1 Introduction

Transactional memory (TM) \cite{11, 15} allows concurrent processes to execute operations on data items within atomic blocks of instructions, called transactions. A transaction may either commit in which case all its updates become visible to other transactions, or abort in which case all its updates are discarded. The paradigm is appealing for its simplicity but implementing it efficiently is challenging. Ideally, the TM system should not introduce any contention between transactions beyond that inherently due to the actual code of the transactions. In other words, if two transactions access disjoint sets of data items (i.e. if they do not conflict), then none of these transactions should delay the other one, i.e., these transactions should not contend on any base object. This requirement has been called strict disjoint-access-parallelism. Base objects are low-level objects, which provide atomic primitives like read/write, load linked/store conditional, compare-and-swap, used to implement the TM system. Two transactions contend on some base object if both access that object during their executions and one of them performs a non-trivial operation on that object, i.e. an operation which may update its state.

Disjoint-access-parallelism was introduced in \cite{12}. Later variants \cite{1, 2, 6} employed the concept of a conflict graph. A conflict graph is a graph whose vertices represent transactions (or operations) performed in an execution $\alpha$ and an edge exists between two nodes if the corresponding transactions (operations) conflict in $\alpha$. In most of these definitions, disjoint-access-parallelism requires any two transactions to contend on a base object only if there is a path in the conflict graph of the minimal execution interval that contains both transactions. Different variants are met in the literature with the names disjoint-access-parallelism or weak disjoint-access-parallelism (most of them use different properties to restrict access to base objects). Stronger versions of disjoint-access-parallelism like strict disjoint-access-parallelism, result in more parallelism (and promote scalability) and therefore they are highly desirable when designing TM implementations: strict disjoint-access-parallelism is indeed ensured by blocking TM algorithms like TL \cite{4}. Nevertheless, a transaction that locks a data item and gets paged out might block all other transactions for a long amount of time. One might require a liveliness property that prevents such blocking. It was shown however in \cite{8} that a TM cannot ensure strict disjoint-access-parallelism if it also needs to ensure serializability \cite{13} and obstruction-freedom \cite{7, 5}. Obstruction-freedom ensures that a transaction can be aborted only when step contention is encountered during the course of its execution. Obstruction-freedom is weaker than lock-freedom or wait-freedom. It allows for designing simpler TM algorithms and therefore it has been given special attention in TM computing \cite{9}.

We study the following question: can we ensure strict disjoint-access-parallelism and obstruction

\*This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project.
freedom if we consider other consistency conditions? In other words, is consistency a major factor against scalability? We focus on snapshot isolation \[3\], a safety property which, roughly speaking, requires that transactions should be executed as if every read operation reads from some snapshot of the memory that was taken when the transaction started. Snapshot isolation is an appealing property for TM computing since it provides the potential to increase throughput for workloads with long transactions \[14\]. If the set of transactions is restricted to those that do not read data items they have previously written, snapshot isolation is a weaker property than strict serializability.

We prove that the answer is still negative. Namely, it is impossible to implement a TM which is strict disjoint-access-parallel and satisfies obstruction-freedom and snapshot isolation. To make our impossibility result stronger, we consider, for its proof, a weak snapshot isolation property which requires only that each transaction reads from some consistent snapshot of the memory taken when it starts, thus ignoring the extra constraint (met in the literature \[6, 14\] for snapshot isolation) that from two concurrent transactions writing to the same data item, only one must commit. It is remarkable that our impossibility result holds for two variants of snapshot isolation, the original one from the database world where a read of a data item following a write to this data item by the same transaction returns the value written by this write, and a simpler one where all reads (even those following writes to the same data item by the same transaction) return the value recorded in the snapshot taken at their start.

We also show how to circumvent the impossibility result if we relax the disjoint-access-parallelism requirement between update transactions: mainly we show how we can get a simplified version of DSTM \[5\], called SI-DSTM, which satisfies snapshot isolation, obstruction-freedom, and the following weaker disjoint-access-parallelism requirement: two operations (executed by two concurrent transactions \(T_1\) and \(T_2\)) on different data items, one of which is a read operation, never contend on the same base object, and two write operations (by \(T_1\) and \(T_2\)) on different data items contend on the same base object only if there is a chain of transactions starting with the transaction that performs one of these write operations and ending with the transaction that performs the other, such that every two consecutive transactions in the chain conflict and the execution interval of each of these transactions overlaps with that of either \(T_1\) or \(T_2\). We call the property satisfied by write transactions weak disjoint-access-parallelism. DSTM satisfies weak disjoint-access-parallelism for all transactions, while SI-DSTM satisfies strict disjoint-access-parallelism for read-only transactions and weak disjoint-access-parallelism for write transactions. Moreover, SI-DSTM is significantly simpler than the original DSTM algorithm.

2 Useful Definitions

A TM algorithm provides implementations for the routines \texttt{readDI} and \texttt{writeDI} which are called to read or write data items, respectively, and for the routines \texttt{CommitTr} and \texttt{AbortTr}, which are called when a transaction tries to commit or abort, respectively. Each time a transaction calls one of these routines we say that it invokes an operation; when the execution of the routine completes, a response is returned. We denote the invocation of \texttt{CommitTr} by a transaction \(T\) as \texttt{commit} \(T\); the response to \texttt{commit} \(T\) can be either \(C_T\) (commit) or \(A_T\) (abort). We denote by (i) \(x.\texttt{write}(v)\) the invocation of \texttt{writeDI} for data item \(x\) with value \(v\); it returns \texttt{ok} if the write was successful or \(A_T\) if the transaction that invoked it has to abort, (ii) \(x.\texttt{read}()\) the invocation of \texttt{readDI} for data item \(x\); it returns a value for \(x\) if the operation was successful or \(A_T\) if the transaction that invoked it has to abort.

A configuration consists of the state of each process and the state of each base object. A step of a process consists of applying a single operation on some base object, its response, and zero or more local operations that may cause the internal state of the process to change; each step is executed atomically. An execution \(\alpha\) is a sequence of steps. An execution is solo if every step is performed by the same process. Two executions \(\alpha_1\) and \(\alpha_2\) starting from configurations \(C_1\) and \(C_2\), respectively, are indistinguishable to transaction \(T\), if the state of \(T\) is the same in \(C_1\) and \(C_2\), and the sequence of
steps performed by $T$ (and thus also the responses it receives) are the same during both executions.

A history $H$ is a sequence of invocations and responses performed by transactions. Given an execution $\alpha$, we denote by $H_\alpha$ the sequence of invocations and responses performed by the transactions in $\alpha$. Let $H|T$ be the longest subsequence of $H$ consisting only of invocations and responses of a transaction $T$. We say that $T$ commits (aborts) in $H$ if $H|T$ ends with $C_T(A_T)$. If $H|T$ ends with a commit invocation, then $T$ is commit-pending. A history $H$ is sequential if no two transactions are concurrent in $H$. $H$ is complete if it does not contain any transactions that have neither committed nor aborted. A read operation $x.read()$ by some transaction $T$ is global if $T$ has not invoked $x.write(*)$ before invoking $x.read()$. Transaction $T$ is legal in a sequential history $H$, if every read operation $r$ whose response is not $A_T$, returns a value $v$ such that: (i) if $r$ is not global, then $v$ is the value written by the last invocation of $x.write$ preceding $r$ in $H$; (ii) if $r$ is global and there are committed transactions preceding $T$ in $H$ which invoke $x.write(*)$, then $v$ is the argument of the last $x.write()$ invocation by those transactions; (iii) otherwise, $v$ is the initial value of $x$. A complete sequential history $H$ is legal if every transaction is legal in $H$.

Let $T$ be a committed or commit-pending transaction in a history $H$. Let $T|read$ be the longest subsequence of $H|T$ consisting only of the global read invocations and their corresponding responses and $T|other$ be the subsequence $H|T - T|read$, i.e. $T|other$ consists of all invocations performed by $T$ (and their responses) other than those comprising $T|read$. Let $\lambda$ be the empty execution. Then we define $T_r$ and $T_o$ in the following way:

- $T_r = T|read \cdot commit_{T_r} \cdot C_{T_r}$ if $T|read \neq \lambda$, and $T_r = \lambda$ otherwise, and
- $T_o = T|other \cdot commit_{T_o} \cdot C_{T_o}$ if $T|other \neq \lambda$, and $T_o = \lambda$ otherwise.

**Definition 2.1.** An execution $\alpha$ satisfies snapshot isolation, if for every committed transaction $T$ (and for some of the commit-pending transactions) in $\alpha$ it is possible to insert a read serialization point $*_{T,r}$ and a write serialization point $*_{T,w}$ such that: (i) $*_{T,r}$ precedes $*_{T,w}$, (ii) both $*_{T,r}$ and $*_{T,w}$ are inserted within the execution interval of $T$, and (iii) if $\sigma_\alpha$ is the sequence defined by these serialization points, in order, and $H_{\sigma_\alpha}$ is the history we get by replacing each $*_{T,r}$ with $T_r$ and each $*_{T,w}$ with $T_o$ in $\sigma_\alpha$, then $H_{\sigma_\alpha}$ is legal.

### 3 The impossibility result

**Theorem 3.1.** No obstruction-free STM can ensure both snapshot isolation and strict disjoint-access-parallelism.

**Proof.** We provide a brief description of the proof. Assume, by contradiction, that there is an obstruction-free STM which ensures both snapshot isolation and strict disjoint-access-parallelism. We consider the following transactions, all executed by distinct processes: (1) $T_1$, executed by process $p_1$, writes value 1 to $a$, $b$, and $c$, (2) $T_2$, executed by process $p_2$, writes value 2 to $a$, $d$, and $e$, (3) $T_3$, executed by process $p_3$, reads from $b$, (4) $T_4$, executed by process $p_4$, reads from $d$, (5) $T_5$, executed by process $p_5$, reads from $e$, (6) $T'_5$, executed by process $p'_5$, reads from $c$, and (7) $T_6$, executed by process $p_6$, reads from $a$.

We will construct two executions: $\delta = \alpha_1 \cdot \alpha_2 \cdot s_1 \cdot s_2 \cdot \gamma_6$, and $\delta' = \alpha_1 \cdot \alpha_2 \cdot s_2 \cdot s_1 \cdot \gamma'_6$, such that: 
(i) $\gamma_6$ and $\gamma'_6$ are solo executions of $T_6$, (ii) in $\gamma_6$, $T_6$ reads the value 2 for $a$, (iii) in $\gamma'_6$, $T_6$ reads the value 1 for $a$, and (iv) $\gamma_6$ and $\gamma'_6$ are indistinguishable. Thus, $T_6$ should read the same value for $a$ in both executions which is a contradiction.

Specifically, $\alpha_1$ is the solo execution of $T_1$ by process $p_1$ from the initial configuration $C_0$ up until the point that $p_1$ is poised to execute step $s_1$ which is critical for $T_3$, i.e. in the execution $\gamma_3$ where $p_3$ executes solo $T_3$ before $s_1$ until $T_3$ commits ($T_3$ will indeed commit because of snapshot isolation) $T_3$ reads 0 for $b$, whereas in the execution $\gamma_{3'}$ where $p_3$ executes solo $T_3$ after $s_1$ until $T_3$ commits, $T_3$ reads 1 for $b$. To define $\alpha_2$ and $s_2$, we let $p_2$ execute solo $T_2$, after $\alpha_1$, until it is poised to execute...
a step which is critical for $T_4$, i.e. in the execution $\gamma'_4$ where $p_4$ executes solo $T_4$ before $s_2$ until $T_4$ commits, $T_4$ reads 0 for $d$, whereas in the execution $\gamma_4$ where $p_4$ executes solo $T_4$ after $s_2$ until $T_4$ commits, $T_4$ reads 2 for $d$.

We argue that (i) $s_1$ applies a non-trivial operation on some base object $o_1$ such that $T_3$ reads $o_1$ in $\gamma_3$, (ii) $\gamma_3$ is legal after $o_1 \cdot o_2 \cdot s_1$ and $\gamma'_3$ is legal after $o_1 \cdot o_2 \cdot s_2$, and (iii) $\gamma_3$ and $\gamma'_3$ do not read any base object written by $o_2 \cdot s_2$. Similarly, we argue that (i) $s_2$ applies a non-trivial operation on some base object $o_2 \neq o_1$ such that $T_4$ reads $o_2$ in $\gamma_4$, (ii) $\gamma_4$ and $\gamma'_4$ are legal after $o_1 \cdot o_2 \cdot s_1$, and (iii) $\gamma_4$ and $\gamma'_4$ do not read any base object written by $o_1 \cdot s_1$.

To argue that $\gamma_6$ and $\gamma'_6$ will return the required values, we consider the following two executions: $\gamma = o_1 \cdot o_2 \cdot s_1 \cdot \gamma_3 \cdot \gamma'_3 \cdot s_2 \cdot \gamma_5 \cdot \gamma_6$ (Fig. 1) and $\gamma' = o_1 \cdot o_2 \cdot s_2 \cdot \gamma_4 \cdot \gamma'_3 \cdot s_1 \cdot \gamma'_5 \cdot \gamma'_6$, where $\gamma_5$ is the solo execution of $T_5$ by $p_5$ until $T_5$ commits, and $\gamma'_5$ is the solo execution of $T'_5$ by $p'_5$ until $T'_5$ commits. We argue that these executions are legal. To conclude the proof, we prove that $\gamma$ and $\delta$ are indistinguishable to $p_6$ and $T_6$ reads 2 for $a$ in $\gamma$; similarly, $\gamma'$ and $\delta'$ are indistinguishable to $p_6$ and $T_6$ reads 1 for $a$ in $\gamma'$.

We remark that the impossibility result also holds for a simpler version of Definition 2.1 where $T|\text{read}$ contains all read operations (global or not) performed by $T$ and $T|\text{other}$ contains all its write operations. moreover, the impossibility result holds if we consider a stronger version of Definition 2.1 which ensures the stated property for every prefix $\alpha'$ of $\text{alpha}$. Finally, we have shown that the impossibility result holds even if the system provides primitives that atomically access more than one base objects.

```
\begin{figure}
\centering
\begin{tikzpicture}[node distance=2cm, auto,>=latex]
  \node (p1) {$p_1$};
  \node (T1) [right of=p1] {$T_1$};
  \node (s1) [right of=T1] {$s_1$};
  \node (p2) [below of=p1] {$p_2$};
  \node (T2) [right of=p2] {$T_2$};
  \node (s2) [right of=T2] {$s_2$};
  \node (p3) [below of=p2] {$p_3$};
  \node (T3) [right of=p3] {$T_3$};
  \node (C) [right of=T3] {$C$};
  \node (p4) [below of=p3] {$p_4$};
  \node (T4) [right of=p4] {$T_4$};
  \node (C) [right of=T4] {$C$};
  \node (p5) [below of=p4] {$p_5$};
  \node (T5) [right of=p5] {$T_5$};
  \node (C) [right of=T5] {$C$};
  \node (p6) [below of=p5] {$p_6$};
  \node (T6) [right of=p6] {$T_6$};
  \node (C) [right of=T6] {$C$};

  \draw[->] (p1) -- (T1);
  \draw[->] (T1) -- (s1);
  \draw[->] (p2) -- (T2);
  \draw[->] (T2) -- (s2);
  \draw[->] (p3) -- (T3);
  \draw[->] (T3) -- (C);
  \draw[->] (p4) -- (T4);
  \draw[->] (T4) -- (C);
  \draw[->] (p5) -- (T5);
  \draw[->] (T5) -- (C);
  \draw[->] (p6) -- (T6);
  \draw[->] (T6) -- (C);

  \node at (0.5,0) {$\alpha_1$};
  \node at (1.5,0) {$\alpha_2$};
  \node at (2.5,-0.5) {$s_1$};
  \node at (3.5,-0.5) {$\gamma_3$};
  \node at (4.5,-0.5) {$\gamma'_3$};
  \node at (5.5,-0.5) {$s_2$};
  \node at (6.5,-0.5) {$\gamma_5$};
  \node at (7.5,-0.5) {$\gamma_6$};
\end{tikzpicture}
\caption{Execution $\gamma$ in which $T_6$ reads 2 from $a$.}
\end{figure}
```

4 SI-DSTM

SI-DSTM is a simplified version of DSTM [11]. It maintains, for each update transaction $T$, a record with fields: (i) Status: stores the current status of $T$ (takes values Active, Committed, or Aborted, initially Active), (ii) pendingStatus: records whether $T$ should eventually abort (takes values Active, Committed, or Aborted, initially Committed), and (iii) readList: stores information about the data items read by $T$. As in DSTM, SI-DSTM maintains two records, Locator and TMObject, for each data item $x$ and ensures a one-to-one correspondence between data items and TMObjects. It $T$ wants to write a data item and the ownership of $x$ is already held by $T$, then the new value is written in the newObject field of current $x$’s locator. Otherwise, as in DSTM, cloning and indirection are employed: a new locator is created for $x$ and its transaction field is initialized.
to point to the transactional record of $T$. Then, $T$ repeatedly tries to change the start field of the TMObject of $x$ to point to this new locator. Before doing so, it writes the value Aborted in the pendingStatus field of the transactional record of the transaction that $T$ found to be the holder of the ownership of $x$. Each transaction validates its read set to ensure that each data item in $T$’s read list is still consistent. We remark that performing this validation only once at commit time is enough to ensure snapshot isolation.

In SI-DSTM, read-only transactions are invisible. Specifically, SI-DSTM does not maintain shared transactional records for read-only transactions. Each such transaction $T$ maintains only a read list in its private memory space. Moreover, read-only transactions never cause other transactions to abort. From the transactional records of update transactions, a read-only transaction reads only the status field in order to find the current value of a data item that it wants to read and it is owned by any of these transactions; it does so in a way similar to that of DSTM. In order strict disjoint-access-parallelism to be ensured between a read-only transaction $T$ and the update transactions, in SI-DSTM, Locator contains an additional field, called pendingStatus, in each transactional record. If a transaction $T_1$ performs a write operation to a data item for which transaction $T_2$ holds the ownership, $T_1$ does not write the value Abort in the status field of $T_2$; it rather writes this value in pendingStatus to indicate that $T_2$ should eventually abort. At commit time, $T_2$ performs an exchange of pendingStatus and status. It then reads status again and returns true or false depending on the value of status. In this way, read-only transactions that read data items owned by $T_2$ contend only with $T_2$ and not with $T_1$ or other update transactions that write in the transactional record of $T_2$. Thus, strict disjoint access parallelism is ensured between a read-only transaction and any other transaction.

References


