Abort Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions

(General Track)

Shlomi Dolev
Ben-Gurion University of the Negev
dolev@cs.dgu.ac.il

Panagiota Fatourou
University of Crete & FORTH-ICS
faturu@csd.uoc.gr

Eleftherios Kosmas
University of Crete & FORTH-ICS
ekosmas@csd.uoc.gr

Abstract

We present a `TM` system that executes transactions without ever causing any aborts. The system uses a set of *t-var lists*, one for each transactional variable. A scheduler undertakes the task of placing the instructions of each transaction in the appropriate t-var lists based on which t-variable each of them accesses. A set of worker threads are responsible to execute these instructions. Because of the way instructions are inserted in and removed from the lists, by the way the worker threads work, and by the fact that the scheduler places all the instructions of a transaction in the appropriate t-var lists before doing so for the instructions of any subsequent transaction, it follows that no conflict will ever occur. Parallelism is fine-grained since it is achieved at the level of transactional instructions instead of transactions themselves (i.e., the instructions of a transaction may be executed concurrently).

1. Introduction

In *asynchronous shared memory* systems, where processes execute in arbitrary speeds and communication between them occurs through accessing basic shared *primitives* (usually provided by the hardware), having processes executing pieces of code in parallel is not an easy task due to synchronization conflicts that may occur among processes that need to concurrently access non-disjoint sets of shared data. A promising parallel programming paradigm is the Transactional Memory (`TM`) approach where pieces of code that may access data that becomes shared in a concurrent environment (such pieces of data are called *transactional variables* or *t-variables*) are indicated as *transactions*. A `TM` system ensures that the execution of a transaction T will either *succeed*, in which case T *commits* and all its updates become visible, or it will be *unsuccessful*, in which case T *aborts* and its updates are discarded. Each committed transaction appears as if it has been executed "instantaneously" in some point of its execution interval.

When a conflict between two transactions occurs, `TM` systems usually abort one of the transactions to ensure consistency; two transactions *conflict* if they both access the same t-variable and one of these accesses is a `write`. To guarantee *progress*, all transactions should eventually commit. This property, albeit highly desirable, is scarcely ensured by the currently available `TM` systems; most of these systems do not even ensure that transactions abort only when they violate the considered consistency condition (this property is known as permissiveness [13]). The work performed by a transaction that aborts is discarded and the transaction is later restarted; this incurs a performance penalty. So, the nature of `TM` is optimistic; if transactions rarely abort then no work is ever discarded. In terms of achieving good performance, the system should additionally guarantee that parallelism is achieved. So, transactions should not be executed sequentially and global contention points should be avoided. `TM` algorithms that never abort transactions are invaluable since they additionally support irrevocable transactions.

In this paper, we present SemanticTM, an opaque [9] `TM` algorithm which achieves (1) the strongest progress guarantee by ensuring that transactions never abort, and (2) fine-grain parallelism at the transactional instruction level: in addition to instructions of different transactions, instructions of the same transaction that do not depend on each other can be executed concurrently. So, SemanticTM ensures wait-freedom.

SemanticTM employs a list for each t-variable. A scheduler places the instructions of each transaction in the appropriate lists in FIFO order. Specifically, an instruction is placed in the list of the t-variable that it accesses. A set of worker threads "consume" instructions from the lists, in order, and execute them. The algorithm is highly fault-tolerant. Even if some worker threads fail by crashing, all transactions whose instructions have been placed in the lists will be executed. We remark that for relatively simple transactions that access a known set of t-variables, and their codes contain `read` and `write` instructions on them, conditionals (i.e. `if`, `else if`, and `else`), loops (i.e. `for`, `while`, etc.), and function calls, the work of the scheduler can be done at compile time (so the scheduler component is worthless in this case). For simplicity of presentation, this is the case that we focus on in this version of the paper. We briefly discuss, in Section 3, how to extend SemanticTM to cope with more complicated transactional codes. A more elaborated discussion, as well as a correctness proof for SemanticTM, will be provided in future versions of the paper.

`TM` algorithms that never abort transactions has been recently presented in [1, 14]. These algorithms use ideas from [21] where a `TM` algorithm supporting one irrevocable transaction at each point in time is presented. In the algorithms in [1, 14], read-only transactions are *wait-free*, i.e. each of them is completed successfully within a finite number of steps; a *read-only* transaction never writes a t-variable in contrast to an *update* transaction that performs `writes` on such variables. However, these algorithms restrict parallelism by executing all update transactions sequentially using a global lock. On the contrary, SemanticTM guarantees that no transaction aborts while exploiting parallelism between both writers and readers. Moreover, SemanticTM does not use any locks and therefore update transactions are also executed in a wait-free way. TM algorithms that support wait-free read-only transactions are presented in [3, 17]. Update transactions in these algorithms may abort and they require locks to execute some of the transactional instructions.

To enhance progress in TM, a lot of research has been performed on designing efficient contention managers and transactional schedulers. A contention manager [11, 19, 20] is a TM component aiming at ensuring progress by providing efficient conflict resolution policies. When two transactions conflict, the contention manager is employed to decide whether simple techniques, like back-off, would be sufficient, or which of the transactions should abort or be paused to allow the other transaction to complete. SemanticTM prevents conflicts from occurring thus making the use of a contention manager unnecessary.

Somewhat closer to the work proposed here, a *transactional scheduler* is a more elaborated TM component which places transactions in a set of queues, usually one for each process; a set of working threads then removes and executes transactions from these queues. In addition to deciding which transaction to delay or abort when a conflict occurs, and when to restart a delayed or aborted transaction, a scheduler makes also decisions on the scheduling queue that the transaction will be placed once its execution will be resumed or restarted. Some of the proposed schedulers always abort one of the two transactions and place it in an appropriate queue to guarantee that the transaction will be restarted only after the conflicting transaction has finished its execution, i.e. they serialize the execution of the two transactions. CarSTM [7], Adaptive Transaction Scheduling [22], and Steal-on-Abort [2] are examples of such schedulers. Attiya and Milani [4] presented a scheduler which alternates between reading and writing epochs. In a reading epoch, priority is given to the execution of read-only transactions in contrast to what happens in a writing epoch. This technique behaves better for read-dominated [12] and bimodal [4] workloads, for which schedulers like those presented in [2, 7, 22] may serialize more than necessary. However, the working threads in the algorithm of [4] use locks; additionally, aborts are not avoided. To evaluate a transactional scheduler, competitive analysis is often employed [4, 5, 10, 11] where the total time needed to complete a set of transactions (known as *makespan*) is compared to the makespan of a clairvoyant scheduler [15].

In [8], a transactional scheduler has been presented which schedules transactions based on future prediction of their data sets on the basis of a short history of past transactions and the accesses that they perform. If a transaction is predicted to conflict with an active transaction then it is serialized. To avoid serializing more than necessary in cases of low contention, a heuristic is used where prediction and serialization are employed only if the completion rate of transactions falls below a certain threshold.

In [18], a lock-based dependence-aware transactional memory system is presented which dynamically detects and resolves conflicts. Its implementation extends ideas from TL II [6] with support of dependence detection and data forwarding. The algorithm serializes transactions that conflict; in case of aborts, cascading aborts may occur. The current version of SemanticTM copes only with transactions that their data sets are known. However, SemanticTM ensures that all transactions will always commit within a bounded number of steps, thus providing stronger progress guarantees.

In [16], a database transaction processing system similar to SemanticTM is proposed; database transactions can be thought of as transactions whose data sets are known, in TM concept. Similarly to SemanticTM, consecutive transactional instructions of a transaction are separated into groups, called *actions*, according to the dataset (part of the database) they access, each worker thread is responsible to execute instructions for a disjoint set of datasets, and each action is scheduled to the appropriate thread. Data dependencies between actions are maintained using extra metadata. Specifically, a shared object (additional to database's tables), called *rendezvous point*, is maintained for the dependencies of each action of some transaction; a single action may have several data dependencies and each of those dependencies will be resolved by the corresponding thread. Using these rendezvous points the execution of a transaction is separated into phases, with each phase containing independent actions. A thread initiating the execution of a transaction, schedules the independent actions (of the first phase) to the appropriate worker threads. When a worker thread resolves the last dependency of some rendezvous point, it initiates the next phase of transaction's execution by scheduling the next independent actions of this transaction. However, due to its execution scheme a transaction executed in this system may have to abort, whereas in SemanticTM transactions never abort.

2. SemanticTM

Main Ideas. SemanticTM uses a set of lists, called *t-var lists*, one for each t-variable. A thread, called *scheduler*, places the instructions of each transaction in the appropriate t-var lists based on which t-variables each of them accesses. It also records any dependencies that may exist between the instructions of the same transaction. A set of *worker* threads reads and executes instructions from the t-var lists. We use compiler support to know, for each instruction, any dependency that may lead to or originate from it. The main structure of SemanticTM is illustrated in Figure 1.

In SemanticTM, all the transactional instructions of each transaction T are placed in the t-var lists before the transactional instructions of any other transaction. Each of the workers repeatedly chooses, uniformly at random, a t-var list and executes the transactional instructions of this list, starting from the first that is ready. Processing transactions in this way ensures that conflicts between transactions never occur; so, transactions never abort. As an example, consider the simple transactions T_1 and T_2, presented in Figures 2 and 3, respectively. Obviously, there are conflicts between these two transactions since they both read and write t-variables x and y. Without loss of generality, assume that the instructions of T_1 are placed in the t-var lists first. Then, the instructions of lines 1 and 2 of T_1 will be placed in the t-var list for x before the write to x on line 6 of T_2. Similarly, the write to y of line 3 of T_1 will be placed in the t-var list for y before the write to y of line 5 of T_2. Since the worker threads respect the order in which instructions have been inserted in the t-var lists when they execute them, the instructions of T_1 on every t-variable will be executed before the instructions of T_2 on this t-variable, and thus no conflict between T_1 and T_2 will occur. This explains why no transaction ever aborts in SemanticTM.

The set of t-variables accessed by a transaction is called the transaction's *data set*. Notice that an array can either be itself a t-variable, or each of its elements can be a t-variable. We call *control flow statements* the conditionals and loops, and we use the instruction `cond` to refer to such a statement. The term *transactional instructions* refers to `read`, `write`, and `cond` instructions. We call *block* the body of a control flow statement (i.e. the set of its instructions); so each `cond` instruction is associated with a block.

Dependencies. If the execution of a transactional instruction e_1 requires the result of the execution of another instruction e_2, then there is a *dependency* between e_1 and e_2. This dependency is an *input* dependency for e_1 and an *output* dependency for e_2. A dependency between a `read` and a `write` is called *data* dependency. We remark that SemanticTM will place five instructions for T_1 in the t-var lists: e_1 which is a `write` on x (line 1), a `read` e_2 and a

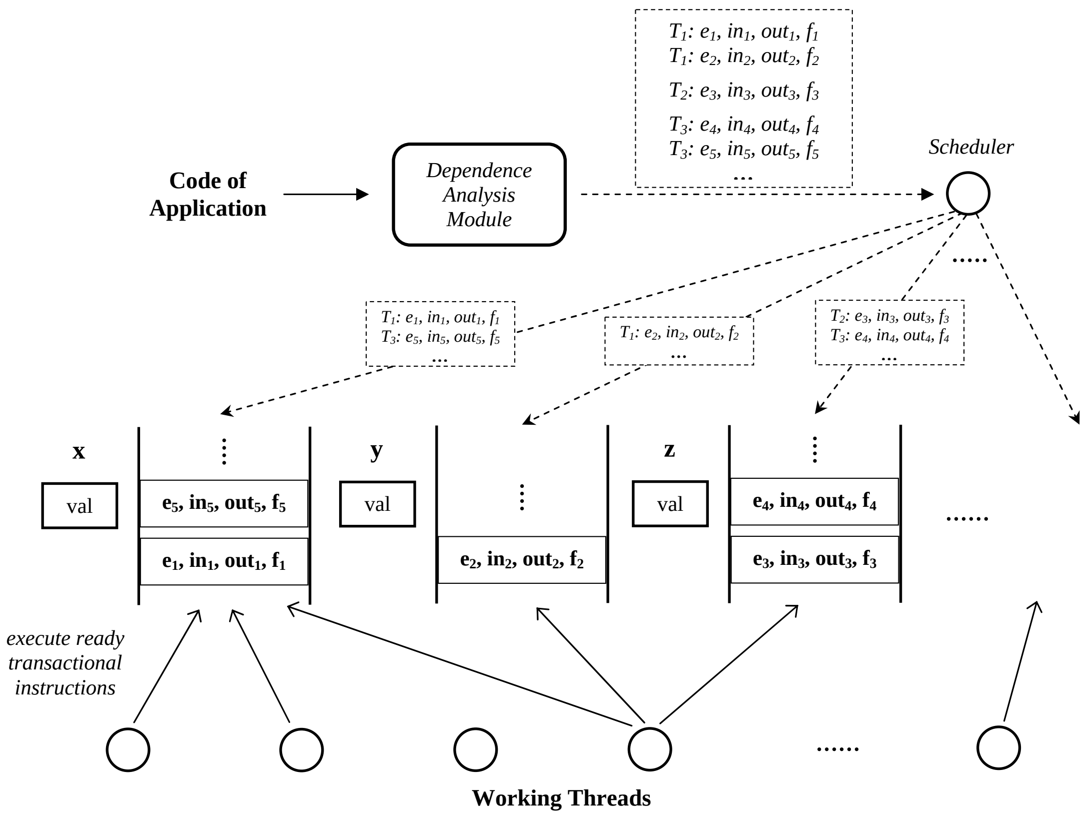

Figure 1. Main components of SemanticTM.

```
1   $x := 3$
2   $x++$
3   $y := x$
```

Figure 2. Transaction T_1.

```
4   $z := 2$
5   $y := z$
6   $x := y$
```

Figure 3. Transaction T_2.

```
7   $x := 1$
8   if (...) then
9       $x := 2$
10  else
11      $x := 4$
12  $y := x$
```

Figure 4. Transaction T_3.

```
13  $x := 1$
14  while ($x < 10$) then
15      $z := x$
16      $x := x * 2$
17  $y := x$
```

Figure 5. Transaction T_4.

`write` e_3 to x (line 2), a `read` e_4 on x and a `write` e_5 to y for line 3. Notice that there is an output dependency originating from e_1 to e_2 and one from e_3 to e_4. It is remarkable that SemanticTM does not maintain input dependencies for any `read` instruction e on a t-variable x, since all `writes` to x on which e depends have been placed in the t-var list of x before e and therefore the `read` can simply get the value from the matadata of x (by the way the algorithm works, this value will be consistent). Thus, SemanticTM records input dependencies only for `write` and `cond` instructions.

A dependency that either leads to or originates from a `cond` instruction is called *control* dependency. For each `cond` instruction, SemanticTM maintains an output control dependency from `cond` to each transactional instruction e of the block associated with it. As an example, there are two output control dependencies for instruction 8 (to 9 and 11). We assume that for each `write` instruction on a t-variable x, or for each `cond` instruction e, a function f can be applied to the values of the input dependencies of e in order either to calculate the new value of x or to evaluate whether the condition is `true` or `false`, respectively. We remark that f should be applied after all the input data dependencies of e have been resolved.

A brief description of all possible dependencies for each transactional instruction is provided in Table 1. The state of a transactional instruction is *waiting*, if at least one of its input dependencies has not been resolved, otherwise, it is *ready*; a transactional instruction is *active* if it is either waiting or ready.

By using compiler support, the dependencies between the instructions of a transaction are known before the beginning of its execution. Each instruction, together with its dependencies (and function), is placed in the appropriate t-var list, as a single *entry*. For example, Figure 1 illustrates the extraction of transactional instructions e_1 and e_2 from a transaction T_1, e_3 from some T_2, and e_4 and e_5 from a transaction T_3, with input dependencies $in_1, \ldots, in_5$, output dependencies $out_1, \ldots, out_5$, and functions $f_1, \ldots, f_5$, respectively. Also, the figure presents their placement into the t-var lists of x, y, and z.

Conditionals. Each part of a conditional (`if`, `else if`, `else`) is associated with a `cond` instruction and a block. Therefore, for an `if` ... `then` ... `else` statement, the two `cond` instructions (for the `if` and the `else` part) and their blocks' transactional instructions will be placed in the appropriate t-var lists. Then, at runtime, one of the two `cond` instructions will be evaluated to `false` so its block's instructions will be invalidated by the working thread that executes this `cond`. A `cond` instruction can be inserted in the t-var list of any

Transactional Instruction	Dependencies			
	Input		Output	
	Data Dep	Control Dep	Data Dep	Control Dep
$e = \texttt{read}(x)$	In SemanticTM, e has no input data dependencies	if e participates in some block, it has an input control dependency originating from the block's `cond`	e forwards the value it reads to `write` and `cond` instructions that depend on it	if e participates in some loop's block, an output control dependency originates from e to its block's `cond`
$e = \texttt{write}(x)$	e may have input data dependencies originating from `reads`	if e participates in some block, it has an input control dependency originating from the block's `cond`	In SemanticTM, e has no output data dependencies	if e participates in some loop's block, an output control dependency originates from e to its block's `cond`
$e = \texttt{cond}$	e may have input data dependencies originating from `reads`	if e is a `cond` of a loop `cond`, it has input control dependency originating from each of its block's instructions `cond`		e has output control dependencies to each of its block's instructions

Table 1. Data dependencies between transactional instructions.

t-variable; in the current version of SemanticTM it is placed in the t-var list of the first instruction of its block.

Notice that a transactional instruction of some block, may have *outside-block* dependencies which come from or lead to instructions that does not belong to the block. For instance, there may be outside-block dependencies from the instruction of line 7 to the `cond` instructions of the `if`...`then` ...`else` or to the instructions of the `cond`s' blocks. Notice that in SemanticTM outside block dependencies are resolved in a direct way because of the way that the transactional instructions are placed in the t-var lists. For example, to execute line 12, SemanticTM places a `read` e and a `write` e' in the t-var lists of x and y, respectively. Then later on, when e is executed, all previous `writes` to x have been performed, so the metadata of x contain a consistent value and e can read the value from there (so e does not have any input dependency). However, there is a dependency from e to e'.

Loops. Let e be a transactional instruction that is included in a loop block; let c be the associated `cond` instruction. SemanticTM places c and each instruction of the block in the appropriate t-var lists only once independently of the number of times that the loop will be executed since this number may be known only at run time.

We remark that the execution of e (and c) in some iteration may depend on the execution of some transactional instructions of the previous iteration; we call such a dependency *across-iteration*. As an example, consider transaction T_4, presented in Figure 5. Notice that, for iterations other than the first, the `cond` and the `read` of lines 14 and 15, respectively, depend on the value of x calculated on line 16 during the previous iteration.

In order to perform c multiple times, an *iteration counter* cnt_c is associated with c. This counter stores the current iteration number of the loop's execution. Moreover, the input control dependency of e is implemented with a counter cnt_e; similarly, the input control dependencies of c are implemented as counters as well. If $cnt_e = cnt_c$, then the input control dependency of e is resolved, otherwise not. Notice that cnt_e can be either equal to or smaller by one from c's iteration counter. This is so, since c can initiate a new iteration only after its input control dependencies originating from its block instructions have been resolved, i.e. after all these instructions have been executed for the current iteration; similarly, these block instructions can be executed only if their input control dependencies (from c) have been resolved.

To ensure correctness, an *iteration number* is associated with each of the input data dependencies of e (or c); this iteration number is stored together with the corresponding input dependency into a `CAS` object. When the iteration number of an input data dependency $inDep$ of e (or c) is smaller than the iteration counter of c, it follows that $inDep$ is unresolved for the current iteration; if all input data dependencies of e have their iteration number fields equal to the iteration counter of c, then all data dependencies of e have been resolved. If the input control dependency is also resolved, then e can be executed. Once e is executed, it resolves the control dependency to c by writing there an iteration number equal to the current iteration counter plus one. When all dependencies of c have been resolved the counter of c increases by one and c can be executed.

Consider a t-var list that contains two transactional instructions e_1 and e_2, in this order, which are included in the same loop block. Assume that there is an across iteration dependency from e_2 to e_1. We remark that the input across-iteration data dependency of e_1 originating from e_2 should be initialized as resolved in order for e_1 to appropriately become ready for the first iteration of the loop. Also, notice that the execution of e_1 during the second iteration should occur only after the execution of the first-iteration instance of e_2. Since e_1 precedes e_2 in the list, the working threads may have to search which element of the corresponding t-var list is ready (instead of just checking whether the first element of the list is ready). Specifically, t-var list should be searched until a transactional instruction is found that does not participate in the same loop (or until its end, if such an instruction does not exist). We remark that, the loop in which a transactional instruction participates can be determined using its output control dependency.

For simplicity, the code of SemanticTM, as presented in Figures 7 and 8, does not cope with nested conditionals and nested loops. We briefly discuss how to support these features in Section 3.

Worker threads. Since working threads choose the t-var list to work on uniformly at random, it may happen that several working threads may (concurrently) execute the same instruction. To synchronize workers that execute the same instructions, the following synchronization techniques are employed. For each transactional instruction e, a $status$ field (with initial value ACTIVE) is maintained in its entry, indicating that e has not yet been performed. As soon as a working thread completes the execution of e, it changes e's $status$ to DONE.

In order to atomically update some t-variable x, the value of x is stored, together with a *version number*, in a CAS object. Recall that several instances of a write instruction to some t-variable x which is contained in a loop block are executed (one for each iteration). Let e be any such instance. The working threads executing e should use the same old value for x, so that x is updated consistently; also, they should calculate the same new value for x for the current iteration. To ensure this, the old value of x is maintained into e's entry as a CAS object which in addition to the value of x stores an iteration number; moreover, the new value of x, is calculated by all working threads using the values provided in input data dependencies of e for the current iteration.

Data Structures. For each t-variable x, SemanticTM stores a record of type varrec which is denoted by $tvar_x$ below. This record consists of two fields, the value val of x and its version ver, which is initially 0. SemanticTM maintains an array $Tvar$ of varrecs with size M, where M is the total number of t-variables.

SemanticTM associates with the ith t-variable x_i a t-var list $List[i]$ which maintains the transactional instructions to be applied on x_i; this list is implemented as a singly-linked list and each of its elements is of type entry. Specifically, $List[i]$ is a pointer to the first element of the t-var list of x_i.

For each transactional instruction e, SemanticTM stores a record called entry. The first field of entry, called ins, describes e and depending on the type ($iType$) of ins, it contains the following fields:

1. In case $iType$ = read:
   - $outDD$, an array that contains information for each output data dependency of e; more specifically, for each transactional instruction $eptr$ that depends on e, a pointer is maintained that points to a record of type varrec, i.e. to the element of $eptr \rightarrow inDD$ where e should write its output value.
2. In case $iType$ = write that is applied on some t-variable x:
   - $inDD$, an array that contains information for each of the input data dependencies of e. Each element of $inDD$ is a CAS object containing a record of type varrec. The ver field of each such varrec is initialized with the value 0 with the following exception: when some input data dependency is an across-iteration dependency the ver field is initialized to 1 (as explained in paragraph Loops above). This initial value is the same with the initial value of the iteration counter of the cond instruction of the block that e participates (if any).
   - f, a function that can be applied to the values maintained in $inDD$, when e becomes ready, in order to calculate the new value of x.
   - $oldvrec$, a CAS object that contains an oldvaluerec record. It is used to maintain the old value of x. If e participates in a loop block, this value may be different for each loop iteration.
3. In case $iType$ = cond:
   - $inDD$, an array that contains similar information for each of the input data dependencies of e as for writes above.
   - f, a function that can be applied to the values maintained in $inDD$, when e becomes ready, in order to evaluate whether cond is true or false.
   - $outCD$, an array that contains the output control dependencies of e; specifically, it contains a pointer to the entry record of each transactional instruction participating in e's block.
   - $inCD$, an array that maintains the input control dependencies of e. If e is a loop cond, $inCD$ contains one element for each transactional instruction included in cond's loop block; otherwise, it is not used. Recall that each input control dependency is implemented as a counter; so each element of $inCD$ is a CAS object containing an unsigned integer which is initialized to 0.

In addition to ins, entry contains also the following fields:

- $status$, a single bit that describes the $status$ of a transactional instruction e. It is initially ACTIVE and changes to DONE after the execution of e has been completed.
- $loop$, a boolean that is true when e either participates in some loop's block or it is a loop's cond; otherwise, it is false.
- $pCond$, a pointer which is either $null$ (if e does not participate in some block), or points to the entry record of the unique cond instruction from which e has an input control dependency (otherwise).
- $iCond$, if e participates to some loop's block, then it is an index in the array $pCond \rightarrow inCD$ where e should write, in order to indicate that it has been executed for each specific iteration.
- cnt, a CAS object that contains an unsigned integer with initial value 0. It is used only if e is a loop cond or if e participates in some block. Specifically, if e is a loop cond, cnt is used to determine the current loop iteration. Each time the next iteration $k \geq 1$ is ready to start its execution, it is updated from $k-1$ to k. If e participates in some loop, cnt is used to resolve the input control dependency of e. When $cnt = k-1$, $k > 1$, the input control dependency of e has been resolved for the $(k-1)$st but not for the kth iteration of the loop. Once all loop iterations have been executed, the status of e will change from ACTIVE to DONE. Finally, if e participates in some block, but not to some loop, cnt is used to resolve the input control dependency of e; so, it changes from 0 to 1, when this block's cond is evaluated successfully.
- $next$, a pointer which is either $null$, or points to the next entry record of the t-val list that e resides.

Description of Pseudocode. The pseudocode of SemanticTM is presented in Figures 7 and 8.

EXECUTEINS. A working thread executes function EXECUTEINS (Figure 7). EXECUTEINS repeatedly chooses a t-variable x (lines 19 and 20) and executes consecutive ready transactional instructions contained in it, in order, starting from the first. Recall that, if some instruction e in x's t-var list participates in a loop block and has been performed for the current iteration, other instructions in later positions of the list may be ready to execute for this iteration before e becomes ready again. Therefore, EXECUTEINS calls function CHOOSEINS to find the first ready transactional instruction in x's t-var list. If CHOOSEINS returns $null$, no instruction in this t-var list is ready to be executed and EXECUTEINS continues by selecting some other t-var list (line 23). Otherwise, EXECUTEINS performs the ready instruction using function PERFORMINS (line 24) which is described bellow. When the execution of e is completed, then the head pointer of x's t-var list is updated (line 25), so that it points to the next instruction (if any) after e in this list.

CHOOSEINS, CHECKCD, CHECKDD. CHOOSEINS (lines 27 - 52) takes as a parameter a pointer to an element el of a t-var list L; the

```
1   shared $Tvar[M]$: varrec
        /* metadata for each t-variable */
2   shared $Lists[M]$: ptr to entry
        /* tvar-list of each t-variable */

3   type varrec
4       $val$: value
5       $ver$: unsigned integer
            /* used to avoid ABA problems when
            updating $val$ or input data dependencies */

6   type oldvaluerec  /* stores the old value of a t-variable */
7       $oldv$: varrec
8       $inum$: unsigned integer

9   type entry
10      $ins$: $\{\langle iType$ : read,
                $outDD[]$ : ptr to varrec$\rangle$,
            $\langle iType$ : write,
                $inDD[]$ : varrec, $f$ : $function$, $oldvrec$ : oldvaluerec$\rangle$,
            $\langle iType$ : cond,
                $inDD[]$ : varrec, $f$ : $function$, $outCD[]$ : ptr to entry,
                $inCD[]$ : unsigned integer$\rangle\}$  /* if it is a loop's cond, then $inCD$ is used to ensure that a new iteration starts
                                                        after all its block instructions have been executed for the current iteration */
11      $status$: {ACTIVE, DONE}
12      $loop$ : boolean              /* true if $ins$ either participates in some loop block or it is a cond of a loop */
13      $pCond$ : ptr to entry        /* if $ins$ participates in some block, it points to the block's cond */
14      $iCond$ : unsigned integer    /* if $ins$ participates in some loop's block, it is an index in the array of $pCond \to inCD$
                                         where $ins$ should write */
15      $cnt$: unsigned integer       /* if $ins$ is not a loop cond, it implements its input control dependency (if any);
                                         otherwise, it implements the iteration counter of cond */
16      $next$: ptr to entry
```

Figure 6. Data structures of SemanticTM.

execution of all elements preceding el in L has been completed. It returns a pointer ($eptr$) to the topmost element of L that is ready (as well as some information about its status and its input dependencies, as described bellow).

Starting from el, CHOOSEINS tries to find a (ready) transactional instruction with $status$ = ACTIVE. For each transactional instruction $eptr$[1] that it traverses, it first reads $eptr \to status$ and $eptr \to cnt$ (line 31) in local variables $status$ and cnt, respectively, and checks whether $eptr$ has been completed. If this true, CHOOSEINS continues with the next transactional instruction in L. Otherwise, CHOOSEINS continues by checking whether $eptr$ is ready or not (lines 32 to 52).

If some previous instruction (in $eptr$'s t-var list) contained in some loop block has already been traversed, then CHOOSEINS maintains this loop's cond and current iteration, using local variables $curlcond$ and $curliter$ (line 27), respectively. If this is the case (1st condition of line 33) and if $eptr$ either participates in some other loop (2nd condition of line 33), or it is performed for a different iteration (i.e., other than $curliter$) of the loop (3rd condition of line 33), then CHOOSEINS returns $\langle null, \perp, \perp\rangle$, identifying that no ready instruction has been found. Otherwise, CHOOSEINS continues as follows.

First the case where $eptr$ is a cond instruction is examined (line 34). If $eptr$ is not a loop's cond (1st condition of line 35) or it is executed for the first time (2nd condition of line 35), then it is a potential instruction to execute. So, the execution of the while loop of line 29 is terminated and CHOOSEINS continues by checking whether $eptr$'s input data dependencies have been resolved or not (lines 50 to 52). Otherwise, $eptr$ is a loop's cond and CHOOSEINS checks whether its control dependencies have been resolved for the current iteration (line 36) by calling CHECKCD. CHECKCD (lines 54 to 56) returns true if all input control dependencies of $eptr$ are resolved for the current iteration (i.e., their iteration number matches the current iteration number which is provided by it cnt parameter); otherwise, it returns false.

If the control dependencies of $eptr$ are resolved (line 36), then CHOOSEINS returns $eptr$, together with cnt and the values of $eptr$'s input data dependencies (that is array $values$). If $eptr$'s control dependencies are not resolved (line 39), then it may be that other ready instructions of its loop block reside in later positions of L should be executed; so, CHOOSEINS maintains a copy of this loop's cond and its current iteration in $curlcond$ and $curliter$, respectively.

Next we examine the case where $eptr$ is not a cond (line 40). If $eptr$ does not participate in some block (line 41), then it is a potential instruction to execute so the execution of the while loop of line 29 is terminated. If $eptr$ participates in some block, but not to some loop's block (line 42) and its input control dependency has been resolved (line 43), then it is a potential instruction to execute so the execution of the while loop of line 29 is terminated; on the other hand, if its input control dependency has not been resolved (line 44), then CHOOSEINS returns $\langle null, \perp, \perp\rangle$, identifying that no ready instruction has been found

If $eptr$ participates in some loop, then this loop's current iteration is read (line 45). If either the cond of $eptr$'s loop has not been executed for the first time ($1st$ condition of the if of line 46), or $eptr$'s input control dependency is not resolved for the current iteration (2nd condition of the if of line 46), then CHOOSEINS returns that no ready instruction has been found. Otherwise, we check if $eptr$ has not been performed for the current iteration of this loop (line 47). If this is the case, $eptr$ is a potential instruction to execute so the execution of the while loop of line 29 is terminated. Otherwise, some other ready instruction of the same loop in later positions of L should be executed. So, a copy of this loop's cond and current iteration are maintained in $curlcond$ and $curliter$, respectively.

After the termination of the while loop of line 29, CHOOSEINS checks if $eptr$ is ready by calling CHECKDD (line 50). If the input data dependencies of $eptr$ have been resolved for the current iteration $iter$, CHECKDD returns true together with the values of the input data dependencies of $eptr$ (that is array $values$); otherwise, it returns false. In the former case, CHOOSEINS returns $eptr$, together with $stat$ and $values$ (line 51), whereas in the latter case, it returns $\langle null, \perp, \perp\rangle$ (line 52).

CHECKDD (lines 58 - 61) takes as a parameter a pointer $eptr$ to some transactional instruction's entry record and an iteration number $iter$. If $eptr$ is a read instruction, then true is returned (line 58). Otherwise, CHECKDD (lines 58 to 60) checks whether the iteration number of each input data dependency of $eptr$ matches $iter$ (line 59). If this is true, CHECKDD returns true; otherwise, it returns false.

PERFORMINS, UPDATETVAR, RESOLVEDD, and RESOLVECD. PERFORMINS (lines 63 - 77) takes as parameters a pointer $eptr$ to the entry record of some transactional instruction, an unsigned integer cnt that is the current iteration in which $eptr$ is executed,

[1] Notice that from this point on, we use $eptr$ to refer both to the t-variable's name and to the pointer to its entry record.

```
17  EXECUTEINS() by working thread $p$:
18      while (true) do                                        /* as long as there is work in the system */
19          $i$ := randomly choose an integer from 1 to $M$     /* choose a t-variable */
20          $ptvar := \&Tvar[i]$                                /* maintain pointer $ptvar$ to the varrec record of this t-variable */
21          while (true) do                                    /* repeatedly try to find ready transactional instructions from the chosen list */
22              $\langle eptr, cnt, values\rangle$:= CHOOSEINS($List[i]$)   /* choose a ready instruction $eptr$ from $pstart$'s t-var list */
23              if ($eptr = null$) then break                  /* if no such instruction exists, then skip this t-var list */
24              PERFORMINS($eptr, cnt, ptvar, values$)         /* perform the chosen instruction */
25              if ($eptr \rightarrow status$ = DONE) then $List[i] := eptr \rightarrow next$   /* if $eptr$ completed, then update t-var list */

26  $\langle$ptr to entry, unsigned integer, $values[]$ : value$\rangle$ CHOOSEINS($pstart$ : ptr to entry) by working thread $p$:
27      $\langle curlcond, curliter\rangle := \langle null, \bot\rangle$   /* initialize the pointer to current loop's cond entry (if any) and its current iteration */
28      $eptr := pstart$
29      while (true)                                           /* as long as completed or performed for the current iteration instructions are reached */
30          if ($eptr = null$) then return $\langle null, \bot, \bot\rangle$   /* if the end of $eptr$'s t-var list has been reached, return that no ready transactional instruction has been found */
31          $\langle status, cnt\rangle := \langle eptr \rightarrow status, eptr \rightarrow cnt\rangle$   /* read the current values of $eptr$'s $status$ and $cnt$ fields */
32          if ($status \neq$ DONE) then                       /* if $eptr$ has not been performed yet */
                /* if some previous instruction (in $eptr$'s t-var list) contained in a loop block has already been traversed, and if $eptr$ either participates in
                   some different loop or it has been performed for a different iteration of this loop, then return that no ready instruction has been found */
33              if ($curlcond \neq null$ AND ($curlcond \neq eptr \rightarrow pCond$ OR $curliter \neq cnt$)) then return $\langle null, \bot, \bot\rangle$
34              if ($eptr \rightarrow ins.iType$ = cond) then  /* if $eptr$ is a cond */
35                  if ($eptr \rightarrow loop$ = false OR $cnt = 0$) then break   /* if either $eptr$ is not a loop's cond or this is the first time to reach $eptr$,
                                                                                     $eptr$ is a potential instruction to execute $\rightarrow$ terminate while */
36                  if (CHECKCD($eptr, cnt$) = true) then      /* if $eptr$'s input control dependencies have been resolved */
37                      $values[1..k] := \{eptr \rightarrow ins.inDD[1].val, \ldots, eptr \rightarrow ins.inDD[k].val\}$   /* read input values, and return */
38                      return ($eptr, cnt, values$)
39                  else $\langle curlcond, curliter\rangle := \langle eptr, cnt\rangle$   /* if $eptr$'s input control dependencies have not been resolved, other ready instructions of its
                                                                                         loop block in later positions of the t-var list should be executed; so, remember this loop */
40              else                                           /* if $eptr$ is not a cond */
41                  if ($eptr \rightarrow pCond = null$) then break   /* if $eptr$ does not participate in some block, it is a potential instruction to execute $\rightarrow$ terminate while */
42                  if ($eptr \rightarrow loop$ = false) then   /* if $eptr$ participates to some block, but not to some loop's block */
43                      if ($eptr \rightarrow cnt = 1$) then break   /* if the input control dependecy of $eptr$ has been resovled, it is a potential intruction to execute $\rightarrow$ terminate while */
44                      else return $\langle null, \bot, \bot\rangle$   /* otherwise, if the input control dependency of $eptr$ has not been resolved, return that no ready transactional instruction has been found */
45                  $liter := eptr \rightarrow pCond \rightarrow cnt$   /* read loop's current iteration */
                    /* if either the execution of $eptr$'s loop has not started yet, or $eptr$'s input control dependency is not resolved for the current iteration, then return that no ready transactional instruction has been found */
46                  if ($cnt = 0$ OR $cnt \neq liter$) then return $\langle null, \bot, \bot\rangle$
                    /* the input control dependency of $eptr$ has been resolved. If $eptr$ has not been performed yet, then it is a potential instruction to execute $\rightarrow$ terminate while */
47                  if ($eptr \rightarrow pCond \rightarrow inCD[eptr \rightarrow iCond] = liter - 1$) then break
                    /* if $eptr$ has been performed for the current iteration, other ready instructions of the same loop block in later
                       positions of the t-var list should be executed; so record information about this loop (if not done already) to remember it */
48                  else if ($curlcond = null$) then $\langle curlcond, curliter\rangle := \langle eptr \rightarrow pCond, liter\rangle$
49          $eptr := eptr \rightarrow next$                      /* traverse next instruction */
50      $\langle bool, values\rangle :=$ CHECKDD($eptr, cnt$)   /* check if input data dependencies of $eptr$ have been resolved */
51      if ($bool$ = true) then return ($eptr, cnt, values$)   /* if yes, return $eptr$ and related info */
52      else return $\langle null, \bot, \bot\rangle$          /* if no, return NULL */

53  boolean CHECKCD($eptr$ : ptr to entry, $iter$ : unsigned integer) by working thread $p$:
54      for each element $el \in eptr \rightarrow ins.inCD$ do   /* $eptr$ is a cond instruction; check whether $eptr$'s input control dependencieshave been resolved for iteration $iter$ */
55          if ($el \neq iter$) then return (false)            /* if not, return false */
56      return (true)                                          /* otherwise, return true */

57  $\langle$boolean, $values[]$ : value$\rangle$ CHECKDD($eptr$ : ptr to entry, $iter$ : unsigned integer) by working thread $p$:
58      if ($eptr \rightarrow itype$ = read) then return $\langle$true, $\bot\rangle$   /* if $eptr$ is a read instruction, then it has no input data dependencies; so, return true */
        for each element $d \in eptr \rightarrow ins.inDD$ with index $j$ do   /* it is checked whether each of $eptr$'s input data dependencies is resolved or not, for iteration $iter$) */
59          if ($d.ver \neq iter$) then return $\langle$false, $\bot\rangle$   /* if this is not true, false is returned */
60          $values[j] := d.val$                                /* if this is true for dependency $d$, $d$'s value is maintained into $values$ array */
61      return $\langle$true, $values\rangle$                  /* if all the input data dependencuies of $eptr$ are resolved, then their values are returned */
```

Figure 7. Pseudocode for EXECUTEINS, CHOOSEINS, CHECKCD and CHECKDD of SemanticTM.

```
62 PerformIns($eptr$ : ptr to entry, $cnt$ : unsigned integer, $ptvar$ : ptr to varrec, $values[1..k]$ : value)
63   if ($eptr \rightarrow ins.iType =$ cond) then                         /* if $eptr$ is cond */
64     $decision := eptr \rightarrow ins.f(values)$                         /* evaluate its condition */
65     if ($decision =$ true AND $eptr \rightarrow loop =$ true) then    /* if condition is evaluated to true and $eptr$ is a loop cond */
66       CAS($eptr \rightarrow cnt, cnt, cnt + 1$)                          /* $eptr$'s $cnt$ is increased by one to prepare $eptr$ for the next iteration */
67       ResolveCD($eptr, cnt, decision$)                                   /* resolve $eptr$'s output control dependencies based on $decision$ for iteration $cnt$ */
68     else                                                                 /* otherwise, $eptr$ is marked as completed */
69       ResolveCD($eptr, cnt, decision$)                                   /* resolve $eptr$'s output control dependencies based on $decision$ for iteration $cnt$ */
70       $eptr \rightarrow status :=$ DONE
71   else                                                                   /* otherwise, $eptr$ is not cond */
72     if ($eptr \rightarrow ins.iType =$ read) then ResolveDD($eptr, cnt, ptvar$)   /* if $eptr$ is read, resolve its output data dependencies */
73     else                                                                 /* otherwise $eptr$ is write, so calculate $ptvar$'s new value and update it */
74       $newval := eptr \rightarrow f(values)$
75       UpdateTvar($ptvar, newval, eptr, cnt$)
                                                 /* if $eptr$ participates in some loop, resolve its output control dependency */
76     if($eptr \rightarrow loop =$ true) then CAS($eptr \rightarrow pCond \rightarrow inCD[eptr \rightarrow iCond], cnt - 1, cnt$)
77     else $eptr \rightarrow status :=$ DONE                               /* otherwise, mark $eptr$ as completed */

78 UpdateTvar($ptvar$ : ptr to varrec, $newval$ : value, $eptr$ : ptr to entry, $cnt$ : unsigned integer) by working thread $p$:
79   $data := *ptvar$                                                       /* read the current value of the t-variable's varrec record */
80   CAS($eptr \rightarrow ins.oldvrec, \langle eptr \rightarrow ins.oldvrec.oldv, cnt - 1\rangle, \langle data, cnt\rangle$)   /* try to store $data$ into $oldvrec$ field of $eptr$ */
81   $\langle oldv, inum\rangle := eptr \rightarrow ins.oldvrec$            /* read $oldvrec$ field of $eptr$ */
82   if ($inum \neq cnt$) then return                                      /* if instruction has already been performed for iteration $stat.cnt$, then return */
83   CAS($ptvar, oldv, \langle newval, oldv.ver + 1\rangle$)               /* otherwise, store $newval$ into $ptvar$'s record and increment its version */

84 ResolveDD($eptr$ : ptr to entry, $iter$ : unsigned integer, $ptvar$ : ptr to varrec) by working thread $p$:
85   $val := ptvar \rightarrow val$                                         /* read the value of the $ptvar$'s $val$ field */
86   for each element $d \in eptr \rightarrow ins.outDD$ do                /* for each output data dependency of $eptr$ */
87     $curval := d \rightarrow val$                                        /* read the current value of the $val$ field of this dependency... */
88     CAS($d, \langle curval, iter - 1\rangle, \langle val, iter\rangle$)   /* ... and update it with the $val$ of $ptvar$, using this dependency's current $val$ and iteration $iter$ */

89 ResolveCD($eptr$ : ptr to entry, $iter$ : unsigned integer, $decision$ : boolean) by working thread $p$:
90   if ($decision := true$) then                                          /* if decision is to continue, then for each output dependency of $eptr$ its $cnt$ is incremented by one */
91     for each element $d \in eptr \rightarrow ins.outCD$ do CAS($d \rightarrow cnt, iter, iter + 1$)
92   else for each element $d \in eptr \rightarrow ins.outCD$ do $d \rightarrow status :=$ DONE   /* otherwise, each output dependecy of $eptr$ is marked as completed */
```

Figure 8. Pseudocode for ChooseIns, UpdateTvar, ResolveDD, and ResolveCD of SemanticTM.

a pointer to the varrec record of the t-variable on which $eptr$ is applied (when $eptr \rightarrow iType \in \{$read, write$\}$), and an array $values$ containing the values of the input data dependencies of $eptr$.

If $eptr$ is a cond, its condition is evaluated (line 64) and the result is stored in the local variable $decision$. If $decision$ is true and $eptr$ is a loop cond, then it is re-initiated for the next loop iteration (line 66) and the output control dependencies of $eptr$ are resolved for the current iteration by calling function ResolveCD. If either $decision$ is false or $eptr$ is not a loop's cond, the output control dependencies of $eptr$ are resolved and $eptr$ is marked as completed (line 70). We remark that if $decision$ is true, ResolveCD (lines 90 - 92) increments by one the counter field of each output control dependency of $eptr$ (line 91); otherwise, each output dependency of $eptr$ is marked as completed (line 92).

We now discuss the case where $eptr$ is not a cond (line 71). If the type of $eptr$ is read (line 72), its output data dependencies are resolved for the current iteration cnt, by calling function ResolveDD. ResolveDD (lines 85 - 88) starts by reading the current data $data$ of $ptvar$ and tries to write it into each of the output dependencies of $eptr$, using the old data of this dependency and the current iteration number.

If the type of $eptr$ is a write on some $ptvar$ (line 73), then the new value $newval$ of $ptvar$ is calculated using $values$ (line 74) and then $ptvar$ is updated with $newval$, by calling function UpdateTvar (line 75). Specifically, UpdateTvar (lines 79 - 83) takes as parameters $tvar$, $newval$, $eptr$, and cnt. It starts by trying to store the current value of x's varrec together with the current iteration (cnt) into $eptr \rightarrow ins.oldvrec$ (line 80). Then, it checks whether $eptr$ is already performed (line 82); if this is true, it returns. Otherwise, the varrec record of x is atomically (using CAS) updated (line 83) using $eptr \rightarrow ins.oldvrec.oldv$ (as the first parameter of this CAS) and $newv$ together with the stored version of $tvar$ (that is $eptr \rightarrow ins.oldvrec.ver$) incremented by one (as the second parameter of this CAS).

Finally, if $eptr$ participates in some loop (line 76) its output control dependency is resolved; otherwise, its $status$ is updated to DONE (line 77).

3. Discussion

SemanticTM can easily cope with nested conditionals as follows. Consider an if instruction e_2 which participates in the block of an outer if instruction e_1. The scheduler will add an output control dependency from e_1 to e_2 but no such dependency from e_1 to the instructions of the block of e_2 and vice versa. This simple technique is enough to support nested conditionals. For nested loops, a similar technique can be employed. However, extra fields are required in the entry record to cope appropriately with the iteration counters of instructions of inner blocks.

The current version of SemanticTM assumes that each transaction accesses a known set of t-variables. This restriction can be overcome by using wildcards; roughly speaking, a *wildcard* is an instruction which accesses a t-variable known only at runtime. As an example, consider a transaction that accesses an element of an array but which exactly element becomes known only at run time.

To cope with this (or similar) case(s), SemanticTM can maintain a t-var list L for the entire array, as well as one list L_i, $1 \leq i \leq m$, for each of its elements, where m is the size of the array. The scheduler places each instruction e that accesses a (possibly unknown) element of the array in L. When later (at runtime), the specific element i to be accessed by e becomes known, e is moved in list L_i. A similar strategy may work for supporting dynamic memory allocation, if we consider the memory heap as an array.

Recall that in the current version of SemanticTM there are output dependencies from all instructions of a block to its `cond` and vice versa. However, the scheduler may choose to add such dependencies from `cond` to those instructions that do not depend on other block instructions, since the rest have to wait for the first instructions in any case before they are ready to execute. Moreover, no output control dependencies to a block's `cond` from those block instructions that do not contribute to the evaluation of the `cond` are needed. Such optimizations may have positive impact on the performance of SemanticTM.

Designing a blocking version of SemanticTM will be much simpler than the version presented here since it will not have to cope with several threads executing the same instruction to achieve fault tolerance. Experimental work to compare the performance of different versions of SemanticTM, as well as that of SemanticTM with the performance of existing TM algorithms is left for the future.

Acknowledgments

We would like to thank Polyvios Pratikakis for several valuable discussions. Also, we would like to thank the anonymous reviewers for their valuable feedback.

This work was supported by the project "IRAKLITOS II - University of Crete" of the Operational Programme for Education and Lifelong Learning 2007 - 2013 (E.P.E.D.V.M.) of the NSRF (2007 - 2013) which is co-funded by the European Union (European Social Fund) and National Resources. It was also supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project and by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources, through the GreenVM project. S. Dolev was partially supported by a Russian Israeli grant from the Israeli Ministry of Science and Technology and the Russian Foundation for Basic Research, the Rita Altura Trust Chair in Computer Sciences, the Lynne and William Frankel Center for Computer Sciences, Israel Science Foundation (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium, Grant from the Institute for Future Defense Technologies Research named for the Medvedi of the Technion, MAFAT, EMC and the Israeli Internet Association.

References

[1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *26th International Symposium on Distributed Computing*, DISC'12, 2012.

[2] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 4–18, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-92989-5. doi: 10.1007/978-3-540-92990-1_3. URL `http://dx.doi.org/10.1007/978-3-540-92990-1_3`.

[3] H. Attiya and E. Hillel. Single-version stms can be multi-version permissive. In *Proceedings of the 12th international conference on Distributed computing and networking*, ICDCN'11, pages 83–94, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 3-642-17678-X, 978-3-642-17678-4. URL `http://dl.acm.org/citation.cfm?id=1946143.1946151`.

[4] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10876-1. doi: 10.1007/978-3-642-10877-8_3. URL `http://dx.doi.org/10.1007/978-3-642-10877-8_3`.

[5] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, PODC '06, pages 308–315, New York, NY, USA, 2006. ACM. ISBN 1-59593-384-0. doi: 10.1145/1146381.1146428. URL `http://doi.acm.org/10.1145/1146381.1146428`.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-44624-9, 978-3-540-44624-8. doi: 10.1007/11864219_14. URL `http://dx.doi.org/10.1007/11864219_14`.

[7] S. Dolev, D. Hendler, and A. Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 125–134, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-989-0. doi: 10.1145/1400751.1400769. URL `http://doi.acm.org/10.1145/1400751.1400769`.

[8] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 7–16, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-396-9. doi: 10.1145/1582716.1582725. URL `http://doi.acm.org/10.1145/1582716.1582725`.

[9] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345233. URL `http://doi.acm.org/10.1145/1345206.1345233`.

[10] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust contention management in software transactional memory. In *OOPSLA '05 Workshop on Synchronization and Concurrency in Object-Oriented Lanugages (SCOOL '05)*, 2005.

[11] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 258–264, New York, NY, USA, 2005. ACM. ISBN 1-58113-994-2. doi: 10.1145/1073814.1073863. URL `http://doi.acm.org/10.1145/1073814.1073863`.

[12] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273029. URL `http://doi.acm.org/10.1145/1272996.1273029`.

[13] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 305–319, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87778-3. doi: 10.1007/978-3-540-87779-0_21. URL `http://dx.doi.org/10.1007/978-3-540-87779-0_21`.

[14] A. Matveev and N. Shavit. Towards a fully pessimistic stm model. In *7th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT'12, 2012.

[15] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theor. Comput. Sci.*, 130(1):17–47.

[16] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, Sept. 2010. ISSN 2150-8097. URL `http://dl.acm.org/citation.cfm?id=1920841.1920959`.

[17] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-888-9. doi: 10.1145/1835698.1835704. URL `http://doi.acm.org/10.1145/1835698.1835704`.

[18] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an stm. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 163–172, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6. doi: 10.1145/1504176.1504201. URL `http://doi.acm.org/10.1145/1504176.1504201`.

[19] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM. ISBN 1-58113-994-2. doi: 10.1145/1073814.1073861. URL `http://doi.acm.org/10.1145/1073814.1073861`.

[20] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM. ISBN 1-58113-994-2. doi: 10.1145/1073814.1073861. URL `http://doi.acm.org/10.1145/1073814.1073861`.

[21] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 285–296, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378584. URL `http://doi.acm.org/10.1145/1378533.1378584`.

[22] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378564. URL `http://doi.acm.org/10.1145/1378533.1378564`.