Permissiveness in Transactional Memories[*]

Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh

EPFL, Switzerland

Abstract. We introduce the notion of permissiveness in transactional memories (TM). Intuitively, a TM is permissive if it never aborts a transaction when it need not. More specifically, a TM is permissive with respect to a safety property p if the TM accepts every history that satisfies p. Permissiveness, like safety and liveness, can be used as a metric to compare TMs. We illustrate that it is impractical to achieve permissiveness deterministically, and then show how randomization can be used to achieve permissiveness efficiently. We introduce Adaptive Validation STM (AVSTM), which is probabilistically permissive with respect to opacity; that is, every opaque history is accepted by AVSTM with positive probability. Moreover, AVSTM guarantees lock freedom. Owing to its permissiveness, AVSTM outperforms other STMs by upto 40% in read dominated workloads in high contention scenarios. But, in low contention scenarios, the bookkeeping done by AVSTM to achieve permissiveness makes it, on average, 20-30% worse than existing STMs.

1 Introduction

Transactional memory (TM) tries to maximize concurrency in an implementation while providing the illusion of sequentiality to the programmer. It holds the promise to exploit the computational power of modern multi-processor architectures within the security afforded by a simple, non-concurrent programming model. A transaction is an atomic program that can commit its actions to memory, or abort without changing the memory. An abort can be caused by the programmer (say, if some exception is raised), or by the TM itself, if there is a risk to violate the correctness of the memory. Typically, this correctness is expressed by some form of serialization for transactions; that is, a transaction can commit only if the state of the memory could have been generated by some sequential execution of the transactions so far.

At first glance, one would expect from a TM that it never aborts a transaction when it need not, i.e., when there is no risk of violating correctness. It turns out, however, that proposed TMs have certain scenarios where a transaction is aborted even if it could have committed without violating correctness. In other words, these TMs do not enable the maximal amount of possible concurrency among a set of transactions. This observation naturally raises the question of whether one can devise an ideal, maximal TM, which never aborts a transaction unless necessary for correctness. We call such a TM *permissive*.

[*] This research was supported by the Swiss National Science Foundation.

In this paper, we formalize the notion of permissiveness and discuss why the existing TM implementations are not permissive. We argue that permissiveness is expensive to achieve in a deterministic TM. We then present a randomized permissive TM. We show in particular that using randomization in choosing the serialization point of every transaction creates an efficient permissive STM.

Formally, a deterministic TM is an online algorithm that is given a sequence of statements and decides for each statement, based on the statements so far, whether or not to accept the statement. A deterministic TM is *permissive* with respect to a given safety property (e.g., serializability) if every history (finite sequence of statements) that satisfies the safety property is accepted by the TM. In Section 2, we show that existing TMs, like TL2 [1], WSTM [3], and DSTM [9], are not permissive with respect to serializability or opacity, a strong form of serializability that arguably corresponds to what should be expected from a TM [7, 9]. Opacity captures the practical notion in TM that transactions execute serially, and even aborting transactions do not view inconsistent state. To our knowledge, the only deterministic TM permissive with respect to serializability or opacity occurs in our recent work [5]. This TM is built using the notion of conflict graphs [10]. But the conflict graph changes globally with every statement. Capturing this change incurs a high cost per statement, and the feasibility of a practical deterministic permissive TM remains questionable.

For randomized TMs, it is natural to consider weaker, probabilistic notions of permissiveness. Formally, a randomized TM is an online algorithm that is given a sequence of statements and decides for each statement, based on a random coin toss, whether to serialize the transaction at the current statement, and based on the statements so far, whether or not to accept the statement. We say that a randomized TM is permissive with respect to a safety property if every history that satisfies the safety property is accepted by the TM with probability 1. Moreover, we say that a randomized TM is *probabilistically permissive* with respect to a safety property if every history that satisfies the safety property is accepted by the TM with positive probability. We do not know of any existing randomized TM that is permissive, or probabilistically permissive, with respect to serializability or opacity. We present Adaptive Validation STM (AVSTM), which is probabilistically permissive for opacity. AVSTM can be configured to be probabilistically permissive for serializability too. AVSTM uses randomization to determine an ordering (serialization) point during the life-time of each transaction. We have designed AVSTM in such a manner that it guarantees lock freedom; that is, infinitely many transactions commit in every infinite history produced by AVSTM.

Typically, the efficiency of a TM is measured by the number of transactions that commit per time unit. So, in theory, putting the bookkeeping aside, a more permissive TM should also be more efficient, as it aborts less often. We evaluat this claim in practice by implementing an AVSTM prototype and comparing its performance to existing TMs. Our evaluation on a multi-processor architecture (4 processor dual-core running Linux) shows that, indeed, AVSTM outperforms existing TMs (such as DSTM, WSTM, TL2) by upto 40% in high-contention scenarios, where many processes are accessing a small set of variables. In low-contention scenarios, AVSTM does not outperform the most efficient TMs, and suffers performance by 20-30%. This is due to the amount of bookkeeping used by AVSTM, which turns out to be more expensive than a few unnecessary aborts in low-contention scenarios. We present a simple scheme to compose TL2 with AVSTM obtaining the advantages of both algorithms. In short, the processes run by default TL2 and dynamically switch to AVSTM when contention increases.

Related work. Many STMs [1, 3, 8, 9, 11] have been proposed in the literature. Most of these guarantee opacity, but none of them is opacity-permissive. Existing STMs guarantee different levels of liveness. DSTM [9] guarantees obstruction freedom. Many contention managers [6, 12] have been proposed to boost obstruction freedom. But, in our knowledge, there is no contention manager that boosts obstruction freedom to yield lock freedom. WSTM [3] guarantees lock freedom.

2 Framework

We formalize the notion of safety and permissiveness in transactional memories.

Preliminaries. Let V be a set $\{1, \ldots, k\}$ of k *variables*. Let $C = \{\mathsf{commit}\} \cup \{\mathsf{abort}\} \cup (\{\mathsf{read}, \mathsf{write}\} \times V)$ be the set of *commands* on the variables in V. For our formalism, we treat these commands as atomic. Let $P = \{1, \ldots, n\}$ be the set of *processes*. Let $\Sigma = C \times P$ be a finite alphabet of statements. A *history* $h \in \Sigma^*$ is a finite sequence of statements in Σ. Given a history h, we define the *projection* $h|_p$ of h on process $p \in P$ as the longest subsequence h' of h such that every statement in h' is in $C \times \{p\}$. Given a projection $h|_p = \sigma_0 \ldots \sigma_m$ of a history h, a statement σ_i is *finishing* in $h|_p$ if it is a commit or an abort. A statement σ_i is *initiating* in $h|_p$ if it is the first statement in $h|_p$, or the previous statement σ_{i-1} is a finishing statement.

Given a projection $h|_p$ of a history h on process p, a consecutive subsequence $x = \sigma_0 \ldots \sigma_m$ of $h|_p$ is a *transaction* of process p in h if (i) σ_0 is initiating in $h|_p$, and (ii) σ_m is either finishing in $h|_p$, or σ_m is the last statement in $h|_p$, and (iii) no other statement in x is finishing in $h|_p$. The transaction x is *committing* in h if σ_m is a commit statement. Given a history h and two transactions x and y in h (possibly of different processes), we say that $x <_h y$ if the finishing statement of x occurs before the initiating statement of y in h. A history h is *sequential* if for every pair (x, y) of transactions in h, either $x <_h y$ or $y <_h x$.

We define a function $com : \Sigma^* \to \Sigma^*$ such that for all histories $h \in \Sigma^*$, the history $com(h)$ is the longest subsequence h' of h such that every statement in h' is part of a committing transaction in h. Thus, $com(h)$ consists of all statements of all committing transactions in h. A statement $\sigma = ((\mathsf{read}, v), p)$ in x is a *global read* of the variable v if there is no write to v before σ in x.

Safety properties. Strict serializability [10] is a commonly used correctness criterion for concurrent systems and, in particular, for transactional systems. In the scope of STMs, a stronger notion of correctness, referred to as *opacity* has been suggested [7, 9] to avoid unexpected side effects, like infinite loops, or array bound violations due to aborting transactions. Opacity requires that a history is strictly serializable, and that the aborting transactions do not see inconsistent values. Transactional memories use direct update semantics (every transaction modifies the shared variables in place and restores them in case of abort), or deferred update semantics (every transaction modifies a local copy, and changes the shared copy upon a commit). We define the notion of a conflict under the deferred update semantics. A statement σ_1 of transaction x and a statement σ_2 of transaction y $(x \neq y)$ *conflict* in a history h if (i) σ_1 is a global read of variable v and σ_2 is a commit and y writes to v, or (ii) σ_1 and σ_2 are both commits, and x and y write to v. Note that the definition of a conflict would be different with direct update semantics.

A history $h = \sigma_0 \dots \sigma_m$ is *strictly equivalent* to a history h' if (i) $h|_p = h'|_p$ for all processes $p \in P$, and (ii) for every pair σ_i, σ_j of statements in h, if σ_i and σ_j conflict and $i < j$, then σ_i occurs before σ_j in h', and (iii) for every pair x, y of transactions in h, if $x <_h y$ then it is not the case that $y <_{h'} x$. A history $h \in \Sigma^*$ is *strictly serializable* if there exists a sequential history h' such that h' is strictly equivalent to $com(h)$. Furthermore, we define that a history h is *opaque* if there exists a sequential history h' such that h' is strictly equivalent to h. (Note that h may contain unfinished transactions.) We note that if a history h is opaque, then h is strictly serializable.

We define the safety property *strict serializability* $\pi_{ss} \subseteq \Sigma^*$ as the set of all strictly serializable histories, and the safety property *opacity* $\pi_{op} \subseteq \Sigma^*$ as the set of all opaque histories.

Transactional memories. We model transactional memories as transition systems, that consist of a set of states, an initial state, an alphabet of statements, and a transition relation between the states.

We define a *TM* $A = \langle Q, q_{init}, \Sigma, \delta \rangle$, where Q is a set of states, q_{init} is the initial state, Σ is the set of statements, and $\delta \subseteq Q \times \Sigma \times \Gamma \times Q$ is the transition relation, where $\Gamma = (0, 1]$ represents the probability of a transition. For all $q \in Q$ and $\sigma \in \Sigma$, if there are m outgoing transitions $(q, \sigma, \gamma_i, q') \in \delta$ with $1 \leq i \leq m$, then we have $\sum_i \gamma_i = 1$. A transition relation δ is *deterministic* if for all $q \in Q$ and $\sigma \in \Sigma$, if $(q, \sigma, \gamma_1, q_1) \in \delta$ and $(q, \sigma, \gamma_2, q_2) \in \delta$, then $q_1 = q_2$ and $\gamma_1 = \gamma_2$. Given a TM A, a sequence $q_0 \dots q_m$ of states is a *run* for a history $h = \sigma_0 \dots \sigma_m$ if (i) $q_0 = q_{init}$, and (ii) for all i such that $0 \leq i \leq m$, we have $(q_i, \sigma_i, \gamma, q_{i+1}) \in \delta$ where γ is positive. The outcome of a TM captures the probability of the different histories accepted by the TM. The *outcome* O_A of the TM A is a function $O_A : \Sigma^* \to [0, 1]$. Given a history h and a TM A, the outcome $O_A(h) = \gamma$ if there exist a set $\rho_1, \dots, \rho_m$ of runs for h with probabilities $\gamma_1 \dots \gamma_m$ such that $\sum_{0 \leq i \leq m} \gamma_i = \gamma$.

Safety and permissiveness of TM. We formalize the safety and permissiveness properties of TM, assuming that the commands in Σ occur atomically. A TM A is π-*safe* for a safety property $\pi \subseteq \Sigma^*$, if for every history $h \in \Sigma^*$ such that $O_A(h) > 0$, the history $h \in \pi$. In other words, a TM is safe with respect to a property if the outcome of the TM is positive only for histories that satisfy the property.

A TM A is *π-permissive* if for every history $h \in \pi$, we have $O_A(com(h)) = 1$. A TM A is *probabilistically π-permissive* if for every history $h \in \pi$, we have $O_A(com(h)) > 0$. Note that a deterministic TM is probabilistically permissive with respect to a property π if and only if it is permissive with respect to π. On the other hand, a randomized TM may not be π-permissive, while being probabilistically π-permissive.

We now show an example why the existing STMs are not permissive. Consider the history $h = ((\mathsf{write}, v_1), p_1), ((\mathsf{read}, v_1), p_2), ((\mathsf{write}, v_2), p_2)$ (commit, p_1), (commit, p_2). The history h is opaque, but its outcome is 0 for STMs like DSTM, TL2, and WSTM. In fact, it is easy to see that any TM that checks at the time of commit that the values of the variables read are equal to what values were read earlier (that is, validates the read set), cannot be permissive with respect to opacity. On the other hand, most of the existing TMs, for reasons of good overall performance, do exploit such a validation strategy to ensure safety. We now give algorithms that guarantee permissiveness in STMs.

3 Permissive Transactional Memories

We start with motivating the notion of permissiveness in TMs. TMs are online algorithms. That is, a TM decides whether to accept a statement or abort the corresponding transaction, only based on the statements seen so far. Let h be the history seen by the TM so far. Let x be the unfinished transaction of the process p in history h, and $h' = h \cdot (c, p)$. A TM A may decide to abort the transaction x in three scenarios:

- *Correctness.* The history h' is not opaque. In this case, any TM safe with respect to opacity needs to abort x. Thus, even a opacity-permissive TM aborts x.
- *Performance.* The history h' is opaque, but A is not sure whether $h \cdot (c, p)$ is opaque. For efficiency, A decides to abort x and retry it. In this case, a permissive TM will find out that h' is opaque, and thus not abort x. It is crucial for a permissive TM is to efficiently compute, given that h is opaque, if h' is opaque or not.
- *Priority.* The history h' is opaque, but the unfinished transaction y of some process $p' \neq p$ has to abort in every history extension of h'. The TM A prioritizes p' and hence aborts x, so that y retains the possibility to commit. In this case, we argue that as a TM does not know the future input after h, even after the TM aborts x, it is possible that due to conflicts, the TM has to abort y too. We believe that the idea of permissiveness can well be integrated with the notion of prioritizing certain processes, by making a process wait (rather than abort) for the commit of another process with higher priority.

Thus, the key to an efficient permissive TM for a given property lies in minimizing the cost of book-keeping required to check on the fly, whether the history produced by the TM satisfies the property.

3.1 A deterministic permissive transactional memory

In recent work [5], we described transactional memory specifications for strict serializability and opacity. The algorithm to obtain these specifications is based on the idea of conflicts graphs [10]. We refer to this algorithm as Spec in this paper. Spec assumes that the commands read, write, commit, and abort execute atomically. Our assumption is justified as we only analyze Spec and do not build a deterministic permissive TM implementation from it.

The important idea of the Spec is the prohibited read and write sets, which allow us to remove finished transactions from the conflict graph. This keeps the conflict graph finite. Spec [5] can be configured to obtain specifications for strict serializability or opacity. As the algorithm provides TM specifications for strict serializability (resp. opacity), it has outcome 1 for all and only those histories which are safe with respect to strict serializability (resp. opacity). In other words, we get a TM which is safe and permissive with respect to strict serializability or opacity.

The cost of the read operation in Spec for n processes is $O(n^2)$. In a practical scenario, most of the statements are reads, which in existing TMs, have a complexity of $O(n)$. Some highly performance oriented TMs, like TL2, just require $O(1)$ for a read statement. So, we believe that the high cost of a read operation in Spec makes it a poor choice for a practical implementation. Hence, we do not create a TM implementation from Spec. We open the question whether there exists a deterministic permissive TM where the read operation is at most linear in the number of processes. Here, we now describe a randomized STM, called Adaptive Validation STM (AVSTM), which is probabilistically permissive w.r.t. opacity, and at the same time, performs well practically. The algorithm derives its name from the fact that transactions adaptively validate themselves. All transactions maintain a possible interval to serialize themselves, and at the time of commit, randomly choose a point within that interval.

3.2 A randomized permissive transactional memory

Algorithm 1 shows the algorithm for AVSTM, which can be used for either strict serializability (SS-AVSTM) or opacity (OP-AVSTM). We do not assume that the commands in Σ are atomic in AVSTM. Only the reads and writes of global variables, and the CAS operation are treated as atomic. This allows us to use AVSTM as a real TM implementation, which we discuss in Section 3.5. The idea of AVSTM is to *randomly* choose, at the time of commit, a possible serialization point for every transaction. In principle, this allows transactions to probabilistically commit in the past, or in the future. For example, if two transactions x, y access the same variable, where x writes and y reads – even if x commits, y may commit afterwards if the transaction x chooses a serialization point in the future.

We first describe the variables and functions used in AVSTM. The function $rs : P \to 2^V$ is the read set and $ws : P \to 2^V$ is the write set of the processes. The function $rv : V \to \mathbb{N}$ gives the read version number and $wv : V \to \mathbb{N}$ gives the write version number of the variables. The function *Global* : $V \to \mathbb{N}$ is the global valuation of V, and *Local* : $P \times V \to \mathbb{N}$ is the process-local valuation of V. Moreover, the functions `min_ser_point` : $P \to \mathbb{N}$ and `max_ser_point` : $P \to \mathbb{N}$ denote the minimum and maximum possible serialization points for the processes respectively. The function `ser_point` : $P \to \mathbb{N}$ represents the tentatively chosen serialization point for the processes. AVSTM uses a variable `commit_num` $\in \mathbb{N}$ that represents the sequence number of the commit being performed, and a variable `owner` $\in P \cup \{\perp\}$ that denotes the process which owns the current commit. When no process is committing, then the owner is $\perp$. The commit sequence number and the owner process are treated as an atomic pair so that they can be atomically manipulated. For the implementation, we encode `commit_num` and `owner` within the same variable as `commit_num` $\cdot (n + 1) + i$, where n is the number of processes, and $i = 0$ if `owner` $=\perp$ and $i =$ `owner` otherwise. Also, every process uses a local variable `lcn` $\in \mathbb{N}$. The read set and write set of all processes are initially empty. Version numbers of all variables and serialization points of all processes are initially set to 0. When a transaction is started by process p, the variable `commit_num` is first read into the local variable `lcn`. Then, `lcn` is written into `ser_point`(p) and `min_ser_point`(p), and `max_ser_point`(p) is initialized to infinity. We now give an informal description of the algorithm.

- *Upon read of variable v by process p:* If v is in the write set of the process, then p returns the local value of v. Otherwise, the global value of v is copied as the local value of v for p. The value `ser_point`(p) is set to the maximum of the write version of v and its previous value. If some process p' is committing a transaction that writes to v, then p first helps p' to commit. Then, v is added to the read set of p. If the global value of v has not changed during this read operation, then the value read is returned. Otherwise, the read is performed again. To ensure opacity, p also needs to check whether it has a positive interval to commit.
- *Upon write of variable v with value* `val` *by process p:* The variable v is added to the write set. The local value of v in process p is set to `val` (overwritten if already existed). The value `ser_point`(p) is chosen as the maximum of its previous value, the write version of v and the read version of v.
- *Upon commit of a transaction by process p:* For a read only transaction, committing a transaction simply requires to set the read set to empty. Otherwise, the process p first aims to gain ownership of the commit. Until then, it helps other processes which are committing. p increments `ser_point`(p) to ensure that for every variable v in the write set, `ser_point`$(p) \geq rv(v)$. If there is no positive interval to commit, then the unfinished transaction of p is aborted. Once p obtains ownership of the commit (`owner` $= p$), the process p starts the *helpCommit* for itself. The commit of p may also be helped by other processes. Finally, the read and write sets of p are reset to empty.
- *Upon abort of a transaction by process p:* The read and write sets are reset to empty.

The procedure *helpCommit*(`lcn`$, p$) is shown in Algorithm 2. It allows a process to help a process p in committing a transaction as follows. First of all, the read version of all variables in $rs(p)$ is incremented to the `ser_point`(p). Then, the write version of all variables in $ws(p)$ is also incremented to `ser_point`(p). Then, the global value of these variables is updated. Then, the maximum serialization point of all processes whose read set intersects with the write set of p is set to `ser_point`(p). Similarly, the minimum serialization point of all processes whose write set intersects with the write set of p is set to `ser_point`(p). Last of all, the commit is made unowned (`owner` $=\perp$). We note that the version numbers increase monotonically. Also the epoch based storage management ensures that a pointer to a memory location is not freed if any process holds a reference to the location. Thus none of the CAS operations in AVSTM suffer from the ABA problem. We now analyze the safety, liveness, and permissiveness of AVSTM.

3.3 Safety and permissiveness of AVSTM

The order of operations in the read statement and in the procedure *helpCommit* is crucial for safety and permissiveness of AVSTM. We observe that the order of statements in the read and *helpCommit* procedure is essential for the safety and permissiveness of AVSTM. Upon a read of a variable v, the value of v is read (line 3) before the version number of v is read (line 4). Also, the read is successful only if the global value of v observed at the end of the read procedure (line 10) is same as the one read at the beginning of the read procedure (line 3). In the procedure *helpCommit*, for the variables being written, the write version number is updated (line 10) before the value of the variable is updated (line 12). Moreover, if the variable read by a process p is being written by some process p', then p first helps p' to commit. Together, this ensures that the version number read in line 4 of the read procedure corresponds exactly to that of the value read in line 3.

We prove the following properties of OP-AVSTM: safety and probabilistic permissiveness with respect to opacity. Similar proofs can be obtained for SS-AVSTM with respect to strict serializability. We also give an example of how OP-AVSTM works on a given opaque history.

Theorem 1. *OP-AVSTM is safe with respect to opacity.*

Proof. The procedure *helpCommit* allows many processes to commit a transaction for a particular process. But, the values and version numbers are committed exactly once, as the version numbers increase monotonically. Thus, the CAS does not suffer from the ABA problem. Also, the transaction x of process p commits only if it has a positive interval at the time of commit. For the case of opacity, it is also ensured that when a variable is read, then the transaction has a positive interval to commit. A transaction x of process p successfully reads a variable only if `max_ser_point` $>$ `min_ser_point`. Note that the order of operations in the read and *helpCommit* is crucial to this correctness. The read operation first reads the variable, and then reads the version number. The procedure *helpCommit* first updates the version number and then updates the variables. This order ensures that if a newer value of a variable is read, then its corresponding version number is

Algorithm 1 Adaptive Validation STM

```
Upon read of variable $v$ by process $p$
  if $v \in ws(p)$ then return $Local(p, v)$
  do forever
    $Local(p, v) := Global(v)$
    local_write_version := $wv(v)$
    ser_point($p$) := max(ser_point($p$), local_write_version)
    $\langle$lcn, $p'\rangle := \langle$commit_num, owner$\rangle$
    if $p' \neq \perp$ and $v \in ws(p')$ then $helpCommit$(lcn, $p'$)
    OP-AVSTM:
      if max(min_ser_point($p$), ser_point($p$)) $\geq$ max_ser_point($p$) then abort
    if $Local(p, v) = Global(v)$ then break
  $rs(p) := rs(p) \cup \{v\}$
  return $Local(p, v)$

Upon write of variable $v$ with value val by process $p$
  local_write_version := $wv(v)$
  local_read_version := $rv(v)$
  ser_point($p$) := max(ser_point($p$), local_write_version, local_read_version)
  $ws(p) := ws(p) \cup \{v\}$
  $Local(p, v)$ := val

Upon commit by process $p$
  if $ws(p) = \emptyset$ then $rs(p) := \emptyset$;    return
  do forever
    $\langle$lcn, $p'\rangle := \langle$commit_num, owner$\rangle$
    while $p' \neq \perp$ do
      $helpCommit$(lcn, $p'$)
      $\langle$lcn, $p'\rangle := \langle$commit_num, owner$\rangle$
    for each variable $v \in ws(p)$ do
      ser_point($p$) := max(ser_point($p$), $rv(v)$)
    ser_point($p$) := max(min_ser_point($p$), ser_point($p$))
    if ser_point($p$) $\geq$ max_ser_point($p$) then abort
    ser_point($p$) := a random number between ser_point($p$) and max_ser_point($p$)
    new_cp := lcn + 1
    if $CAS(\langle$commit_num, owner$\rangle$, $\langle$lcn, $\perp\rangle$, $\langle$new_cp, $p\rangle) = \langle$lcn, $\perp\rangle$ then break
  $helpCommit$(new_cp, $p$)
  $rs(p) := \emptyset$;    $ws(p) := \emptyset$

Upon abort by process $p$
  $rs(p) := \emptyset$;    $ws(p) := \emptyset$
```

also read. When a transaction x of a process p starts, the variable min_ser_point and ser_point are set to the serialization point of the last committed transaction. This ensures that for non-overlapping transactions x, y, if y finishes before x starts, then x serializes after y. Moreover, the transaction x of process p can commit only if max_ser_point > min_ser_point at the time of commit. Also,

Algorithm 2 $\mathit{helpCommit}(\texttt{lcn}, p)$

```
local_ser := ser_point($p$)
local_read_set := $rs(p)$
local_write_set := $ws(p)$
if $\langle$lcn, $p\rangle \neq \langle$commit_num, owner$\rangle$ then return
for each variable $v \in$ local_read_set
  local_read_version := $rv(v)$
  if local_read_version $<$ local_ser then
    $CAS(rv(v)$, local_read_version, local_ser$)$
for each variable $v \in$ local_write_set
  local_write_version := $wv(v)$
  local_read_version := $rv(v)$
  old_val := $Global(v)$
  new_val := $Local(p, v)$
  if $\langle$lcn, $p\rangle \neq \langle$commit_num, owner$\rangle$ then return
  if local_read_version $<$ local_ser then
    $CAS(rv(v)$, local_read_version, local_ser$)$
  if local_write_version $<$ local_ser then
    $CAS(wv(v)$, local_write_version, local_ser$)$
  $CAS(Global(v)$, old_val, new_val$)$
for all processes $p' \neq p$ do
  for each variable $v \in rs(p') \cap ws(p)$ do
    local_max := max_ser_point($p'$)
    if local_ser $<$ local_max then
      $CAS($max_ser_point($p'$), local_max, local_ser$)$
  for each variable $v \in ws(p') \cap ws(p)$ do
    local_min := min_ser_point($p'$)
    if local_ser $>$ local_min then
    $CAS($min_ser_point($p'$), local_min, local_ser$)$
$CAS(\langle$commit_num, owner$\rangle, \langle$lcn, $p\rangle, \langle$lcn, $\perp\rangle)$
```

for every variable $v \in rs(p)$, when v is read by transaction x, the write version $wv(v) <$ `min_ser_point`. Moreover, after v is read, no transaction that writes to v commits with a serialization point less than `max_ser_point`. Similarly, the variables in $ws(p)$ have not been written later than `min_ser_point`. Thus, the transaction x sees a state of the variables, consistent in the interval `min_ser_point` to `max_ser_point`. As this holds for every committing, aborting, and unfinished transaction of every process, opacity is guaranteed by OP-AVSTM.

□

Theorem 2. *OP-AVSTM is probabilistically permissive with respect to opacity.*

Proof sketch. For any opaque history h, every transaction serializes at some point within its lifetime. We can thus mark the serialization point of every transaction in h. As the length of h is finite, there is positive probability that every transaction in h chooses the required serialization point in OP-AVSTM. Hence, h is accepted by OP-AVSTM with positive probability.

□

Example of probabilistic permissiveness of OP-AVSTM. Consider an opaque history $h = ((\mathsf{write},\ v_1),\ p_1),\ ((\mathsf{write},\ v_2),\ p_2),\ ((\mathsf{read},\ v_1),\ p_2)\ (\mathsf{commit},\ p_1),\ (\mathsf{commit},\ p_2)$. First, the process p_1 writes to v_1. Then, the process p_2 writes to v_2. Then p_2 reads the variable v_1. When p_1 commits, let the chosen serialization point be c. The write version of v_1 is seen as the initial value 0 by p_2 as p_2 reads before p_1 updates the write version number of v_1 to c. In this case, it is ensured that the maximum serialization point of p_2 is also set to c. Now, when p_2 commits, it can choose a serialization point less than c and successfully commit. Even if the commands in Σ are not considered to be atomic, a similar argument can be made about the probabilistic permissiveness of AVSTM.

3.4 Liveness of AVSTM

Different notions of liveness have been proposed for transactional memories. A TM is *obstruction free* if for every infinite history h produced by the TM, if some process $p \in P$ takes infinitely many steps in isolation in h, then p commits infinitely often. A TM is *lock free* if every infinite history h produced by the TM contains infinitely many commits. A TM is *wait free* if every infinite history h produced by the TM contains infinitely many commits for every process $p \in P$.

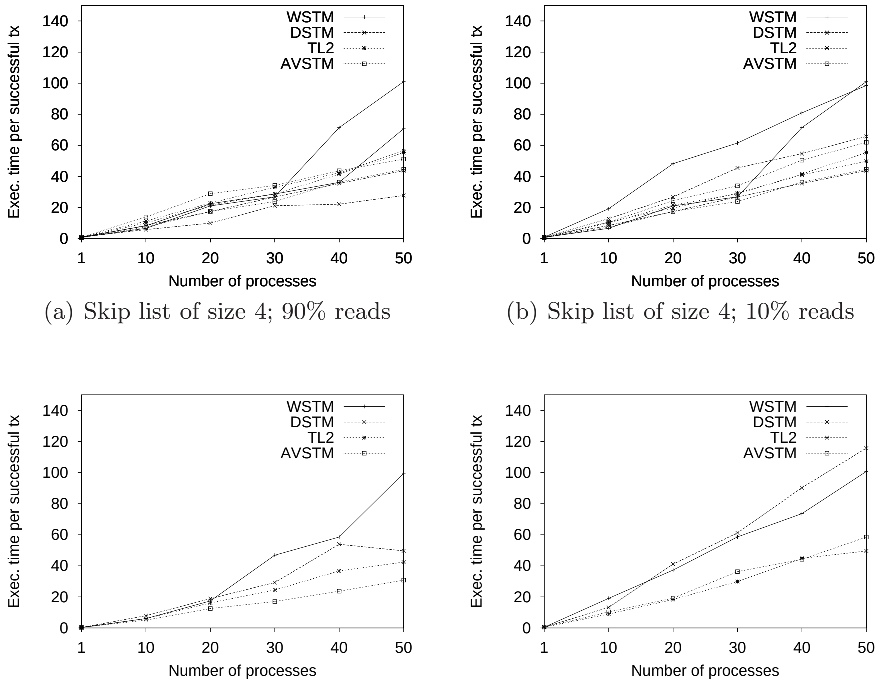
Theorem 3. *The algorithms SS-AVSTM and OP-AVSTM are lock free.*

Proof. We prove the theorem by contradiction. Let there exist a time t after which no process commits a transaction. We first note that no operation in AVSTM is blocking. Thus, every process, when scheduled, executes a statement of the algorithm. Also, transactions are of finite size. We note that a process can loop in the read or commit operation only if some other process performs a successful commit. So, if there is no commit of any transaction after time t, then there must be an infinite number of aborts of transactions after t. A transaction aborts only if it cannot find an interval to commit. As the maximum serialization point is initially ∞ when a transaction starts, the only way that a transaction does not find an interval to commit is when the maximum serialization point is set to a finite value. This occurs only within the procedure *helpCommit*. We also note that the maximum serialization point is changed in the procedure *helpCommit* at most k times, where k is the number of variables. Moreover, the procedure *helpCommit* for a particular transaction can be executed at most once by every process. Thus, there exists a time $t' \geq t$ such that the maximum serialization point of any process does not change after t'. Thus, there is no abort after t'. This contradicts our assumption. Hence, AVSTM guarantees lock freedom.

□

4 Implementation and experiments

To evaluate the practical importance of the notion of permissiveness, we implemented AVSTM within the LibLTX package [3]. The LibLTX package includes



(a) Skip list of size 4; 90% reads

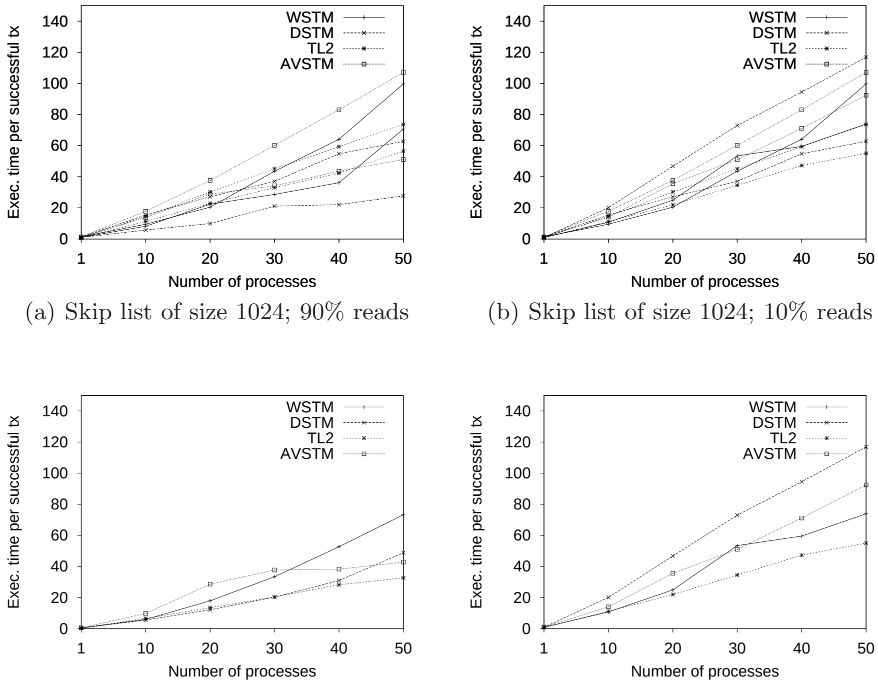
(b) Skip list of size 4; 10% reads

(c) Red-black tree of size 4; 90% reads

(d) Red-black tree of size 4; 10% reads

Fig. 1. Performance results on benchmarks in high contention. The execution time is measured in micro-seconds.

an implementation of DSTM with the Polka contention manager (which typically gives best performance results to DSTM [12]) and WSTM [3]. We also implemented an STM based on TL2 [1] within the LibLTX package. We integrated the storage management of AVSTM with the epoch-based garbage collector [2], where a memory pointer is not freed if any process holds a reference to it. This allows the simple use of CAS in the procedure *helpCommit*, without causing any ABA problem. We compare the performance of AVSTM configured for opacity, with DSTM, WSTM, and TL2 in high contention scenarios, that is, when many processes are accessing a small set of shared variables. We experimented on a quad dual core (8 processors) 2.8 GHz server with 16 GB RAM. Executing a large number of processes on a small number of processors creates a practical scenario, where a process holding a lock may get descheduled, and the liveness property of lock freedom becomes critical for performance. We use two different benchmarks: skip lists and red black trees. For both of these benchmarks, we experiment with a data size of 4 items, and with two different types of workloads, one is read dominated with 90% reads, and other is write dominated with 10% reads. The results (Figure 1) show that AVSTM always outperforms DSTM and WSTM



(a) Skip list of size 1024; 90% reads

(b) Skip list of size 1024; 10% reads

(c) Red-black tree of size 1024; 90% reads

(d) Red-black tree of size 1024; 10% reads

Fig. 2. Performance results on benchmarks in low contention. The execution time is measured in micro-seconds.

by upto 40% in high contention scenarios. Also, in our experiments, AVSTM outperforms TL2 in high contention when the workload is read dominated. We admit that the official implementation of TL2 will perform better than our version of TL2, but our experiments do show that AVSTM is comparable to the existing TM algorithms in the literature. We also note that AVSTM is lock free while TL2 is not. TL2 performs better than AVSTM in high contention when the workload is write dominated. This, we believe, has an interesting theoretical explanation. When the workload is write dominated, even AVSTM has to abort very often in order to be safe. This gives TL2 an advantage over AVSTM as TL2 uses a simpler invalidation scheme. On the other hand, when the workload is read dominated, AVSTM has to abort less often than TL2. This is because TL2 aborts many opaque histories, whereas the bookkeeping of AVSTM helps it to avoid some redundant aborts.

Permissiveness in AVSTM does come with a price. As we saw in the algorithm for AVSTM, different processes access the global data concurrently. Thus, AVSTM incurs high overhead due to poor cache performance. To evaluate this overhead, we evaluate the performance of different STMs in a low contention scenario of 1024 data items. The results are shown in Figure 2. We find that although AVSTM is not as good as other TMs, it does not yet pay an overwhelming penalty for the bookkeeping needed to achieve permissiveness. On average, AVSTM performs 20-30% worse than existing TM algorithms. We now discuss how AVSTM can be used as a fallback mechanism with TL2 to boost performance and progress guarantees in high contention scenarios.

Combining AVSTM with TL2

Our experiments gave us a clue on how we can make the best use of AVSTM, which has excellent performance in high contention scenarios. AVSTM also gives the progress guarantee of lock freedom. On the other hand, AVSTM is not a good performer in low contention scenarios. We propose to use AVSTM as a fallback mechanism, that is, it be combined with other STMs to get good performance and guaranteed progress when high contention brings both performance and progress at risk. We discuss here, as an example, how AVSTM can be combined with TL2.

A TM that is running in low contention uses the TL2 mode. A process that faces a series of aborts changes the mode from TL2 to AVSTM. Now, other processes which are not in the final phase of committing (once the locks have been acquired and the read set has been validated) can safely change their mode to AVSTM by observing the change of mode, for example, during the validation phase in the commit. A process which is in the final phase of commit has to be dealt with properly. When a process p_1 running in AVSTM mode tries to access a variable which is locked in TL2 mode by process p_2, following cases may occur:

- If the process p_1 observes that the process p_2 is not yet in the final phase of commit (by reading a flag for example), then p_1 can safely assume that p_2 will not write to the variable.
- If the process p_1 observes that p_2 is in the final phase of commit, then p_1 helps p_2 to commit. For this to work properly, processes running in TL2 mode should commit using a compare and swap (CAS). As the write set is generally small, this introduces negligible overhead.

5 Concluding Remarks

We presented a notion of permissiveness in TMs. As liveness guarantees are hard to provide in TMs, we believe that permissiveness can be an interesting, complementary metric while evaluating TMs theoretically. We discussed the high performance cost of a deterministic permissive STM due to the overhead of bookkeeping. We presented a randomized STM, AVSTM, that is probabilistically permissive for strict serializability and opacity. The randomization allows probabilistic decisions, and hence lowers the cost. We showed the practical importance of permissiveness by experiments that demonstrate how AVSTM outperforms existing STMs in high-contention scenarios. We also provided a strategy to use the randomized permissive STM in combination with TL2 to boost performance and progress guarantees.

Future work. We look ahead to prove a lower bound on the time complexity of an opacity-permissive deterministic TM, supporting the intuition that a practical deterministic TM cannot be opacity-permissive. We also plan to extend the formalism of permissiveness to *quantify* the amount of permissiveness of a TM. For example, a TM is k-permissive (where $0 \leq k \leq 1$) with respect to opacity if on every opaque history, the ratio of unnecessarily aborting transactions to the total number of transactions is at most k. This allows us to compare even non-permissive TMs by their degree of permissiveness. Also, as in other formalizations of STMs [4, 5, 13], we assumed the atomicity of individual commands (read, write, commit). Generally, the commit is not atomic and it would be interesting to revisit the notion of permissiveness with a finer grained model in mind.

References

1. David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *DISC*, pages 194–208. Springer, 2006.
2. Keir Fraser. *Practical Lock Freedom.* PhD thesis, Computer Laboratory, University of Cambridge, 2003.
3. Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 2007.
4. Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *PLDI*, pages 372–382. ACM Press, 2008.
5. Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Nondeterminism and completeness in transactional memories. In *CONCUR*. Springer, 2008.
6. Rachid Guerraoui, Maurice Herlihy, Michał Kapałka, and Bastian Pochon. Robust contention management in software transactional memory. In *SCOOL*, October 2005.
7. Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP*. ACM Press, 2008.
8. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.
9. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101. ACM Press, 2003.
10. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, pages 631–653, 1979.
11. Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, pages 284–298. Springer, 2006.
12. William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248. ACM Press, 2005.
13. Michael L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN WTC*, 2006.