Approximately Opaque Multi-version Transactional Memory

Basem Assiri
Louisana State University
Baton Rouge, LA 70803, USA
bassir1@lsu.edu

Costas Busch
Louisana State University
Baton Rouge, LA 70803, USA
busch@csc.lsu.edu

Abstract

In multi-version transactional memory read-only transactions do not have to abort, while update transactions may abort. There are situations where system delays do not allow to have precise consistency, such as in large scale network and database applications, due to network delays or other factors. In order to cope with such systems, we introduce here the notion of approximate consistency in transactional memory. We define *K-opacity* as a relaxed consistency property where read instructions in a read-only transaction may read one of K most recent written values, while read instructions in an update transaction read always the latest value. The relaxed consistency for read-only transactions has two benefits: (i) it reduces space requirements, since a new object version is saved once every K object updates, which reduces the total number of saved object versions by a factor of K, and (ii) it reduces the number of aborts, since there is smaller chance for read-only transactions to abort update transactions. This framework allows to have worst-case consistency guarantees and simultaneously good performance characteristics. In addition to correctness proofs, we demonstrate the performance benefits of our approach with experimental analysis. We tested our algorithm for different values of K using different benchmarks and we observed that when we increase K the number of aborts decreases and at the same time the throughput increases.

I. Introduction

Software Transactional Memory (STM) is a very important paradigm that supports parallel computing [11]. STMs use the principle of shared memory transaction which is a finite sequence of instructions that read or write local and shared memory [8]. *Read-only* transactions have only read instructions, while *update* transactions have at least one write instruction. When all instructions are executed, a transaction commits or aborts depending on shared memory conflicts.

For many applications and algorithms it is beneficial to avoid the abort of read-only transactions since they do not affect directly the consistency of the memory. In order to minimize aborts, *multi-version* STMs keep multiple versions for each object in the memory [2]. In this way, when an update transaction commits, it creates new versions for the objects in its write set. With object versions read-only transactions do not have to abort, while transactions that update objects may abort. However, using multi-version STMs increases significantly the systems' space complexity.

Correctness in transactional memory is proven with *opacity* [7], which is a consistency property that requires a legal serialization of an execution such that transaction intervals do not overlap (atomicity), and read instructions always return the most recent value (legality). In order to improve the performance of multi-version STMs, we propose to relax the definition of opacity in a way that allows read instructions to access stale versions.

Actually, relaxing opacity to avoid some aborts is sometimes necessary on non-sensitive data and non-sensitive systems, or where data changes frequently. In large scale network systems when an update happens to an object in a local memory, it takes some time to update the object's global view. In fact, some read instructions might be executed on other local copies during such delay (between the updates of local and global memory) which makes them illegal and causes aborts. Also long read-only transactions may cause many aborts. In addition, in real life, there are some types of systems that do not require precise computations [12][10]. For example inventory queries, such as through Online Analytical Processing (OLAP), can return approximated results to nested and complicated database queries [3][12][6]. Decision Support Systems also work with approximated results such as queries about average income and the percentage of newborns in the country [1]. There is also no risk for advertising and recommendation systems to have approximated results (suggesting inaccurate restaurant or song would not harm). Moreover, sensors (for temperatures or weather forecasting) usually give approximated reads and that satisfy the specification of some systems.

In this paper we introduce the notion of *approximately opaque* consistency in order to design a multi-version transactional memory algorithm, where transactions are still atomic, but read instructions in read-only transactions may return one of the K most recent written values. We say that that these read instructions are K-legal, and also the respective execution is K-opaque. This allows to create a new object version once every K committed updates on the object, which reduces the total number of saved object versions by a factor of K. Moreover, K-opacity reduces the chance to abort transactions that update objects as the concurrent read instructions in read-only transactions may return one of the $K-1$ older values. In our algorithm we apply K-opacity on read-only transactions, while update transactions access only the latest value of an object (they are precise), which is necessary to avoid propagation of wrong values. This is beneficial in applications which do not require precise results from read-only transactions (for example, inventory checking).

We give a formal proof that our algorithm is K-opaque for read-only transactions and opaque (1-opaque) for update transactions. We also demonstrate the performance benefits of our approach with experimental analysis. We tested our algorithm for different values of K using different benchmarks and we observed that when we increase K the number of aborts decreases and at the same time the throughput increases.

II. System Model

In our algorithm, when a transaction T arrives, it gets a unique timestamp i which is also used as an identifier, namely T_i. Each instruction within a transaction corresponds to two events, its invocation and response. Every event is instantaneous and it has a real time that it occurs. The time between the invocation and its response is the instruction interval.

A history H is a sequence that includes all events of the involved transactions. H is called complete if all transactions within H are either committed or aborted [7]. With respect to history H we can define the order $<_H$ of the transactions such that for each two transactions T_i and T_j in H, $T_i <_H T_j$ if all events of T_i appear before all events of T_j. We say that the relation $<_H$ respects the *real time order* of the transactions. Note that the relation $<_H$ may be a partial order on T_i and T_j in H if there is an execution overlap between the transactions.

A history S is sequential if it is complete, and $<_S$ is a total order. Also, we say that two histories are $equivalent$ if they have the same set of events. Suppose that S is equivalent to H. We say that S preserves the *real time order* of H if for any two transactions T_i and T_j, $T_i <_H T_j$ implies $T_i <_S T_j$.

Now, to define the legality, a *legal read instruction* is the one that reads the last written value to an object and that value was written by a committed transaction with a smaller timestamp. However, a *K-legal read instruction* is one that reads the value of the object from any of the K last writes to that object with respect to the transaction timestamp (means those last K writes that belong to transactions with timestamps smaller than the timestamp of the reader transaction). Thus, the execution of transaction T_i is K-legal, if all read instructions in T_i are K-legal. Then, S is K-legal if all transactions in S are K-legal. Consequently, a legal history is a special case of K-legal history by taking $K = 1$. (We will distinguish between K-legality and 1-legality for read-only and update transactions, respectively, in our algorithm analysis below.)

A history H is *opaque* if it can be transformed to a complete history H' (by handling pending transactions) which has an equivalent legal sequential history S which further preserves the real time order $<_{H'}$ for the involved transactions. Similarly, H is *K-opaque* if it can be transformed to a complete history H' which has an equivalent K-legal sequential history S which preserves the real time order $<_{H'}$ for the involved transactions.

III. Design of the Algorithm

Our multi-version algorithm (Algorithm 1 in appendix) is timestamp-based as in previous works that do not consider approximate opacity [9][4]. Each object o has multiple versions that are stored in a list $o.vl$. We denote a version of object o as $v_i = (ts, data, rl)$, where ts is the timestamp of the transaction that creates (writes) this version, $data$ is the value of o, and rl is a reader list that includes the timestamps of all transactions that have been reading this version. In our algorithm, we create a new version of o each K commits and we save it in $o.vl$. The last written value is maintained in $o.lastCommit = (ts, data, rl)$ which is an independent version that is overwritten with each commit on o to record the last written value.

In many systems, the kinds of transactions (read-only or update) are identified at the beginning such as read balance and bank statements in bank systems, or product quantities in inventory systems. The read instruction in read-only transaction T_i tries to read the last written value in $o.lastCommit$ if $o.lastCommit.ts$ is smaller than its own (in Algorithm 2 in appendix). Otherwise, if $o.lastCommist.ts > i$, then it reads from a suitable saved version in $o.vl$. Also, it adds T_i timestamp to that version's rl. When T_i finishes the execution of all instructions, it commits directly.

For an update transaction T_i, for any read instruction it checks only $o.lastCommit$ and adds i to $o.lastCommit.rl$ (Algorithm 2). Then, as shown in Algorithm 1, if $o.lastCommit.ts > i$, T_i aborts immediately. Otherwise, it gets $o.lastCommit.ts$ (for validation) and reads the data of o. For the write instructions the transaction T_i just writes to its local memory and it maintains its own write set $wSet$ during the execution.

When the update transaction T_i finishes the execution of all instructions, it attempts to commit by calling TryC (Algotithm 3 in appendix). For any object o that was read by T_i, if the $o.lastCommit$ has been overwritten, then T_i aborts. Moreover, T_i aborts if it has a write that invalidates another transaction T_m where $m > i$ (Algorithm 4 in appendix). In TryC, T_i locks each object o in its $wSet$ and if it commits it overwrites the $o.lastCommit$ version and updates $o.lastCommit.ts = i$. We create a new version in $o.vl$ only every K commits of the object o. We let the new version's ts to be equal to i. After that, we release all locks. In our algorithm read-only transactions never abort, while update transactions may abort. We have also implemented a garbage collection algorithm for unneeded object versions (see appendix).

IV. Correctness of the Algorithm

In the correctness analysis we prove that our algorithm is opaque for update transactions, and K-opaque for read-only transactions. Let H be an arbitrary execution history, and H' the respective complete history. Consider the sequential execution S which is a serialization of the transactions in H' such that the order of transactions is determined by the timestamps of the transactions, such that if in H' for any two transaction T_i and T_j, $i < j$, then $T_i <_s T_j$.

Lemma 1: S preserves the real time order of H'.

Proof: According to Algorithm 1, for a transaction T_i the timestamp i is obtained through an atomic operation $i \leftarrow$ $timestamp$.getAndInc(); If $T_i <_{H'} T_j$ then, it has to be that $i < j$. Since S orders transactions in the timestamp order, then we also have that $T_i <_S T_j$, as needed. ■

We continue to prove that S is K-legal with respect to any object o for read-only transactions, and then we prove that it is 1-legal for update transactions.

Lemma 2: For any object o, the history S is K-legal with respect to read-only transactions accessing o.

Proof: Let T_i be a read-only transaction. Note that in our algorithm read-only transactions do not abort, and hence T_i does not abort. Suppose T_i executes instruction $o.r_i(y)$. According to function GetLatestVersion(), we have that T_i observes either $o.lastCommit.ts < i$ or $o.lastCommit.ts > i$. We examine these two cases separately.

i. $o.lastCommit.ts < i$:
then GetLatestVersion() returns $o.lastCommit$ and data $y = o.lastCommit.data$, which is the latest version of the object at that moment when o is accessed by T_i. Let T_j be the transaction that committed the value, that is, $j = o.lastCommit.ts < i$. Since S preserves the timestamp order, T_j appears before T_i in S. Suppose that there is another transaction T_k, with $j < k < i$, that commits a value to object o. If T_k commits after $o.r_i(y)$ locks o (in GetLatestVersion()), then according to function Validate(), T_k has to abort because T_i is in the reader list of o when T_k attempts to commit. On the other hand, if T_k commits before $o.r_i(y)$ locks o, then T_i must have read the value committed by T_k, or in other words $T_k = T_j$.

ii. $o.lastCommit.ts > i$:
then GetLatestVersion() returns a version v, and data $y = v.data$, where v belongs to version list $o.vl$ and it is the latest version of o with timestamp $v.ts = j < i$. Let T_j be the transaction that created version v. We need to prove that there cannot be more than $K-1$ other committed transactions for object o between the time that T_j commits and T_i starts in H'. We observe that any transaction T_k that commits a value for o after T_j must have timestamp $k > j$, since otherwise the interval of T_k would contain the interval of T_j in H', and according to function Validate() T_k would abort. We have that in S, T_j appears before T_i, since $j < i$. Let X be the set of transactions which appear in S between T_j and T_i and commit a value for o (for any $T_k \in X$ it holds $j < k < i$). We want to show that $|X| \leq K-1$. Similar to the reasons explained above in case i, X cannot contain any transaction which commits after $o.r_i(y)$ locks o. Moreover, X cannot contain any transaction T_k that commits before T_j, since in H' interval T_j would contain interval T_k and according to function Validate() T_j would abort. Hence, all the transactions in X must have timestamp greater than j and must commit in H' after T_j. If $|X| \geq K$, according to our algorithm, a newer version v' of o must have been saved (in $o.vl$) after T_j commits and before T_i starts, by some transaction $T_\zeta \in X$. However, this is impossible, since T_i would have read v' and not v.

Therefore, we have that in case i execution S is 1-legal, while in case ii the execution S is K-legal. ■

Lemma 3: For any object o, the history S is 1-legal with respect to update transactions accessing o.

Proof: Now consider the update transactions that access object o. Let T_i be an update transaction that invokes instruction $o.r_i(y)$. According to the algorithm, if $o.lastCommit.ts > i$, then T_i is aborted and instruction $o.r_i(y)$ never completed. On the other hand, if $o.lastCommit.ts < i$ then $o.r_i(y)$ completes with $y = o.lastCommit.data$. Let $j = o.lastCommit.ts$, namely, T_i reads the value written by T_j, with $j < i$. Similar to the proof of Lemma 2, any transaction T_k that commits a value for o after T_j must have timestamp $k > j$.

In S transaction T_j appears before T_i. We only need to show that in S there is no other committed transaction between T_j and T_i for object o. Suppose that there is a transaction T_k, with $j < k < i$, which appears between T_j and T_i in S and commits a value to o. If T_k commits before T_j in H', function Validate() would cause to abort T_j. If T_k commits before $o.r_i(y)$ locks object o, then T_i must have used the value committed by T_k. On the other hand, if T_k commits after $o.r_i(y)$ locks object o, then according to function Validate() T_k has to abort, since T_i is in the reader list of o when T_k attempts to commit, and $i > k$. ■

Since H' respects the real time order of H, considering all objects used in H, from Lemmas 1, 2 and 3 we obtain the following theorem.

Theorem 4: Any execution history H of our algorithm is K-opaque with respect to read-only transactions and 1-opaque with respect to update transactions.

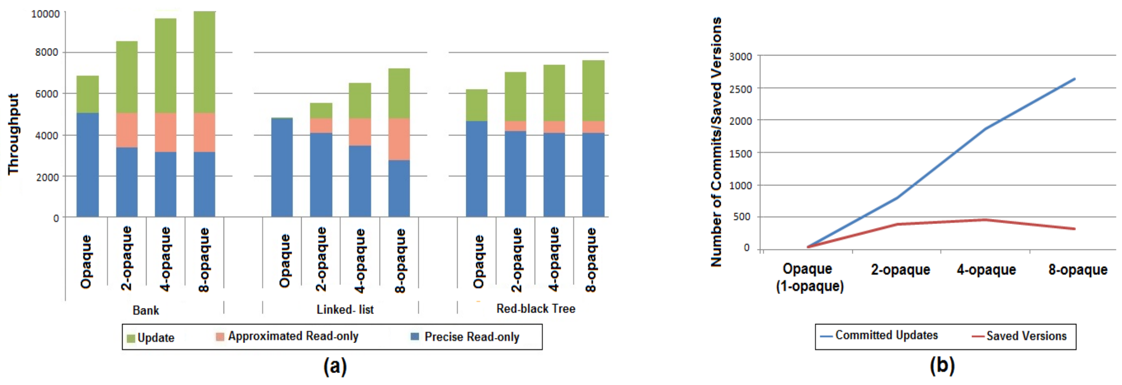

Figure 1: (a) Compare the Throughputs (Committed Transactions Per Time) of Opaque, 2-opaque, 4-opaque and 8-opaque Using Bank, Linked-list and Red-black Tree Benchmarks, (b) Compare the Number of Committed Updates to the Number of the Saved Versions in Opaque, 2-opaque, 4-opaque and 8-opaque Using Linked-list Benchmark

It is easy to check that the proposed algorithm does not deadlock, since function GetLatestVersion() accesses one object at a time, and function TryC() accesses objects in a predetermined order, avoiding racing situations.

Lemma 5: Our algorithm does not deadlock.

Assume that the transactions in our algorithm access the set of objects $O = (o_1, o_2, ..., o_n)$. Let V be the set of all committed versions (updating $o_i.lastCommit$) and V' the set of all saved versions (saved in $o_i.vl$).

Theorem 6: In any execution of our Algorithm, the total number of saved object versions is $|V'| = \Theta(|V|/K + |O|)$.

Proof: Throughout the execution, for any object o_i let the number of committed versions be v_{o_i} (updating $o_i.lastCommit$). The total number of committed versions for all objects is $|V| = \sum_{i=1}^{n} v_{o_i}$. Our algorithm saves a new version for object o_i each K object commits. Thus, the total number of saved versions for object o_i (saved in $o_i.vl$) is v_{o_i}/K, and consequently, the total number of saved versions $|V'|$ for all objects will be $|V|/K$. In addition, each object has a $lastCommit$ version which adds a number of $|O|$ versions to $|V|/K$. ■

Now, if we exclude the last committed versions, the total version space of regular (not ours) multi-version transactional memory is $\Theta(|V|)$, since every version is saved at some point of time. On the other hand, from Theorem 6, with our approximately opaque multi-version algorithm we only create a new version each K commits reducing this number to $\Theta(|V|/K)$, a reduction by a factor of K. Furthermore, with the garbage collector, old versions are deleted which reduces the active number of $|V'|$ at any point of the execution.

V. Experimental Results

In our experimental analysis, we simulate Bank, Linked-list and Red-black Tree Benchmarks from TinySTM-1.0.5 [5], but we modify the structure of the object to match our specification. We run the experiments on a machine with dual Intel(R) Xeon(R) CPU E5-2630 (6 cores total) clocked at 2.30 GHz. Each run of the benchmark takes about 5500 milliseconds using 10 threads. In Bank benchmark, there are three kinds of operations which are read balance, write amount and transfer. In the Linked-list and Red-black Tree benchmarks, we have search operations, add and delete node. Read balance (in Bank) and search (in Linled-list and Red-black Tree) are read-only, but write, transfer (in Bank) and add/delete node (in Linled-list and Red-black Tree) are update transactions. In our execution we generate 50% reads-only transactions and 50% update ones.

In Figure 1(a), we compare the throughput (commits per time) of an opaque execution (1-opaque), 2-opaque, 4-opaque and 8-opaque using the three benchmarks. Clearly, the relaxed opacity in 2-opaque, 4-opaque and 8-opaque helps to avoid some aborts and to improve the throughput. Furthermore, in 1-opaque all read-only transactions are precise but in 2-opaque, 4-opaque and 8-opaque the percentage of approximated read-only transactions is smaller than the percentage of the precise ones. We note that there is an increase in the number of committed updates since relaxing the opacity of a read-only transaction sometimes allows to avoid many aborts; as 1-opaque read-only transaction may conflict with many update ones.

Figure 1(b) shows a comparison between the number of committed updates and the number of the saved versions using Linked-list Benchmark. In 1-opaque the number of committed updates and the number of the saved versions are the same, since we save a new version with each committed update. In 2-opaque and 4-opaque, the number of saved version increases because such relaxations allow to commit very large number of updates. However, in 8-opaque the number of committed updates increases but the number of non-saved versions is very large (as we save 1 version every 8 commits), so the number of saved versions decreases.

References

[1] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. Aqua: A fast decision support systems using approximate query answers. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 754–757. Morgan Kaufmann Publishers Inc., 1999.

[2] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[3] Edgar F Codd, Sharon B Codd, and Clynch T Salley. Providing olap (on-line analytical processing) to user-analysts: An it mandate. *Codd and Date*, 32, 1993.

[4] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, 2010.

[5] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[6] Jim N Gray, Raymond A Lorie, Gianfranco R Putzolu, and Irving L Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[7] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[8] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[9] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version stm algorithm. In *Distributed Computing and Networking*, pages 212–226. Springer Berlin Heidelberg, 2014.

[10] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3):9, 2014.

[11] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[12] Mukul K Sinha. Nonsensitive data and approximate transactions. *IEEE Transactions on Software Engineering*, (3):314–322, 1983.

Appendix

A. The Algorithm

Algorithm 1: K-opaque Algorithm

```
/* global variable initialization */
$timestamp \leftarrow 0$;
$liveT \leftarrow \emptyset$;
foreach transaction $T_i$ do
    /* $i$ gets a unique timestamp */
    $i \leftarrow timestamp.getAndInc()$;
    $T_i.status \leftarrow active$;
    $T_i.wSet \leftarrow \emptyset$;
    while there is an unexecuted instruction $x$ do
        /* if instruction is read */
        if $x = o.r(y)$ then
            $v \leftarrow$ GetLatestVersion$(i, o)$;
            if $v.ts > i$ then
                /* this check only for read instruction in update transaction to have immediate abort */
                $T_i.status \leftarrow aborted$;
                return;
            $y \leftarrow v.data$;
            $o_{local}.ts \leftarrow v.ts$; //$o_{local}$ is the object $o$ in the local memory
        else
            /* $x = o.w(y)$; write local copy */
            $o_{local}.data \leftarrow y$;
            $T_i.wSet \leftarrow o \cup T_i.wSet$;
    if TryC$(i)$ then
        $T_i.status \leftarrow committed$;
    else
        $T_i.status \leftarrow aborted$;
    return;
```

Algorithm 2: GetLatestVersion(i, o)

```
$last \leftarrow null$;
Lock $o$;
if $T_i.kind = readonly$ then
    if $o.lastCommit.ts < i$ then
        $last \leftarrow o.lastCommit$;
        Add $i$ to list $o.lastCommit.rl$;
    else
        $v \leftarrow$ the most recent version in $o.vl$ with timestamp smaller than $i$;
        $last \leftarrow v$;
        Add $i$ to list $v.rl$;
else
    /* update transaction */
    $last \leftarrow o.lastCommit$;
    Add $i$ to list $o.lastCommit.rl$;
Unlock $o$;
return $last$;
```

Algorithm 3: TryC(i, o)

```
/* check if $T_i$ is readonly */
if $T_i.kind = readonly$ then
    /* remove $T_i$ from $liveT$ */
    $liveT \leftarrow liveT \setminus i$;
    return $true$;
/* $T_i$ has to be update transaction */
$L \leftarrow \emptyset$;
/* assume a predetermined order for the objects */ forall the $o \in i.wSet$ do
    Lock $o$;
    $L \leftarrow L \cup o$;
    if Validate$(i, o) = false$ then
        $liveT \leftarrow liveT \setminus i$;
        unlock all locked objects in $L$;
        return $false$;
forall the $o$ in $i.wSet$ do
    $o.versionCounter$.getAndInc();
    if $o.versionCounter \mod k = 0$ then
        /* add new version to $o.vl$ */
        Add $(i, o_{local}.data, nil)$ to $o.vl$;
    /* overwrite $o.lastCommit$ */
    $o.lastCommit \leftarrow (i, o_{local}.data, nil)$;
Unlock all objects in $L$;
return $true$;
```

Algorithm 4: Validate(i, o)

```
/* Check if $lastCommit$ has been overwritten */
if $o.lastCommit.ts > o_{local}.ts$ then
    return $false$;
/* Check if some other transaction $T_m$ has read the same version read by $T_i$, where $m > o_{local.ts}$ */
if $o.lastCommit.rl$ contains a transaction $T_m$, where $m > i$ then
    return $false$;
return $true$;
```