Abort Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions*

(Extended Abstract)

Shlomi Dolev[1], Panagiota Fatourou[2], and Eleftherios Kosmas[2]

[1] Ben-Gurion University of the Negev
dolev@cs.bgu.ac.il
[2] University of Crete & FORTH-ICS
{faturu,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 among them occurs by 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 Transactional Memory (TM) where pieces of code that may access data that become 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*, so 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 at least 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 [7]). 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 have the additional benefit that they 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.

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

* An extended version has been presented in the 2013 Workshop on Transactional Computing (TRANSACT).

it accesses. A set of worker threads execute instructions from the lists, in order. 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 paper. It is remarkable that SemanticTM is wait-free in this case.

`TM` algorithms that never abort transactions have been recently presented in [1, 3, 10, 11]. Although read-only transactions in these algorithms are wait-free, the algorithms in [1, 10] restrict parallelism by executing all update transactions sequentially using a global lock; on the other hand, in the algorithms presented in [3, 11] update transactions may abort and they require locks to execute some of the transactional instructions. Work on transactional scheduling [2, 4, 6, 15] is related to SemanticTM but most transactional schedulers use locks and aborts are not avoided. In [12], a lock-based dependence-aware `TM` system is presented which dynamically detects and resolves conflicts. 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 for simple transactions, all transactions will always commit within a bounded number of steps.

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. Some *worker* threads execute instructions from the t-var lists. We use compiler support to know, for each instruction, any dependency that lead to or originate from it. Figure 1 shows the main structure of SemanticTM.

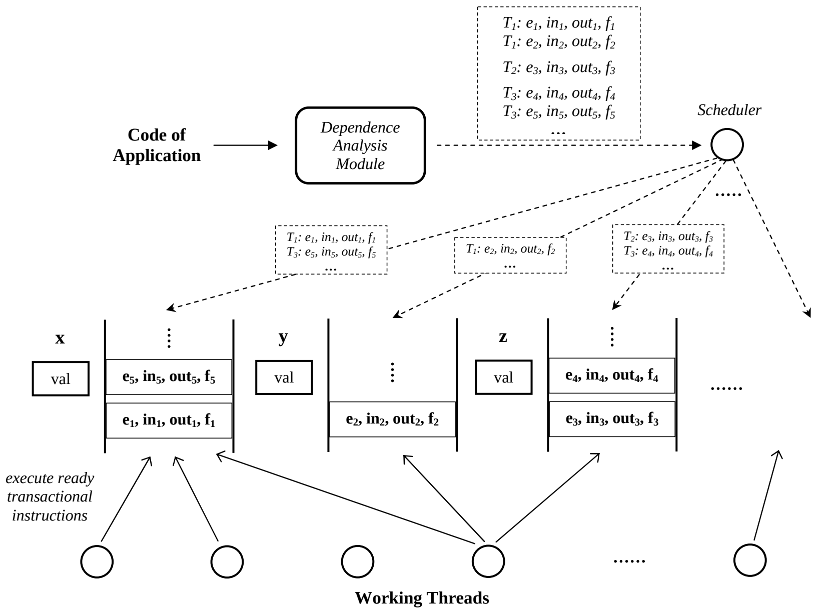

Fig. 1. Main components of SemanticTM. For simple transactions the scheduling component is not needed.

In SemanticTM, all the instructions of each transaction T are placed in the t-var lists before the instructions of any subsequent transaction. Each of the workers repeatedly chooses, *uniformly at random*, a t-var list and executes the instructions of this list, starting from the first ready. Processing transactions in this way ensures that conflicts never occur; so, transactions never abort. As an example, consider the simple transactions T_1, T_2 of Figures 2, 3, respectively. Since they both read and write t-variables x and y, there are conflicts between them. 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 lists when they execute them, the instructions of T_1 on each 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.

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

Fig. 2. T_1.

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

Fig. 3. T_2.

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

Fig. 4. T_3.

The set of t-variables accessed by a transaction is its *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 *instructions* of a transaction are `read`, `write`, and `cond` instructions. We call *block* the set of its instructions in the body of a control flow statement; so each `cond` instruction is associated with a block.

Dependencies. If the execution of an 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 `write` e_3 to x (line 2), a `read` e_4 on x and a `write` e_5 to y for line 3. There is an output dependency from e_1 to e_2 and one from e_3 to e_4. 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 thus the `read` can 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 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. Table 1 (in Appendix **??**) provides a brief description of all possible dependencies for each instruction. The state of an instruction is *waiting*, if at least one of its input dependencies has not been resolved, otherwise, it is *ready*; an 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 instructions e_1 and e_2 from a transaction T_1, e_3 from T_2, and e_4 and e_5 from 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, and 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. Then, at runtime, only one of the `cond` instructions will be evaluated to `true`, whereas all the others will be evaluated to `false` and their blocks' instructions will be invalidated by the working threads that execute these `conds`. In the current version of SemanticTM a `cond` instruction 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 `conds`' blocks. In SemanticTM outside block dependencies are resolved in a direct

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.

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.*

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.

Acknowledgements. This work has been 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), co-funded by the European Union (European Social Fund) and National Resources It has also been 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.

The research has also been supported by Israel Science Foundation (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium, Deutsche Telekom Labs at BGU, Orange Research Labs, EMC, the Israeli Ministry of Science and Technology (MOST), the Institute for Future Defense Technologies Research named for the Medvedi, Shwartzman and Gensler Families, the Israel Internet Association (ISOC-IL), the Lynne and William Frankel Center for Computer Science at Ben-Gurion University, and the Rita Altura Trust Chair in Computer Science.

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.

[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.

[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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.