STSM Report : Theory of transactional memory

1 Purpose of the STSM

Due to the recent proliferation of multicore machines, simplifying concurrent programming has become a necessity, to exploit their computational power. A *universal construction* [3] is a methodology for automatically executing pieces of sequential code in a concurrent environment, while ensuring correctness. Universal constructions provide concurrent implementations of any sequential data structure: Each operation supported by the data structure is a piece of code that can be executed. Universal constructions provide functionality similar to Transactional Memory (TM) [4].

We are interested in universal constructions that allow for increased parallelism by being disjoint-access parallel. Roughly speaking, an implementation is *disjoint-access parallel* if two processes that operate on disjoint parts of the simulated state do not interfere with each other, i.e., they do not access the same base objects. Therefore, disjoint-access parallelism allows unrelated operations to progress in parallel. We are also interested in ensuring strong progress guarantees : An implementation is *wait-free* if, in every execution, *each* (non-faulty) process completes its operation within a finite number of steps, even if other processes may fail (by crashing) or are very slow.

In a previous work [2] we first proved that designing universal constructions which ensure both disjoint access parallelism and wait-freedom is not possible. We proved this impossibility result by considering a dynamic data structure that can grow arbitrarily large during an execution. Specifically, we considered a singly-linked unsorted list of integers that supports the operations Append(x), which appends x to the end of the list, and Search(x), which searches the list for x starting from the first element of the list. We showed that, in any implementation resulting from the application of a universal construction to this data structure, there is an execution of Search that never terminates.

Some impossibility results, related to ours, have been proved for transactional memory implementations. A discussion on these results is provided in [2]. Universal constructions and transactional memory algorithms are closely related. They both have the same goal of simplifying parallel programming by providing mechanisms to efficiently execute sequential code in a concurrent environment. A transactional memory algorithm informs the external environment when a transaction is aborted, so it can choose whether or not to re-execute the transaction. A call to a universal construction returns only when the simulated code has been successfully applied to the simulated data structure. This is the main difference between these two paradigms. However, it is common behavior of an external environment to restart an aborted transaction until it eventually commits. Moreover, meaningful progress conditions [1, 6] in transactional memory require that the number of times each transaction aborts is finite. Our impossibility result applies to transactional memory algorithms that satisfy this progress property. Disjoint-access parallelism is defined for transactions in a similar way as for universal constructions.

In [2] we also showed how this impossility result can be circumvented, by restricting attention to data structures whose operations can each only access a bounded number of different data items. Stacks and queues are examples of dynamic data structures that have this property. We present a universal construction that ensures wait-freedom and disjoint-access parallelism for such data structures. The resulting concurrent implementations are linearizable [5] and satisfy a much stronger disjoint-access parallelism property than we used to prove the impossibility result. For other dynamic data structures, our universal construction still ensures linearizability and disjoint-access parallelism. However then, instead of wait-freedom, it only ensures lock-freedom which guarantees that, in every execution, *some* (non-faulty) process completes its operation within a finite number of steps. An interesting question that is not answered in [2] is whether we can circumvent the impossibility result of [2] by relaxing the notion of disjoint-access parallelism. An appealing way to do so is by allowing processes that execute operations to access a restricted set of common base object even in case they operate on disjoint parts of the data structure. In particular, we consider implementations where processes can access a shared timestamp object even if they do not conflict on some data item. Our goal is to study whether such implementations can be both wait-free and disjoint-access parallel in respect of the remaining set of base objects

2 Description of the work carried out during the STSM

2.1 Preliminaries

A *data structure* is a sequential implementation of an abstract data type. In particular, it provides a representation for the objects specified by the abstract data type and the (sequential) code for each of the operations. A *data item* is a piece of the representation of an object implemented by the data structure. For example, the data items are the nodes of the singly-linked list. The *state* of a data structure consists of the collection of data items in the representation and a set of values, one for each of the data items.

A *static* data item is a data item that exists in the initial state. In our single linked list example, the pointers first and last, pointing to the first and last element of the list, respectively, are static data items. When the data structure is dynamic, the data items accessed by an instance of an operation (in a sequential execution α) may depend on the instances of operations that have been performed before it in a. For example, the set of nodes accessed by an instance of SEARCHdepends on the sequence of nodes that have been previously appended to the list.

We consider an *asynchronous shared-memory* system with n processes $p_1, \ldots, p_n$ that communicate by accessing shared objects, such as *registers* and LL/SC objects. A register R stores a value from some set and supports the operations read(R), which returns the value of R, and write(R, v), which writes the value v in R. An LL/SC *object* R stores a value from some set and supports the operations LL, which returns the current value of R, and SC. By executing SC(R, v), a process p_i attempts to set the value of R to v. This update occurs only if no process has changed the value of R (by executing SC) since p_i last executed LL(R). If the update occurs, true is returned and we say the SC is successful; otherwise, the value of R does not change and false is returned.

A *universal construction* supports a single operation, called PERFORM, which takes as parameters a piece of sequential code and a list of input arguments for this code. The algorithm that implements PERFORM applies a sequence of operations on shared objects provided by the system. We use the term *base objects* to refer to these objects and we call the operations on them *primitives*. A primitive is *non-trivial* if it may change the value of the base object; otherwise, the primitive is called *trivial*.

A *configuration* provides a global view of the system at some point in time. In an *initial configuration*, each process is in its initial state and each base object has its initial value. A *step* consists of a primitive applied to a base object by a process and may also contain local computation by that process. An *execution* is a (finite or infinite) sequence $C_i, \phi_i, C_{i+1}, \phi_{i+1}, \ldots, \phi_{j-1}, C_j$ of alternating configurations (C_k) and steps (ϕ_k), where the application of ϕ_k to configuration C_k results in configuration C_{k+1}, for each $i \leq k < j$.

An execution is *solo* if all its steps are taken by the same process.

From this point on, for simplicity, we use the term operation to refer to an instance of an operation. The *execution interval* of an operation starts with the first step of the corresponding call to PERFORM and terminates when that call returns. Two operations *overlap* if the call to PERFORM for one of them occurs during the execution interval of the other. If a process has invoked PERFORM for an operation that has not yet returned, we say that the operation is *active*. A process can have at most one active operation in any configuration. A configuration is *quiescent* if no operation is active in the configuration.

Let α be any execution. We assume that processes may experience *crash failures*. If a process p does not fail in α, we say that p is *correct* in α. *Linearizability* [5] ensures that, for every completed operation in α and some of the uncompleted operations, there is some point within the execution interval of the operation called its *linearization point*, such that the response returned by the operation in α is the same as the response it would return if all these operations were executed serially in the order determined by their linearization points. When this holds, we say that the responses of the operations are *consistent*. An implementation is *linearizable* if all its executions are linearizable. An implementation is *wait-free* [3] if, in every execution, each correct process completes each operation it performs within a finite number of steps.

Since we consider linearizable universal constructions, every quiescent configuration of an execution of a universal construction applied to a sequential data structure defines a state. This is the state of the data structure resulting from applying each operation linearized prior to this configuration, in order, starting from the initial state of the data structure.

Fix any execution $\alpha = C_0, \phi_0, C_1, \phi_1, \ldots$, produced by a linearizable universal construction U. Then there is some linearization of the completed operations in α and a subset of the uncompleted operations in α such that the responses of all these operations are consistent. Let *op* be any one of these operations, let I_{op} be its execution interval, let C_i denote the first configuration of I_{op}, and let C_j be the first configuration at which *op* has been linearized. Since each process has at most one uncompleted operation in α and each operation is linearized within its execution interval, the set of operations linearized before C_i is finite. Let S_k denote the state of the data structure at configuration C_k, which results from applying each operation linearized prior to configuration C_k, in order, starting from the initial state of the data structure. Define $DS(op, \alpha)$, the data set of *op* in α, to be the set of all data items accessed by *op* when executed by itself starting from S_k, for $i \leq k < j$.

The *conflict graph* of an execution interval I of α is an undirected graph, where vertices represent operations whose execution intervals overlap with I and an edge connects two operations *op* and *op*$'$ if and only if $DS(op, \alpha) \cap DS(op', \alpha) \neq \emptyset$. Two operations *contend* on a base object b if they both apply a primitive to b and at least one of these primitives is non-trivial.

Consider an implementation that uses a set of base objects $\mathcal{B}$ and a timestamp object $T \notin \mathcal{B}$. We say that an execution α is *partial disjoint-access parallel* if, for every overlapping pair of operation instances I and I' in α that access at least one base object in B in common, I and I' are connected in the conflict graph of the minimal execution interval that contains both I and I. An implementation is partial disjoint-access parallel if all its executions are partial disjoint-access parallel. A universal construction is *partial disjoint-access parallel* if all implementations resulting from it are partial disjoint-access parallel.

A timestamp object supports two operations: *GetTimestamp*, which returns a timestamp, (without loss of generality, we may assume that two different instances of *GetTimestamp* in the same execution never return the same timestamp) and $Compare(t, t')$, which returns true if the instance of *GetTimestamp* that returned t finished before the instance of *GetTimestamp* that returned t' began.

2.2 Preliminary results

We have studied the possibility to design a universal construction that ensures both partial disjoint-access parallelism and wait-freedom. It is worth pointing out that many transactional memory algorithms that exhibit good performance, e.g., NoRec, TL2, etc., use a timestamp object. We provide an algorithm that works for any data structure provided that the number of the entry points to the data structure is bounded. We have not yet fully formalized the notion of an entry point to a data structure. Intuitively however, an entry point is a data item belonging to the data structure from which an operation can access other data items that cannot be accessed directly. In our singly linked list example, there are two entry points, namely, the pointers first and last.

In the following we describe the main ideas of our algorithm. It is an extension of the algorithm presented in [2]]: it uses the timestamp object to allow to an operation to distinguish the data items that were added to the data structure after the operation started . This enables the operation to complete without violating disjoint-access parallel.

The algorithm maintains a record of type `oprec` that stores information for each initiated operation. When a process p wants to execute an operation op, it starts by creating a new `oprec` for *op* which it initializes appropriately. In particular, this record provides a pointer to the code of op, its input parameters, its output, the status of op, and an array indicating whether op should help other operations after its completion and before it returns. The algorithm also maintains a record of type `varrec` for each data item x. This record contains a val field, which is an `LL/SC` object that stores the value of x, and an array A of n `LL/SC` objects, indexed by process identifiers, which stores pointers to `oprec` records of operations that are accessing x. This array is used by operations to announce that they access x and to determine conflicts with other operations that are also accessing x.

The execution of an operation *op* is done in a sequence of one or more *simulation phases* followed by a *modification phase*. In a simulation phase, the steps of the operation are simulated locally without modifying the shared representation of the simulated state; specifically, the instructions of the operations are read and the execution of each one of them is simulated.

Each operation *op* stores a timestamp, together with a pointer to the entry point of the data structure from where the operation started, in the shared representation of all the data items that it creates. This timestamp is obtained when *op* enters its modification phase. Operation *op* obtains another timestamp t when it starts its execution. If, during its simulation phase, op accesses a data item whose timestamp is greater than t, op announces itself in the entry point that is recorded in the shared representation of this data item. Each operation entering from this entry point later on will subsequently help op to complete. It is remarkable that we do not use the timestamps to guarantee consistency, but to ensure progress without violating partial disjoint-access parallelism. In particular, the fact that op accesses a data item with a larger timestamp implies that operation *op* that created this data item was concurrent with *op*. Since *op* and *op* both access this data item and op also accessed the entry point, the process executing op can access the base objects associated with the entry point without violating dap. Specifically, op notifies future operations that will enter the data structure through this entry point about its existence so that they help op before exectuting their own operations. In other words, *op* is connected through some path with *op* in the conflict graph, as well as with any other operation that accesses that entry point after *op* and is concurrent with *op*.

We use the same conflict detection and helping mechanisms used in [2]. Each time a data item x is accessed for the first time during the execution of *op* by p, p checks whether there are conflicts with other operations accessing x, before reading the value of x. This is done as follows. The `varrec` record of x contains an *Announce* array, A, indexed by process id, and p *announces op* to x by storing a pointer to *op*'s `oprec` in $A[p]$. Then, p reads the rest of the elements of this array in order to detect conflicts with other operations that access x.

Whenever a conflict is detected between *op* and some other operation *op*$'$ simulated by some process q, while $'$ is accessing x for *op*, the algorithm ensures that either *op* will temporarily stop executing its current simulation phase and continue by helping *op*$'$, or *op*$'$ will be notified to restart its current simulation phase, in order to ensure consistency.

The algorithm uses the process ids of the processes that initiated two operations that conflict to determine which process has the higher priority. More specifically, assume that an operation *op* has detected a conflict with an operation *op*$'$. If the id of the owner of *op* is greater than the id of the owner of *op*$'$, then *op* temporarily stops executing its own program and continues by helping *op*$'$. Otherwise, *op* ensures that *op*$'$ will eventually see that there was a conflict and it will restart.

In the second case, the algorithm should: i) provide to *op* a method to inform *op*$'$ that it has to restart, and ii) ensure that *op*$'$ discovers that it should restart. These problems are solved as follows. An `LL/SC` object called *status* is maintained in the `oprec` of *op*, which takes the values *simulating*, *restart*, *modifying*, and *done*, depending on whether the process is currently simulating its own operation, it has been notified to restart or helping some other operation with which it conflicts, it is in its modification phase, or it has completed the execution of its operation, respectively. The *status* of *op*$'$ is initially set to *simulating*, when its `oprec` is initialized. Whenever *op* wants to notify *op*$'$ that it has to restart, it changes the *status* field in the `oprec` of *op*$'$ from *simulating* to *restart*. Moreover, to ensure that starvation of an announced operation is avoided process p that has initiated *op* should continue helping the operations that it notified to restart during *op*'s execution, after it has been completed and before it responds.

Also, we conjecture that if new entry points can be created as the data structure grows and they are unbounded, then there is no wait-free universal construction that is partial disjoint-access parallel.

Observe that a process may create a new entry point by storing locally a pointer to the last data item accessed by its last operation and use this locally stored pointer as the entry point for its next operation (or for any future operations) in order to speed up its execution.

3 Future collaboration with the host institution and Foreseen publications

To finalize the work and discuss new ideas, we have already planned a visit of Panagiota Fatourou (for one week) and of Eleftherios Kosmas (for one month) at the University of Bordeaux 1 in September 2012. We expect to submit a paper containing these results in one of the main distributed and/or parallel computing conferences. In particular we consider one of the following conferences: PODC13, SPAA 2013 or DISC 2013. During the visit of Alessia Milani to FORTH ICS, we worked on the journal version of our previous publication [2], improving the step complexity of the algorithm presented there and the lower bound. We plan to submit this work to a prestigious journal, such as Distributed Computing or Journal of Parallel and Distributed Computing.

References

[1] V. Bushkov, R. Guerraoui, and M. Kapalka. On the liveness of transactional memory. Technical Report EPFL-REPORT-170486, École Polytechnique Fédérale de Lausanne, 2011.

[2] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *PODC*, pages 115–124, 2012.

[3] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[4] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[5] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, Jul 1990.

[6] J.-T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber. Robustm: A robust software transactional system. In S. D. *et al.*, editor, *Stabilization, Safety, and Security in Distributed Systems, Proceedings*, volume 6366 of *Lecture Notes in Computer Science*. Springer, 2010.