a brief outline of that protocol. (Capital letters have been used in [1] to denote transactions and shared objects).

3.1 Overview of the base protocol [1] by D. Imbs and M. Raynal

---

Protocol 1:

---

operation $read_t$ (x):
(01) **if** ($t.x$ not exists) **then**
(02) allocate local space for $t.x$;
(03) $t.lrs \leftarrow t.lrs \cup \{x\}$;
(04) lock $t_\psi.x$; $t.x \leftarrow t_\psi.x$; $t_\psi.x.rs \leftarrow t_\psi.x.rs \cup \{t\}$; unlock $t_\psi.x$;
(05) **if** ($t \in t_\psi.x.fbd$) **then** return (abort) **end if**
(06) **end if**;
(07) return(value of $t.x$)

operation $write_t(x, v)$:
(08) $t.read_only \leftarrow false$;
(09) **if** ($t.x$ not exists) **then** allocate local space for $t.x$ **end if**;
(10) $t.x \leftarrow v$;
(11) $t.lws \leftarrow t.lws \cup \{x\}$;

operation $try_to_commit_t$ ():
(12) **if** ($t.read_only$)
(13) **then** return(commit);
(14) **else** lock all the objects in $t.lrs \cup t.lws$;
(15) **if** ($t \in t_\psi.ow$) **then** release all the locks; return(abort) **end if**;
(16) **for each** $x \in t.lws$ **do** $t_\psi.x \leftarrow t.x$ **end for**;
(17) $t_\psi.ow \leftarrow t_\psi.ow \cup (\cup_{x \in t.lws} t_\psi.x.rs)$;
(18) **for each** $x \in t.lws$ **do** $t_\psi.x.fbd \leftarrow t_\psi.ow; t_\psi.x.rs \leftarrow \emptyset$; **end for**;
(19) release all the locks;
(20) return(commit)
(21) **end if**

Protocol 1 uses single shared copy of each base object x. Each transaction has its own local copy of the base object associated with its read/write steps. Note that a local copy of object x available with a transaction t is denoted by $t.x$, and the globally shared copy is denoted by $t_\psi.x$. To keep track of the conflicts between the transactions, the following control variables are used.

- ow**:** *overwritten* set (denoted as $t_\psi.ow$) that contains the ids of the transactions that read some object x that was modified later (so, there is a conflict).
- rs**:** *read* set (denoted as $t_\psi.x.rs$) associated with each global object $t_\psi.x$. It stores the ids of the transactions that read from the object $t_\psi.x$ since its last update.
- fbd**:** *forbidden* set (denoted as $t_\psi.x.fbd$) associated with each global object $t_\psi.x$. If $t \in t_\psi.x.fbd$, then it means that the transaction t has read an object $t_\psi.y$ that since then has been overwritten (hence $t \in t_\psi.ow$), and the overwriting of $t_\psi.y$ is such that any future read of $t_\psi.x$ by t will be invalid (i.e., the value obtained by t from $t_\psi.y$ and any value it will obtain from $t_\psi.x$ in the future cannot be mutually consistent).

In the following discussion, all the references to line numbers are associated with Protocol 1. When a transaction t_1 performs a read operation on $t_\psi.x$, it adds its id in $t_\psi.x.rs$ (line 04). Later, when another transaction t_2 modifies objects $t_\psi.x$ and $t_\psi.y$, it adds t_1 to $t_\psi.ow$ (line 17), followed by updating both $t_\psi.x.fbd$ and $t_\psi.y.fbd$ (line 18). Now, if t_1 tries to access $t_\psi.y$, it will detect the conflict by noticing its id in $t_\psi.y.fbd$, and consequently abort (due to line 05). Besides, a transaction t maintains two local sets, $t.lrs$ (*local read set*) and $t.lws$ (*local write set*) to document its read/write operations (lines 03, 11). Before committing, the validation phase comprises of ensuring that the transaction does not belong to set $t_\psi.ow$ (line 15). However, a transaction that has no write operation is committed immediately (lines 12-13), and is thus treated differently.

3.2 Overview of HParSTM

The higher level design of HParSTM can be characterized by the following features. An in [1], we have a control variable $t_\psi.ow$, and shared copies of base objects using sets rs and fbd as control variables at the global level. We also replicate these variables (lock based local copies of objects using rs and fbd, and a set ow) at each level (node) of transaction tree. Similar semantics are associated with the operations on these control variables (rs, fbd and ow). A (sub)transaction t's local copy $t.x$ is accessible to t as well as its descendants. When the descendants of t access $t.x$, they add their ids to $t.x.rs$ (Of course, transaction t does not have to add its id to $t.x.rs$ as $t.x$ is its local object). Later, if $t.x$ is modified by t or its children during their commit, then all the ids present in $t.x.rs$ are added to $t.ow$, followed by updating $x.fbd$ using $t.ow$, and clearing $t.x.rs$ (similar to Protocol 1: lines 17-18).

If a transaction t does not have a local copy of an object x in its local space, then it first tries to read from its nearest ancestor having a local copy of object x. If none of the ancestors currently have a local copy of x, then t tries to read from the globally shared copy of x.

Super transaction and super tree. We associate all the globally shared objects/variables with a highest level fictitious transaction called *super transaction*, denoted by t_ψ, such that all the transactions previously referred to as root-level transactions are now children of the super transaction. We call the resulting tree as *super tree*. Now, each transaction tree is a subtree rooted at a child of the super transaction in super tree.

4 Discussion of Contention Management Issues

While designing HParSTM, we face several contention management issues related to nested transactions. Let the read operation performed by a transaction t_1 on an object x associated with transaction t_2 be denoted by $r_{t_1}(t_2.x)$. Similarly, the write operation is denoted by $w_{t_1}(t_2.x)$. We discuss the main issues using Fig. 1.

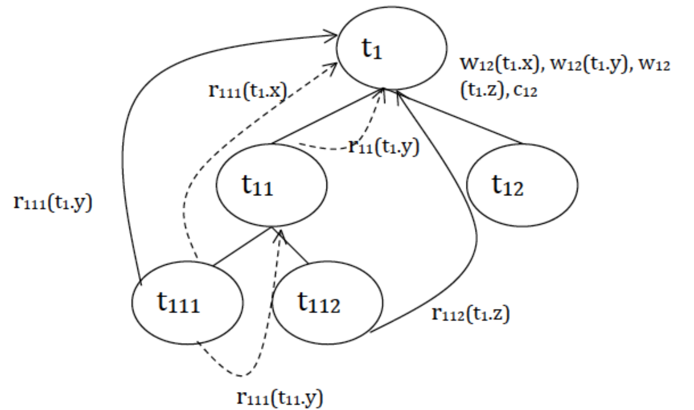

Fig. 1: Depicting contention management issues

4.1 Consistency Checking at the Time of Read Operation

In [1], a transaction t has to take into account only its own id while checking the consistency of its read step on a globally shared object. This is not sufficient in case of a nested transaction. When a subtransaction t tries to read from an object $t^{'}.x$ from its ancestor $t^{'}$'s local space, it should ensure that $t^{'}.x.fbd$ does not contain the id of (i) t or any of its intermediate ancestors up to $t^{'}$ and (ii) the subtransactions that have already committed and merged with them.

4.2 Avoiding Cyclic Conflict through Transitivity across Levels

For analyzing the cyclic conflict between transactions across different levels, consider the following history based on Fig.

1 (consider dotted arrows).

$\mathcal{H}_1 = r_{t_{111}}(t_1.x), w_{t_{12}}(t_1.x), w_{t_{12}}(t_1.y), w_{t_{12}}(t_1.z), c_{t_{12}}, r_{t_{11}}(t_1.y), r_{t_{111}}(t_{11}.y)$.

Here, first t_{111} reads from $t_1.x$ ($\Rightarrow t_{111} \in t_1.x.rs$). Later, t_{12} modifies $t_1.x, t_1.y$ and $t_1.z$ ($\Rightarrow t_{111} \in t_1.x.fbd, t_1.y.fbd$ and $t_1.z.fbd$). Next, t_{11} successfully reads from $t_1.y$ (as $t_{11} \notin t_1.y.fbd$) and creates its own local copy $t_{11}.y$. Now, the question is if t_{111} should be allowed to access $t_{11}.y$? The answer is no as it creates a cyclic conflict relation between t_{12} and t_{111}. Transaction t_{12} modified $t_1.x$ read earlier by t_{111}, giving the serial order t_{111}, t_{12}. Next, t_{11} reads $t_1.y$ after t_{12} modifies $t_1.x, t_1.y$ and $t_1.z$ together. Notice that t_{111} reading from $t_{11}.y$ is as good as reading directly from $t_1.y$, as the value of $t_{11}.y$ is copied from $t_1.y$. That gives the serial order t_{12}, t_{111}, and hence the contradiction (cycle).

Solution. Checking the $t^{'}.x.rs \cup t^{'}.x.fbd$ of each ancestor $t^{'}$ of t_{111} to figure out the ids of descendants of t_{11} that previously read x would be a very expensive task. A simple solution to the problem is as follows. At the time of reading $t_1.y$, t_{11} can notice that $t_{111} \in t_1.x.fbd$, and consequently adds t_{111} to $t_{11}.y.fbd$ and $t_{11}.ow$ (as $t_{111} \in t_{11}.y.rs$) before it finishes its read operation. Now, if t_{111} tries to access $t_{11}.y$, it will fail as $t_{111} \in t_{11}.y.fbd$.

4.3 Keeping Track of Incompatible Read Operations

$\mathcal{H}_2 = r_{t_{111}}(t_1.y), w_{t_{12}}(t_1.x), w_{t_{12}}(t_1.y), w_{t_{12}}(t_1.z), c_{t_{12}}, r_{t_{112}}(t_1.z), (c_{111}, c_{112})$?

Consider the history $\mathcal{H}_2$ in Fig. 1 (following solid arrows). Here, suppose $t_{111} \in t_1.y.rs$ initially. Next, t_{12}, during its commit ($w_{t_{12}}(t_1.x)$), adds t_{111} to $t_1.x.fbd$, $t_1.y.fbd, t_1.z.fbd$ and $t_1.ow$. Now, t_{112}, on $r_{t_{112}}(t_1.z)$, finds $t_{111} \in t_1.y.fbd$. As t_{111} has not yet committed and merged with t_{112}'s ancestor t_{11}, it is legal for t_{112} to read from $t_1.z$ at this point.

Now, the conflicting write operations of t_{12} are sandwiched between the respective read operations of t_{111} and t_{112} on t_1's objects. Hence, t_{11} cannot be serialized at t_1's level if both t_{111} and t_{112} are allowed to commit. In other words, the two read operations $r_{t_{111}}(t_1.y)$ and $r_{t_{112}}(t_1.z)$ are *incompatible* for t_{11}. Thus, *incompatible* operations and transactions are defined as follows.

Definition 1 (Incompatible operations and transactions). *Let t_1, t_2 be any two transactions in the super tree, and t be the least common ancestor of t_1 and t_2. Let $r_{t_1}(t^{'}.x)$ and $r_{t_2}(t^{'}.y)$ be successful read operations of t_1 and t_2 respectively on $t^{'}$'s objects x and y, where $t^{'}$ is an ancestor of t. Then these two read operations are incompatible if $t_1 \in t^{'}.y.fbd$ at the time of $r_{t_2}(t^{'}.y)$, or vice-versa. The two such transactions, t_1 and t_2, are called incompatible transactions.*

Solution. Each subtransaction t maintains a set called *its* (*incompatible transaction set*) to keep track of the transactions it is incompatible with, and a set *mts* (*merged transaction set*) that contains the ids of subtransactions that have merged with t, initially containing t. Further, set *visited* contains the ids of t's descendants that have visited t to search for a local copy of an object. Suppose t reads x from an ancestor $t^{'}$. Let $t^{'}_{prev}$ be the transaction preceding $t^{'}$ in the path from $t^{'}$ to t in the super tree. Then ids of all the transactions, that are in $t^{'}.x.fbd$ and also present in $t^{'}_{prev}.visited$, are added to *t.its*. Later, when t tries to merge with its parent, it ensures that none of the transactions in *t.its* has already merged with its parent. Similarly, the parent has to ensure that none of the transactions in its own local set *its* is a part of its child transaction trying to commit (merge).

5 Protocol

Protocol 2 describes HParSTM.

5.1 Transaction State

A transaction t begins with begin(*parent_id*), where *parent_id* denotes the id of the parent of t. The *parent_id* is *null* for a root-level transaction. Then, it accesses memory locations using read or write operations. Finally, it completes either by a try_to_commit call or by a abort call. The methods search_anc, check_compatibility and abort are helper functions. The *t.lrs* (*local read set*) is used to record the objects it read as well as the ancestors to which those objects belong. A set *t.lws* (*local write set*) is used to record its write steps. The access to each copy of a base object is protected by a lock. The set *t.ow* is used to store the ids of those descendants of t that have read a value from its locally shared objects whose values have been modified since the reading. The variable *t.parent* denotes the id of the parent of transaction t. The set *t.prefix* contains the ids of all the ancestors of t in the super tree, including t. The set of currently active children of t is given by *t.acs* (*active children set*). Further, *t.mts* (*merged transaction set*) contains the ids of t as well as those descendants of t whose results have been propagated to (merged with) t. A descendant $t^{'}$ of t is included in *t.mts* only when $t^{'}$ and all of its intermediate ancestors upto t commit. The set *t.its* (*incompatible transaction set*) associated with a transaction t denotes a set of transactions that t is incompatible with and hence cannot be merged together. Further, a set *t.visited* contains the ids of t's descendants that visited node t while searching for a local copy of an object.

Protocol 2: HParSTM

-$\triangle$ denotes "lower level to higher level";
-unlock $t_{prev}.x$ (line 27) is applicable only when $t_{prev} \neq t$;
-$try_to_merge_{t_p}(t)$ is executed by the parent t_p for one child t at a time.

1. **State of base object x:**
2. $val :\in V$
3. rs and fbd : $\subset T$
4. **State of transaction t:**
5. $parent \in T$, parent's id (t_p)
6. $lws \subset X$
7. $lrs \subset X$
8. $prefix, acs, mts, its, visited$ and $ow \subset T$
9. los, a set of local copies of objects
10. **Operation** $begin_t(t_p)$:
11. $t.parent \leftarrow t_p$;
12. $t.prefix \leftarrow t_p.prefix \cup \{t\}$;
13. $t.mts \leftarrow \{t\}$;
14. $t_p.acs \leftarrow t_p.acs \cup \{t\}$;
15. **Operation** $invoke_child_t(t_c)$:
16. $t_c.begin(t)$;
17. **Operation** $read_t(x)$:
18. lock $t.x$;
19. **if** ($t.x.val \neq null$) **then**
20. $v \leftarrow t.x.val$;
21. unlock $t.x$; return v; **end if**;
22. $v = search_anc_t(x)$;
23. unlock $t.x$; return v;
24. **Operation** $search_anc_t(x)$:
25. $s_m \leftarrow t.mts$; $t_{prev} \leftarrow t$;
26. **for each** $\triangle t' \in (t.prefix \setminus \{t\})$ **do**
27. lock $t'.x$; unlock $t_{prev}.x$;
28. $t'.x.rs \leftarrow t'.x.rs \cup \{t\}$;
29. $t'.visited \leftarrow t'.visited \cup \{t\}$;
30. **if** ($t'.x.val \neq null$) **then**
31. **if** (($t'.x.fbd \cap s_m \neq \emptyset) \vee$
32. $\neg check_compatibility_t(t')$) **then**
33. unlock $t'.x$; $t.abort(1)$; **end if**
34. $t.x.val \leftarrow t'.x.val$;
35. $t.x.fbd \leftarrow t'.x.fbd \cap t_{prev}.visited$;
36. unlock $t'.x$;
37. $t.its \leftarrow t.its \cup t.x.fbd$;
38. $t.ow \leftarrow t.ow \cup t.x.fbd$;
39. $t.lrs \leftarrow t.lrs \cup \{x\}$;
40. return $t.x.val$; **end if**
41. $s_m \leftarrow s_m \cup t'.mts$;
42. $t_{prev} \leftarrow t'$; **end for**
43. **Operation** $write_t(x, v)$:
44. lock $t.x$; $t.x.val \leftarrow v$;
45. $t.ow \leftarrow t.ow \cup t.x.rs$;
46. $t.x.fbd \leftarrow t.ow$; $t.x.rs \leftarrow \emptyset$;
47. unlock $t.x$;
48. $t.lws \leftarrow t.lws \cup \{x\}$;
49. **Operation** $try_to_merge_{t_p}(t)$:
50. lock all $t_p.x : x \in (t.lrs \cup t.lws)$;
51. **if** (($t.lws \neq \emptyset \wedge (t_p.ow \cap t.mts) \neq \emptyset) \vee$
52. ($\neg check_compatibility_t(t_p)$)) **then**
53. $abort_t(1)$; **end if**
54. **for each** $x \in (t.los \setminus t_p.los) \wedge t.x.val \neq null$ **do**
55. $t_p.x.val \leftarrow t.x.val$;
56. $t_p.x.fbd \leftarrow t.x.fbd$; **end for**
57. $t_p.ow \leftarrow t_p.ow \cup t.ow \cup (\cup_{x \in t.lws} t_p.x.rs)$;
58. **for each** $x \in t.lws$ **do**
59. $t_p.x.val \leftarrow t.x.val$;
60. $t_p.x.fbd \leftarrow t_p.ow$;
61. $t_p.x.rs \leftarrow \emptyset$; **end for**
62. $t_p.lws \leftarrow t_p.lws \cup t.lws$;
63. $t_p.lrs \leftarrow t_p.lrs \cup t.lrs$;
64. $t_p.mts \leftarrow t_p.mts \cup t.mts$;
65. $t_p.its \leftarrow t_p.its \cup t.its$;
66. $t_p.acs \leftarrow t_p.acs \setminus \{t\}$;
67. release all the locks;
68. **Operation** $check_compatibility_t(t')$:
69. return (($t.mts \cap t'.its = \emptyset) \wedge (t.its \cap t'.mts = \emptyset)$);
70. **Operation** $try_to_commit_t()$:
71. **if** ($t.parent \neq t_\psi$) **then**
72. $try_to_merge_{t_p}(t)$;
73. return ($commit$);
74. **else** // t is a root-level transaction
75. **if** ($t.lws$ is empty) **then**
76. return ($commit$); **end if**
77. lock all objects $t_\psi.x : x \in (t.lrs \cup t.lws)$;
78. **if** ($t.mts \cap t_\psi.ow \neq \emptyset$) **then**
79. $abort_t(1)$; **end if**
80. **for each** $x \in t.lws$ **do**
81. $t_\psi.x.val \leftarrow t.x.val$; **end for**
82. $t_\psi.ow \leftarrow t_\psi.ow \cup (\cup_{x \in t.lws} t_\psi.x.rs)$;
83. **for each** $x \in t.lws$ **do**
84. $t_\psi.x.fbd \leftarrow t_\psi.ow$;
85. $t_\psi.x.rs \leftarrow \emptyset$; **end for**
86. release all the locks;
87. return ($commit$);
88. **Operation** $abort_t(abort_type)$:
89. release all the locks;
90. **if** ($abort_type = 1$) **then**
91. **if** ($t_p \neq null$) **then**
92. $t_p.acs \leftarrow t_p.acs \setminus \{t\}$; **end if**
93. $abort_type = 2$; **end if**
94. **for each** $t' \in t.acs$ **do**
95. $t'.abort(abort_type)$; **end for**
96. return ($abort$);

5.2 Working of HParSTM:

In HParSTM, the allocation of space for local copy of an object (x) in the local space of a transaction is automatically done whenever required. For a transaction t, this typically happens when it (or its descendant) tries to obtain a lock on a local copy $t.x$, and $t.x$ does not already exist. At the time of

memory allocation, the initial state of $t.x$ is: $t.x.val = null$, $t.x.rs = \emptyset$ and $t.x.fbd = \emptyset$. Further, in the text, wherever a transaction is looking for a local copy of an object to read, we mean non-null valued copy of that object. Several steps of the protocol are self-explanatory. We describe only the salient features. The procedures of the protocol are discussed as follows.

$begin_t(t_p)$: This method is called by the parent transaction t_p at the time of invoking a new child transaction t.

$invoke_child_t(t_c)$: This method is used by transaction t to invoke a new child t_c.

$read_t(x)$: If a local copy of an object x is available with the transaction, then it reads the value from that copy. Otherwise, it calls search_anc.

$search_anc_t(x)$: This procedure searches for the nearest ancestor of t having a non-null valued local copy of x, starting from the parent of t. At each level $t^{'}$, $t^{'}.x$ is first locked (and the previous level lock is released), and $t^{'}.x.rs$ as well as $t^{'}.visited$ are updated. If $t^{'}.x$ is non-null, then the consistency check is done as follows. It ensures that (1) none of transactions in the set mts of t or any of t's intermediate ancestors up to $t^{'}$ belongs to $t^{'}.x.fbd$, and (2) t is compatible with $t^{'}$. If the read step is found to be inconsistent, then the transaction releases the lock on $t^{'}.x$ and aborts. If the read step is valid, then it assigns the value to $t^{'}.x$ to $t.x$. Let t_{prev} be the transaction preceding $t^{'}$ in $t.prefix$ in the order of lower level to higher level. All the transactions that are common in $t^{'}.x.fbd$ and $t_{prev}.visited$ are added to $t.x.fbd$, and then $t^{'}.x$ is unlocked. Later, $t.x.fbd$ is used to update $t.its$ and $t.ow$. Then entry x is added to $t.lrs$.

$write_t(x, v)$: All the writes take place in the local space. The effect of the write is reflected in the global object only when the root transaction commits. In order to perform a write on $t.x$ in the local space, the transaction locks the local copy $t.x$, updates the value of $x.val$, adds $x.rs$ to ow, and updates $x.fbd$ using ow. Then, t unlocks $t.x$, and adds x in $t.lws$.

$try_to_merge_{t_p}(t)$: This is a synchronized method, and hence only one of the children of t_p can invoke it at a time. For each $x \in (t.lrs \cup t.lws)$ locks are obtained on parent's (t_p) object $t_p.x$. Remaining objects of t_p are still accessible to its descendants for locking and reading. Before merging its local sets, t has to first check the consistency of its steps for a successful merge process. If the consistency checking fails, then the transaction aborts and releases the locks on parent's objects. It is possible that read and write sets of the subtransaction contain certain objects that are not present in the local read and write sets of its parent. Such objects in the parent's local space are updated using the child's local copy, and the control variables are also updated accordingly.

Next, for all objects $\in t.lws$, the ids in $t_p.x.rs$ are added to $t_p.ow$. Now, for each object $x \in t.lws$, value of $t_p.x$ is updated using $t.x$, $t_p.x.fbd$ is updated using $t_p.ow$, and $t_p.x.rs$ is cleared. Finally, sets $t.lws, t.lrs$, are merged with corresponding sets of the parent t_p. Finally, the sets $t.mts$ and $t.its$ are merged with those of the parent, and t is removed from the parent's set $t_p.acs$.

$check_compatibility_t(t^{'})$: This method returns true if t and $t^{'}$ are compatible i.e. $((t.mts \cap t^{'}.its = \emptyset) \wedge (t.its \cap t^{'}.mts = \emptyset))$. Otherwise, it returns false.

$try_to_commit_t()$: The nature of a commit process of a transaction t depends upon its type. If t is a non-root transaction, its effect is not reflected on the global objects immediately, rather it tries to merge its local read and write sets with the local read and write sets of its parent.

In case t is a root-level transaction, then the behaviour of the validation process for t is the similar to that proposed in [1] for a non-nested transaction. When the root transaction commits, the objects in its local write set are modified globally i.e. the change is reflected in the globally shared copy of objects available with t_ψ. If it is *read only*, then it commits immediately. Otherwise, it locks all the objects in its sets $t.lrs$ and $t.lws$. Then, it checks if any of the subtransactions in its set $t.mts$ belongs to the $t_\psi.ow$ set. If yes, then it means that the consistency of the root transaction has been compromised, and the transaction releases all the locks before aborting. Otherwise it updates the values of all the global objects in its write set, followed by updating $t_\psi.ow$ and $t_\psi.x.fbd$ for each x it writes. Finally, the root transaction releases all the locks and commits. Note that, during the commit of a root transaction, there is no checking of incompatibility or update operations for the sets acs, mts, its, lrs, lws of its parent.

$abort_t(abort_type)$: This method is invoked when a transaction t has to be aborted. Before aborting, transaction t releases all the locks in its possession. If $abort_type = 1$ then, it removes its id from the set acs of its parent, provided it is not a root level transaction (whose parent is t_ψ). The descendants of an aborting transaction do not necessarily need to update the acs of their parent. Therefore, the $abort_type$ is set to 2 during the abort of the descendants. Finally, t calls the abort(2) method of its active children (if any).

6 Informal discussion of proof for correctness

To set up an analogy between Protocol 1 [1] and HParSTM, the local copies of objects associated with a node t in a transaction tree can be treated as globally shared copies for the children (descendants) of t. Let t_c be a child of t. Then, observe that the entire transaction subtree rooted at t_c can be treated as a single transaction.

Construction of level-wise history $\mathcal{H}_t$ at node t: The history at node t is constructed as follows. Any read or write operation performed on $t.x$ by t_c or its descendants is attributed to t_c. Further, each local read or write operation performed by t itself on $t.x$ is considered to have been performed by a fictitious committed child of t, having that operation as its only step.

Definition of linearization point: An instant of time, i, in the lifespan (local timeline) τ_t of transaction t is denoted by τ_t^i. Now, the linearization point, ℓ_{t_c}, for subtransaction t_c is the time τ_t^i at which the entire execution of t_c can treated to have occurred. Let $\widehat{t_c}$ denote "some transaction in $t_c.mts$". Now, ℓ_{t_c} is defined as follows.

- If t_c aborts, ℓ_{t_c} is the time just before $\widehat{t_c}$ is added to the set $t.ow$ (line 57 or 82).
- If t_c is read only transaction and commits, ℓ_{t_c} is placed at the earliest of (1) the occurrence time of the test during its last read operation (line 31-32 of the *search_anc* operation) and (2) the time just before $\widehat{t_c}$ is added to $t.ow$ (if it ever is).
- If t_c is an update transaction and commits, ℓ_{t_c} is placed just after the execution of line 57 or line 82 by t (update of $t.ow$).

Following a bottom up approach, we consider (consider) the level-wise history at a level (node) t, and show the correctness at that level. Next, we consider the history at the parent of t. Further, we separate the history of committed transactions from that of aborted ones.

Committed transactions: The history restricted to committed transactions, obtained from $\mathcal{H}_t$, is given by $\Pi(\mathcal{H}_t)$. The steps of the aborted transactions are not included in $\Pi(\mathcal{H}_t)$. Now, using the definition of linearization points defined above and using $\widehat{t_c}$ wherever necessary, the sets of proofs given in [1] can be applied to $\mathcal{H}_t$. Further, observe that checking for compatibility at the time of merging (line 52) ensures that $\Pi(\mathcal{H}_t)$ does not include incompatible transactions. Thus, we can prove the correctness of committed transactions at each node.

Aborted transactions: For proving the correctness, we consider only one aborted transaction t_a at a time. Further, in case t_a is incompatible with t (or some transaction that merged with t) at time τ_t^1, then we consider the part of $\mathcal{H}_t$ up to a time just before τ_t^1. Note that t_a cannot have performed any read operation on t's object after time τ_t^1 due to compatibility checking at the time of reading (line 32).

To show the linearization point of t_a at its parent level, we consider its read steps only, and treat it as a committed read only transaction. Now, t_a being a committed transaction, we consider it in conjunction with the history of committed transactions, and prove the correctness in the same way as done for committed transactions. Further, considering t_a is a child of t, observe that t_a does not update its parent's object (line 53 precedes lines 54-67). Thus, results of an aborted transaction t_a are not propagated upward in the transaction tree.

7 Related Work

Moss and Hosking discussed the reference model for closed and open nesting in transactional memory and described preliminary architectural sketches [12]. In addition, they proposed a simpler model called linear nesting in which nested transactions run sequentially.

Recently, focus has been on supporting nested parallelism in STM [2, 3, 4, 10]. Agrawal et al. proposed CWSTM, a theoretical STM algorithm that supports nested parallel transactions with the lowest upper bound of time complexity [2]. In [4], Barreto et al. proposed a practical implementation of the CWSTM algorithm which achieves depth-independent time complexity of TM barriers. However, their work builds upon rather complex data structures such as concurrent stacks that could introduce additional runtime and state overheads [4]. Baek et. al. provide yet another implementation for supporting nested parallelism. Finally, Volos et al. proposed NePaLTM that supports nested parallelism inside transactions [10]. While efficiently supporting nested parallelism when no or low transactional synchronization is used, NePaLTM serially executes nested parallel transactions using mutual exclusion locks. All of these works are based on the assumption (constraint) that the parent transaction does not execute while it has active child subtransactions. In contrast, HParSTM enhances concurrency by allowing all the nodes in the transaction tree to execute in parallel.

8 Conclusion

This paper presents an STM protocol for supporting parallelism at all the levels of nested transactions i.e. between sibling transactions as well as parent and child transactions. It also offers an unique approach for contention management for nested transactions across different levels by using control variables at each level of nesting. HParSTM relies upon unbounded transaction identifier space, and hence may suffer from memory overheads. However, this issue could be addressed by employing the idea proposed in [4] for efficiently reusing transaction identifiers under restricted number of worker threads. Finally, our work can be useful in developing new research directions in this area.

References

[1] Damien Imbs, Michel Raynal. *A Lock-Based STM Protocol That Satisfies Opacity and Progressiveness*, OPODIS 2008, LNCS 5401, pp. 226-245, 2008.

[2] K. Agrawal, J. T. Fineman, and J. Sukha. *Nested parallelism in transactional memory.* In PPoPP 08: Proceedings of the 13th ACM SIG- PLAN Symposium on Principles and Practice of Parallel Program- ming, pages 163-174, 2008.

[3] W. Baek and C. Kozyrakis. *NesTM: Implementing and Evaluating Nested Parallelism in Software Transactional Memory.* In Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2009.

[4] J. Barreto, A. Dragojevic, P. Ferreira, R. Guerraoui, and M. Kapalka. *Leveraging parallel nesting in transactional memory.* In PPoPP 10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 91100, New York, NY, USA, 2010.

[5] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. *Implementing and Evaluating Nested Parallelism in Software Transactional Memory.* In SPAA'10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures, pages 253-262, June 2010.

[6] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. *Composable memory transactions.* In PPoPP 05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Program- ming, pages 48-60, 2005.

[7] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. *Supporting Nested Transactional Memory in LogTM.* SIGPLAN Notices (Proceedings of the 2006 ASPLOS Conference), 41(11):359-370, 2006.

[8] Theo Harder, Kurt Rothermel, *Concurrency control issues in nested transactions*, The VLDB Journal- The International Journal on Very Large Data Bases, v.2 n.1, p.39-74, January 1993.

[9] Felber, P., Gramoli, V. and Guerraoui, R. *Elastic Transactions.* 23rd International Symposium on Distributed Computing, pp. 93-107. DISC 2009.

[10] H. Volos, A. Welc, A.-R. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. *NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems.* In ECOOP, 2009.

[11] R. Guerraoui, M. Kapalka, *On the correctness of transactional memory.* In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, pages 175-184, 2008.

[12] J. E. B. Moss and T. Hosking. *Nested Transactional Memory: Model and Preliminary Architecture Sketches.* In OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL). University of Rochester, October 2005.