Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom

Faith Ellen[*]
University of Toronto
faith@cs.toronto.edu

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

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

Alessia Milani[§]
University of Bordeaux
milani@labri.fr

Corentin Travers[§]
University of Bordeaux
travers@labri.fr

ABSTRACT

Disjoint-access parallelism and wait-freedom are two desirable properties for implementations of concurrent objects. *Disjoint-access parallelism* guarantees that processes operating on different parts of an implemented object do not interfere with each other by accessing common base objects. Thus, disjoint-access parallel algorithms allow for increased parallelism. *Wait-freedom* guarantees progress for each nonfaulty process, even when other processes run at arbitrary speeds or crash.

A *universal construction* provides a general mechanism for obtaining a concurrent implementation of any object from its sequential code. We identify a natural property of universal constructions and prove that there is no universal construction (with this property) that ensures both disjoint-access parallelism and wait-freedom. This impossibility result also holds for transactional memory implementations that require a process to re-execute its transaction if it has been aborted and guarantee each transaction is aborted only a finite number of times.

Our proof is obtained by considering a dynamic object that can grow arbitrarily large during an execution. In contrast, we present a universal construction which produces concurrent implementations that are both wait-free and disjoint-access parallel, when applied to objects that have a bound on the number of data items accessed by each operation they support.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed data structures; D.1.3 [**Concurrent Programming**]: Distributed programming

General Terms

Algorithms, Theory

Keywords

disjoint-access parallelism, impossibility result, universal construction, wait-freedom

1. INTRODUCTION

Due to the recent proliferation of multicore machines, simplifying concurrent programming has become a necessity, to exploit their computational power. A *universal construction* [21] is a methodology for automatically executing pieces of sequential code in a concurrent environment, while ensuring correctness. Thus, universal constructions provide functionality similar to Transactional Memory (TM) [23]. In particular, 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.

Many existing universal constructions [1, 12, 16, 17, 20, 21] restrict parallelism by executing each of the desired operations one after the other. 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.

[*]Supported in part by the Natural Science and Engineering Research Council of Canada. Some of the work was done while visiting Labri CNRS-UMR 5800, Bordeaux, France.

[†]Supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639), Hi-PEAC2 (FP7-ICT-217068), and ENCORE (FP7-ICT-248647) projects.

[‡]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.

[§]Supported in part by the ANR project DISPLEXITY.

PODC'12, July 16–18, 2012, Madeira, Portugal.

In this paper, we present both positive and negative results. We first identify a natural property of universal constructions and prove that designing universal constructions (with this property) which ensure both disjoint access parallelism and wait-freedom is not possible. We prove this impossibility result by considering a dynamic data structure that can grow arbitrarily large during an execution. Specifically, we consider a singly-linked unsorted list of integers that supports the operations APPEND(L, x), which appends x to the end of the list L, and SEARCH(L, x), which searches the list L for x starting from the first element of the list. We show 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.

Since the publication of the original definition of disjoint-access parallelism [25], many variants have been proposed [2, 9, 19]. These definitions are usually stated in terms of a conflict graph. A *conflict graph* is a graph whose nodes is a set of operations in an execution. An edge exists between each pair of operations that conflict. Two operations *conflict* if they access the same data item. A *data item* is a piece of the sequential data structure that is being simulated. For instance, in the linked list implementation discussed above, a data item may be a list node or a pointer to the first or last node of the list. In a variant of this definition, an edge between conflicting operations exists only if they are concurrent. Two processes *contend* on a base object, if they both access this base object and one of these accesses is a *non-trivial* operation (i.e., it may modify the state of the object). In a disjoint-access parallel implementation, two processes performing operations op and op' can contend on the same base object only if the conflict graph of the minimal execution interval that contains both op and op' satisfies a certain property. Different variants of disjoint-access parallelism use different properties to restrict access to a base object by two processes performing operations. Note that any data structure in which all operations access a common data item, for example, the root of a tree, is trivially disjoint access parallel under all these definitions.

For the proof of the impossibility result, we introduce *feeble disjoint-access parallelism*, which is weaker than all existing disjoint-access parallelism definitions. Thus, our impossibility result still holds if we replace our disjoint-access parallelism definition with any existing definition of disjoint-access parallelism.

Next, we show how this impossibility result can be circumvented, by restricting attention to data structures whose operations can each only access a bounded number of different data items. Specifically, there is a constant b such that any operation accesses at most b different data items when it is applied sequentially to the data structure, starting from any (legal) state. 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 [24] and satisfy a much stronger disjoint-access parallelism property than we used to prove the impossibility result.

Disjoint-access parallelism and its variants were originally formalized in the context of fixed size data structures, or when the data items that each operation accesses are known when the operation starts its execution. Dealing with these cases is much simpler than considering an arbitrary dynamic data structure where the set of data items accessed by an operation may depend on the operations that have been previously executed and on the operations that are performed concurrently.

The universal construction presented in this paper is the first that provably ensures both wait-freedom and disjoint-access parallelism for dynamic data structures in which each operation accesses a bounded number of data items. For other dynamic data structures, our universal construction still ensures linearizability and disjoint-access parallelism. Instead of wait-freedom, it ensures that progress is *non-blocking*. This guarantees that, in every execution, from every (legal) state, *some* process finishes its operation within a finite number of steps.

2. RELATED WORK

Some impossibility results, related to ours, have been provided for transactional memory algorithms. Transactional Memory (TM) [23] is a mechanism that allows a programmer of a sequential program to identify those parts of the sequential code that require synchronization as *transactions*. Thus, a transaction includes a sequence of operations on data items. When the transaction is being executed in a concurrent environment, these data items can be accessed by several processes simultaneously. If the transaction commits, all its changes become visible to other transactions and they appear as if they all take place at one point in time during the execution of the transaction. Otherwise, the transaction can abort and none of its changes are applied to the data items.

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 [11, 31] in transactional memory require that the number of times each transaction aborts is finite. This property is similar to the *wait-freedom* property for universal constructions. In a recent paper [11], this property is called *local progress*. Our impossibility result applies to transactional memory algorithms that satisfy this progress property. Disjoint-access parallelism is defined for transactions in the same way as for universal constructions.

Strict disjoint-access parallelism [19] requires that an edge exists between two operations (or transactions) in the conflict graph of the minimal execution interval that contains both operations (transactions) if the processes performing these operations (transactions) contend on a base object. A TM algorithm is *obstruction-free* if a transaction can be aborted only when contention is encountered during the course of its execution. In [19], Guerraoui and Kapalka proved that no obstruction-free TM can be strictly disjoint access parallel. Obstruction-freedom is a weaker progress property than wait-freedom, so their impossibility result also applies to wait-free implementations (or implementations

that ensure local progress). However, it only applies to this strict variant of disjoint-access parallelism, while we consider a much weaker disjoint-access parallelism definition. It is worth-pointing out that several obstruction-free TM algorithms [18, 22, 26, 29] satisfy a weaker version of disjoint-access parallelism than this strict variant. It is unclear whether helping, which is the major technique for achieving strong progress guarantees, can be (easily) achieved assuming strict disjoint-access parallelism. For instance, consider a scenario where transaction T_1 accesses data items x and y, transaction T_2 accesses x, and T_3 accesses y. Since T_2 and T_3 access disjoint data items, strict disjoint-access parallelism says that they cannot contend on any common base objects. In particular, this limits the help that each of them can provide to T_1.

Bushkov *et al.* [11] prove that no TM algorithm (whether or not it is disjoint-access parallel) can ensure local progress. However, they prove this impossibility result under the assumption that the TM algorithm does not have access to the code of each transaction (and, as mentioned in their introduction, their impossibility result does not hold without this restriction). In their model, the TM algorithm allows the external environment to invoke actions for reading a data item, writing a data item, starting a transaction, and trying to commit or abort it. The TM algorithm is only aware of the sequence of invocations that have been performed. Thus, a transaction can be helped only after the TM algorithm knows the entire set of data items that the transaction should modify. However, there are TM algorithms that do allow threads to have access to the code of transactions. For instance, RobuSTM [31] is a TM algorithm in which the code of a transaction is made available to threads so that they can help one another to ensure strong progress guarantees.

Proving impossibility results in a model in which the TM algorithm does not have access to the code of transactions is usually done by considering certain high-level histories that contain only invocations and responses of high-level operations on data items (and not on the base objects that are used to implement these data items in a concurrent environment). Our model gives the universal construction access to the code of an invoked operation. Consequently, to prove our impossibility result we had to work with low-level histories, containing steps on base objects, which is technically more difficult.

Attiya *et al.* [9] proved that there is no disjoint-access parallel TM algorithm where read-only transactions are wait-free and *invisible* (i.e., they do not apply non-trivial operations on base objects). This impossibility result is proved for the variant of disjoint-access parallelism where processes executing two operations (transactions) concurrently contend on a base object only if there is a path between the two operations (transactions) in the conflict graph. We prove our lower bound for a weaker definition of disjoint-access parallelism and it applies even for implementations with visible reads. We remark that the impossibility result in [9] does not contradict our algorithm, since our implementation employs *visible* reads.

In [27], the concept of *MV-permissiveness* was introduced. A TM algorithm satisfies this property if a transaction aborts only when it is an update transaction that conflicts with another update transaction. An *update transaction* contains updates to data items. The paper [27] proved that no transactional memory algorithm satisfies both disjoint access parallelism (specifically, the variant of disjoint-access parallelism presented in [9]) and MV-permissiveness. However, the paper assumes that the TM algorithm does not have access to the code of transactions and is based on the requirement that the code for creating, reading, or writing data items terminates within a finite number of steps. This lower bound can be beaten if this requirement is violated. Attiya and Hillel [8] presented a strict disjoint-access parallel lock-based TM algorithm that satisfies MV-permissiveness.

More constraining versions of disjoint-access parallelism are used when designing algorithms [5, 6, 25]. Specifically, two operations are allowed to access the same base object if they are connected by a path of length at most d in the conflict graph [2, 5, 6]. This version of disjoint-access parallelism is known as the *d-local contention property* [2, 5, 6]. The first wait-free disjoint-access parallel implementations [25, 30] had $O(n)$-local contention, where n is the number of processes in the system, and assumed that each operation accesses a fixed set of data items. Afek *et al.* [2] presented a wait-free, disjoint-access parallel universal construction that has $O(k + log^* n)$-local contention, provided that each operation accesses at most k pre-determined memory locations. It relies heavily on knowledge of k. This work extends the work of Attiya and Dagan [5], who considered operations on pairs of locations, i.e. where $k = 2$. Afek *et al.* [2] leave as an open question the problem of finding highly concurrent wait-free implementations of data structures that support operations with no bounds on the number of data items they access. In this paper, we prove that, in general, there are no solutions unless we relax some of these properties.

Attiya and Hillel [7] provide a k-local non-blocking implementation of k-read-modify-write objects. The algorithm assumes that double-compare-and-swap (`DCAS`) primitives are available. A `DCAS` atomically executes `CAS` on two memory words. Combining the algorithm in [7] and the non-blocking implementation of `DCAS` by Attiya and Dagan [5] results in a $O(k + log^* n)$-local non-blocking implementation of a k-read-modify-write object that only relies on single-word `CAS` primitives. Their algorithm can be adapted to work for operations whose data set is defined on the fly, but it only ensures that progress is non-blocking.

A number of wait-free universal constructions [1, 16, 17, 20, 21] work by copying the entire data structure locally, applying the active operations sequentially on their local copy, and then changing a shared pointer to point to this copy. The resulting algorithms are not disjoint access parallel, unless vacuously so.

Anderson and Moir [3] show how to implement a k-word atomic `CAS` using `LL/SC`. To ensure wait-freedom, a process may help other processes after its operation has been completed, as well as during its execution. They employ their k-word `CAS` implementation to get a universal construction that produces wait-free implementations of multi-object operations. Both the k-word `CAS` implementation and the universal construction allow operations on different data items to proceed in parallel. However, they are not disjoint-access parallel, because some operations contend on the same base objects even if there are no (direct or transitive) conflicts between them. The helping technique that is employed by our algorithm combines and extends the helping techniques

presented in [3] to achieve both wait-freedom and disjoint-access parallelism.

Anderson and Moir [4] presented another universal construction that uses indirection to avoid copying the entire data structure. They store the data structure in an array which is divided into a set of consecutive data blocks. Those blocks are addressed by a set of pointers, all stored in one LL/SC object. An adaptive version of this algorithm is presented in [16]. An algorithm is *adaptive* if its step complexity depends on the maximum number of active processes at each point in time, rather than on the total number n of processes in the system. Neither of these universal constructions is disjoint-access parallel.

Barnes [10] presented a disjoint-access parallel universal construction, but the algorithms that result from this universal construction are only non-blocking. In Barnes' algorithm, a process p executing an operation *op* first simulates the execution of *op* locally, using a local dictionary where it stores the data items accessed during the simulation of *op* and their new values. Once p completes the local simulation of *op*, it tries to lock the data items stored in its dictionary. The data items are locked in a specific order to avoid deadlocks. Then, p applies the modifications of *op* to shared memory and releases the locks. A process that requires a lock which is not free, releases the locks it holds, helps the process that owns the lock to finish the operation it executes, and then re-starts its execution. To enable this helping mechanism, a process shares its dictionary immediately prior to its locking phase. The lock-free TM algorithm presented in [18] works in a similar way.

As in Barnes' algorithm, a process executing an operation *op* in our algorithm, first locally simulates *op* using a local dictionary, and then it tries to apply the changes. However, in our algorithm, a conflict between two operations can be detected during the simulation phase, so helping may occur at an earlier stage of *op*'s execution. More advanced helping techniques are required to ensure both wait-freedom and disjoint-access parallelism.

Chuong *et al.* [12] presented a wait-free version of Barnes' algorithm that is not disjoint-access parallel and applies operations to the data structure one at a time. Their algorithm is *transaction-friendly*, i.e., it allows operations to be aborted. Helping in this algorithm is simpler than in our algorithm. Moreover, the conflict detection and resolution mechanisms employed by our algorithm are more advanced to ensure disjoint-access parallelism. The presentation of the pseudocode of our algorithm follows [12].

The first software transactional memory algorithm [28] was disjoint-access parallel, but it is only non-blocking and is restricted to transactions that access a pre-determined set of memory locations. There are other TM algorithms [14, 18, 22, 26, 29] without this restriction that are disjoint-access parallel. However, all of them satisfy weaker progress properties than wait-freedom. TL [14] ensures strict disjoint access parallelism, but it is blocking.

A hybrid approach between transactional memory and universal constructions has been presented by Crain *et al.* [13]. Their universal construction takes, as input, sequential code that has been appropriately annotated for processing by a TM algorithm. Each transaction is repeatedly invoked until it commits. They use a linked list to store all committed transactions. A process helping a transaction to complete scans the list to determine whether the transaction has already completed. Thus, their implementation is not disjoint-access parallel. It also assumes that no failures occur.

3. 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 it supports. As an example, we will consider an unsorted singly-linked list of integers that supports the operations Append(v), which appends the element v to the end of the list (by accessing a pointer *end* that points to the last element in the list, appending a node containing v to that element, and updating the pointer to point to the newly appended node), and Search(v), which searches the list for v starting from the first element of the list.

A *data item* is a piece of the representation of an object implemented by the data structure. In our example, the data items are the nodes of the singly-linked list and the pointers $first$ and $last$ that point to the first and the last element of the list, respectively. 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 example, the pointers $first$ and $last$ 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 α. For example, the set of nodes accessed by an instance of Search depends on the sequence of nodes that have been previously appended to the list.

An operation of a data structure is *value oblivious* if, in every (sequential) execution, the set of data items that each instance of this operation accesses in any sequence of consecutive instances of this operation does not depend on the values of the input parameters of these instances. In our example, Append is a value oblivious operation, but Search is not.

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* provides a general mechanism to automatically execute pieces of sequential code in a concurrent environment. It 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*. To avoid ambiguities and to simplify the exposition, we require that all data items in the sequential code are only accessed via the instruction CreateDI, ReadDI, and WriteDI, which create a new data item, read (any part of) the data item, and write to (any part of) the data item, respectively.

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 α is *indistinguishable* from another execution α' for some processes, if each of these processes takes the same steps in α and α', and each of these steps has the same response in α and α'. 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* [24] 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* [21] 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.

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. We are now ready to present the definition of disjoint-access parallelism that we use to prove our impossibility result. It is weaker than all the variants discussed in Section 2.

Definition 1. (**Feeble Disjoint-Access Parallelism**). *An implementation resulting from a universal construction applied to a (sequential) data structure is* feebly disjoint-access parallel *if, for every solo execution α_1 of an operation op_1 and every solo execution α_2 of an operation op_2, both starting from the same quiescent configuration C, if the sequential code of op_1 and op_2 access disjoint sets of data items when each is executed starting from the state of the data structure represented by configuration C, then α_1 and α_2 contend on no base objects. A universal construction is* feebly disjoint-access parallel *if all implementations resulting from it are feebly disjoint-access parallel.*

We continue with definitions that are needed to define the version of disjoint-access parallelism ensured by our algorithm. 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. For $i \leq k < j$, let S_k denote the state of the data structure which results from applying each operation linearized in α 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 $DS(op, \alpha) \cap DS(op', \alpha) \neq \emptyset$. The following variant of disjoint-access parallelism is ensured by our algorithm.

Definition 2. (**Disjoint-Access Parallelism**). *An implementation resulting from a universal construction applied to a (sequential) data structure is* disjoint-access parallel *if, for every execution containing a process executing* Perform*(op_1) and a process executing* Perform*(op_2) that contend on some base object, there is a path between op_1 and op_2 in the conflict graph of the minimal execution interval containing op_1 and op_2.*

The original definition of disjoint-access parallelism in [25] differs from Definition 2 in that it does not allow two operations op_1 and op_2 to read the same base object even if there is no path between op_1 and op_2 in the conflict graph of the minimal execution interval that contains them. T Also, that definition imposes a bound on the step complexity of disjoint-access parallel algorithms. Our definition is a slightly stronger version of the disjoint-access parallel variant defined in [9] in the context of transactional memory. This definition allows two operations to contend, (but not *concurrently* contend) on the same base object if there is no path connecting them in the conflict graph. This definition makes the lower bound proved there stronger, whereas our definition makes the design of an algorithm (which is our goal) more difficult. Our definition is obviously weaker than strict disjoint-access parallelism [19], since our definition allows two processes to contend even if the data sets of the operations they are executing are disjoint.

4. IMPOSSIBILITY RESULT

To prove the impossibility of a wait-free universal construction with feeble disjoint-access parallelism, we consider an implementation resulting from the application of an arbitrary feebly disjoint-access parallel universal construction to the singly-linked list discussed in Section 3. We show that there is an execution in which an instance of SEARCH does not terminate. The idea is that, as the process p performing this instance proceeds through the list, another process, q, is continually appending new elements with different values. If q performs each instance of APPEND before p gets too close to the end of the list, disjoint-access parallelism prevents q from helping p. This is because q's knowledge is consistent with the possibility that p's instance of SEARCH could terminate successfully before it accesses a data item accessed by q's current instance of APPEND. Also, process p cannot determine which nodes were appended by process q after it started the SEARCH. The proof relies on the following natural assumption about universal constructions. Roughly speaking, it formalizes that the operations of the concurrent implementation resulting from applying a universal construction to a sequential data structure should simulate the behavior of the operations of the sequential data structure.

ASSUMPTION 3 (VALUE-OBLIVIOUSNESS ASSUMPTION). *If an operation of a data structure is value oblivious, then, in any implementation resulting from the application of a universal construction to this data structure, the sets of base objects read from and written to during any solo execution of a sequence of consecutive instances of this operation starting from a quiescent configuration do not depend on the values of the input parameters.*

We consider executions of the implementation of a singly-linked list L in which process p performs a single instance of SEARCH$(L, 0)$ and process q performs instances of APPEND(L, i), for $i \geq 1$, and possibly one instance of APPEND$(L, 0)$. We may assume the implementation is deterministic: If it is randomized, we fix a sequence of coin tosses for each process and only consider executions using these coin tosses.

Let C_0 be the initial configuration in which L is empty. Let α denote the infinite solo execution by q starting from C_0 in which q performs APPEND(L, i) for all positive integers i, in increasing order. For $i \geq 1$, let C_i be the configuration obtained when process q performs APPEND(L, i) starting from configuration C_{i-1}. Let α_i denote the sequence of steps performed in this execution. Let $B(i)$ denote the set of base objects written to by the steps in α_i and let $A(i)$ denote the set of base objects these steps read from but do not write to. Notice that the sets $A(i)$ and $B(i)$ partition the set of base objects accessed in α_i. In configuration C_i, the list L consists of i nodes, with values $1, \ldots, i$ in increasing order.

For $1 < j \leq i$, let C_i^j be the configuration obtained from configuration C_0 when process q performs the first i operations of execution α, except that the j'th operation, APPEND(L, j), is replaced by APPEND$(L, 0)$; namely, when q performs APPEND$(L, 1)$, ..., APPEND$(L, j-1)$, APPEND$(L, 0)$, APPEND$(L, j+1)$, ..., APPEND(L, i). Since APPEND is value oblivious, the same set of base objects are written to during the executions leading to configurations C_i and C_i^j. Only base objects in $\cup\{B(k) \mid j \leq k \leq i\}$ can have different values in C_i and C_i^j.

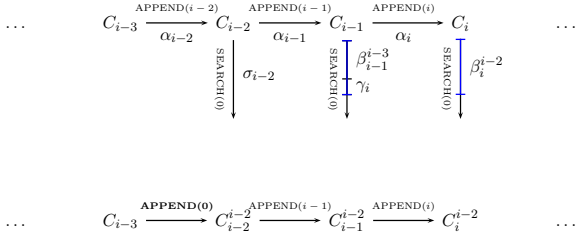

Figure 1: Configurations and Sequences of Steps used in the Proof

For $i \geq 3$, let σ_i be the steps of the solo execution of SEARCH$(L, 0)$ by p starting from configuration C_i. For $1 < j \leq i$, let β_i^j be the longest prefix of σ_i in which p does not access any base object in $\cup\{B(k) \mid k \geq j\}$ and does not write to any base object in $\cup\{A(k) \mid k \geq j\}$

LEMMA 4. *For $i \geq 3$ and $1 < j \leq i$, $\beta_i^j = \beta_{i+1}^j$ and β_{i+1}^{i-1} is a prefix of β_{i+2}^i.*

PROOF. Only base objects in $B(i+1)$ have different values in configurations C_i and C_{i+1}. Since β_i^j and β_{i+1}^j do not access any base objects in $B(i+1)$, it follows from their definitions that $\beta_i^j = \beta_{i+1}^j$. In particular, $\beta_{i+2}^i = \beta_{i+1}^i$, which, by definition contains β_{i+1}^{i-1} as a prefix. □

For $i \geq 3$, let γ_{i+2} be the (possibly empty) suffix of β_{i+2}^i such that $\beta_{i+1}^{i-1}\gamma_{i+2} = \beta_{i+2}^i$. Figure 1 illustrates these definitions.

Let $\alpha' = \alpha_1\alpha_2\alpha_3\alpha_4\beta_4^2\alpha_5\gamma_5\alpha_6\gamma_6\cdots$. We show that this infinite sequence of steps gives rise to an infinite valid execution starting from C_0 in which there is an instance of SEARCH$(L, 0)$ that never terminates. The first steps of this execution are illustrated in Figure 2.

Since β_4^2 does not write to any base objects accessed in $\alpha_2\alpha_3\cdots$ and, for $i \geq 4$, $\beta_{i+1}^{i-1} = \beta_i^{i-2}\gamma_{i+1}$ does not write to any base object accessed in $\alpha_{i-1}\alpha_i\cdots$, the executions arising from α and α' starting from C_0 are indistinguishable to process q. Furthermore, since β_{i+1}^{i-1} and, hence, γ_{i+1} does not access any base object written to by $\alpha_{i-1}\alpha_i\cdots$, it follows that $\alpha_1\alpha_2\alpha_3\alpha_4\beta_4^2\alpha_5\gamma_5\cdots\alpha_j\gamma_j$ and $\alpha_1\alpha_2\alpha_3\alpha_4\cdots\alpha_j\beta_j^{j-2}$ are indistinguishable to process p for all $j \geq 4$. Thus α' is a valid execution.

Next, for each $i \geq 4$, we prove that there exists $j > i$ such that γ_j is nonempty. By the value obliviousness assumption, only base objects in $B(i-2) \cup B(i-1) \cup B(i)$ can have different values in C_i and C_i^{i-2}. Since β_i^{i-2} does not access any of these base objects, β_i^{i-2} is also a prefix of SEARCH$(L, 0)$ starting from C_i^{i-2}. Since SEARCH$(L, 0)$ starting from C_i^{i-2} is successful, but starting from C_i is unsuccessful, SEARCH$(L, 0)$ is not completed after β_i^{i-2}. Therefore β_i^{i-2} is a proper prefix of σ_i. Let b be the base object accessed in the first step following β_i^{i-2} in σ_i. For $j \geq i+1$, only base objects in $\cup\{B(k) \mid i+1 \leq k \leq j\}$ can have different values in C_i and C_j. Therefore the first step following β_i^{i-2} in σ_j is the same as the first step following β_i^{i-2} in σ_i.

To obtain a contradiction, suppose that $\beta_i^{i-2} = \beta_{i+3}^{i+1}$. Then b is the base object accessed in the first step following

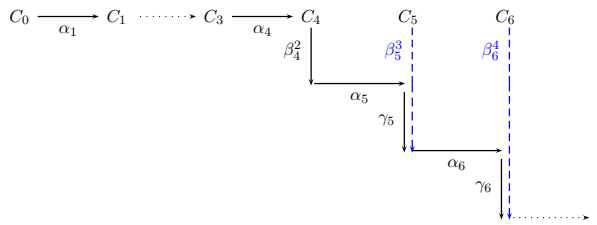

Figure 2: An Infinite Execution with a Non-terminating Search Operation

β_{i+3}^{i+1} in σ_{i+3}. By definition of β_{i+3}^{i+1}, there is some $\ell \geq i+1$ such that the first step following β_{i+3}^{i+1} in σ_{i+3} is either an access to $b \in B(\ell)$ or a write to $b \in A(\ell)$.

Let S denote the state of the data structure in configuration $C_{\ell-1}^{\ell-3}$. In state S, the list has $\ell-1$ nodes and the third last node has value 0. Thus, the set of data items accessed by SEARCH$(L,0)$ starting from state S consists of $L.first$ and the first $\ell-3$ nodes of the list. This is disjoint from the set of data items accessed by APPEND(L,ℓ) starting from state S, which consists of $L.last$, the last node of the list, and the newly appended node. Hence, by feeble disjoint access parallelism, the solo executions of APPEND(L,ℓ) and SEARCH$(L,0)$ starting from $C_{\ell-1}^{\ell-3}$ contend on no base objects.

By the value obliviousness assumption, $B(\ell)$ is the set of base objects written to in the solo execution of APPEND(L,ℓ) starting from $C_{\ell-1}^{\ell-3}$ and $A(\ell)$ is the set of base objects read from, but not written to in that execution.

By the value obliviousness assumption, only base objects in $B(\ell-3) \cup B(\ell-2) \cup B(\ell-1)$ can have different values in $C_{\ell-1}$ and $C_{\ell-1}^{\ell-3}$. Since β_i^{i-2} does not access any of these base objects, β_i^{i-2} is also a prefix of SEARCH$(L,0)$ starting from $C_{\ell-1}^{\ell-3}$ and the first step following β_i^{i-2} in this execution is the same as the first step following β_i^{i-2} in σ_i. Recall that this is either an access to $b \in B(\ell)$ or a write to $b \in A(\ell)$. Thus, the solo executions of APPEND(L,ℓ) and SEARCH$(L,0)$ starting from $C_{\ell-1}^{\ell-3}$ contend on b. This is a contradiction. Hence, $\beta_i^{i-2} \neq \beta_{i+3}^{i+1}$ and it follows that at least one of γ_{i+1}, γ_{i+2}, and γ_{i+3} is nonempty.

Therefore γ_j is nonempty for infinitely many numbers j and, in the infinite execution α', process p never completes its operation SEARCH$(L,0)$, despite taking an infinite number of steps. Hence, the implementation is not wait-free and we have proved the following result:

THEOREM 5. *No feebly disjoint-access parallel universal construction is wait-free.*

5. THE DAP-UC ALGORITHM

To execute an operation *op*, a process p locally simulates the execution of *op*'s instructions without modifying the shared representation of the simulated state. This part of the execution is the simulation phase of *op*. Specifically, each time p accesses a data item while simulating *op*, it stores a copy in a local dictionary. All subsequent accesses by p to this data item (during the same simulation phase of *op*) are performed on this local copy. Once all instructions of *op* have been locally simulated, *op* enters its modifying phase.

```
1   type varrec
2       value val
3       ptr to oprec A[1..n]
4   type statrec
5       {⟨simulating⟩,
6        ⟨restart, ptr to oprec restartedby⟩,
7        ⟨modifying, ptr to dictionary of dictrec changes,
8                                          value output⟩
9        ⟨done⟩
10      } status
11  type oprec
12      code  program
13      process id  owner
14      value input
15      value output
16      ptr to statrec status
17      ptr to oprec tohelp[1..n]
18  type dictrec
19      ptr to varrec key
20      value newval
```

Figure 3: Type definitions

At that time, one of the local dictionaries of the helpers of *op* becomes shared. All helpers of *op* then use this dictionary and apply the modifications listed in it. In this way, all helpers of *op* apply the same updates for *op*, and consistency is guaranteed.

The algorithm maintains a record for each data item x. The first time *op* accesses x, it makes an announcement by writing appropriate information in x's record. It also detects conflicts with other operations that are accessing x by reading this record. So, conflicts are detected without violating disjoint access parallelism. The algorithm uses a simple priority scheme, based on the process identifiers of the owners of the operations, to resolve conflicts among processes. When an operation *op* determines a conflict with an operation *op*$'$ of higher priority, *op* helps *op*$'$ to complete before it continues its execution. Otherwise, *op* causes *op*$'$ to restart and the owner of *op* will help *op*$'$ to complete once it finishes with the execution of *op*, before it starts the execution of a new operation. The algorithm also ensures that before *op*$'$ restarts its simulation phase, it will help *op* to complete. These actions guarantee that processes never starve.

We continue with the details of the algorithm. The algorithm maintains a record of type `oprec` (lines 11-17) 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* and initializing it appropriately (line 22). 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. We call p the *owner* of *op*. To execute *op*, p calls HELP (line 23). To ensure wait-freedom, before *op* returns, it helps all other operations listed in the *tohelp* array of its *oprec* record (lines 24-25). These are operations with which *op* had a conflict during the course of its execution, so disjoint-access parallelism is not violated. The algorithm also maintains a record of type `varrec` (lines 1-3) 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 `oprec` records of operations that are accessing x. This array is used by operations to

```
21  value PERFORM(prog, input) by process p:
22      opptr := pointer to a new oprec record
        opptr → program := prog, opptr → input := input, opptr → output := ⊥
        opptr → owner := p, opptr → status := simulating, opptr → tophelp[1..n] := [nil, ..., nil]
23      HELP(opptr)                                                          /* p helps its own operation */
24      for p' := 1 to n excluding p do                                      /* p helps operations that have been restarted by its operation op */
25          if (opptr → tohelp[p'] ≠ nil) then HELP(opptr → tohelp[p'])
26      return(opptr → output)

27  HELP(opptr) by process p:
28      opstatus := LL(opptr → status)
29      while (opstatus ≠ done)
30          if opstatus = ⟨restart, opptr'⟩ then                             /* op' has restarted op */
31              HELP(opptr')                                                 /* first help op' */
32              SC(opptr → status, ⟨simulating⟩)                             /* try to change the status of op back to simulating */
33              opstatus := LL(opptr → status)
34          if opstatus = ⟨simulating⟩ then                                  /* start a new simulation phase */
35              dict := pointer to a new empty dictionary of dictrec records
                                                                             /* to store the values of the data items */
36              ins := the first instruction in opptr → program
37              while ins ≠ return(v)                                        /* simulate instruction ins of op */
38                  if ins is (WRITEDI(x, v) or READDI(x)) and (there is no dictrec with key x in dict)
                    then                                                     /* first access of x by this attempt of op */
39                      ANNOUNCE(opptr, x)                                   /* announce that op is accessing x */
40                      CONFLICTS(opptr, x)                                  /* possibly, help or restart other operations accessing x */
41                      if ins = READDI(x) then val_x := x → val  else val_x := v   /* ins is a write to x of v */
42                      add new dictrec ⟨x, val_x⟩ to dict                   /* create a local copy of x */
43                  else if ins is CREATEDI() then
44                      x := pointer to a new varrec record
45                      x → A[1..n] := [nil, ..., nil]
46                      add new dictrec ⟨x, nil⟩ to dict
47                  else                                                     /* ins is WRITEDI(x, v) or READDI(x) and there is a dictrec with key x in dict */
                                                                             /* or ins is not a WRITEDI(), READDI() or CREATEDI() instruction */
48                      execute ins, using/changing the value in the appropriate entry of dict if necessary
49                  if ¬VL(opptr → status) then break                        /* end of the simulation of ins */
50                  ins := next instruction of opptr → program
                /* end while */
51              if ins is return(v) then                                     /* v may be empty */
52                  SC(opptr → status,⟨modifying, dict, v⟩)                  /* try to change status of op to modifying */
                                                                             /* successful iff simulation is over and status of op not changed since beginning of simulation */
53              opstatus := LL(opptr → status)
54          if opstatus = ⟨modifying, changes, out⟩ then
55              opptr → outputs := out
56              for each dictrec ⟨x, v⟩ in the dictionary pointed to by changes do
57                  LL(x → val)                                              /* try to make writes visible */
58                  if ¬VL(opptr → status) then return                       /* opptr → status = done */
59                  SC(x → val, v)
60                  LL(x → val)
61                  if ¬VL(opptr → status) then return                       /* opptr → status = done */
62                  SC(x → val, v)
                /* end for */
63              SC(opptr → status, done)
64              opstatus := LL(opptr → status)
        /* end while */
65  return

66  ANNOUNCE(opptr, x) by process p:
67      q := opptr → owner
68      LL(x → A[q])
69      if ¬ VL(opptr → status) then return
70      SC(x → A[q], opptr)
71      LL(x → A[q])
72      if ¬VL(opptr → status) then return
73      SC(x → A[q], opptr)
74      return

75  CONFLICTS(opptr, x) by process p:
76      for p' := 1 to n excluding opptr → owner do
77          opptr' := LL(x → A[p'])
78          if (opptr' ≠ nil) then                                           /* possible conflict between op and op' */
79              opstatus' := LL(oppptr' → status)
80              if ¬VL(opptr → status) then return
81              if (opstatus' = ⟨modifying, changes, output⟩)
82                  then HELP(opptr')
83              else if (opstatus' = ⟨simulating⟩) then
84                  if (opptr → owner < p') then
                                                                             /* op has higher priority than op', restart op' */
85                      opptr → tohelp[p'] := opptr'
86                      if ¬VL(opptr → status) then return
87                      SC(opptr' → status, ⟨restart, opptr⟩)
88                      if (LL(oppptr' → status) = ⟨modifying, changes, output⟩) then
                            HELP(opptr')
89                  else HELP(opptr')                                        /* opptr → owner > p' */
90      return
```

Figure 4: The code of Perform, Help, Announce, and Conflicts.

announce that they access x and to determine conflicts with other operations that are also accessing x.

The execution of *op* is done in a sequence of one or more *simulation phases* (lines 34-53) followed by a *modification phase* (lines 54-62). In a simulation phase, the instructions of *op* are read (lines 36, 37, and 50) and the execution of each one of them is simulated locally. The first time each process q helping *op* (including its owner) needs to access a data item (lines 38, 43), it creates a local copy of it in its (local) dictionary (lines 42, 46). All subsequent accesses by q to this data item (during the current simulation phase of *op*) are performed on this local copy (line 48). During the modification phase, q makes the updates of *op* visible by applying them to the shared memory (lines 56-62).

The *status* field of *op* determines the execution phase of *op*. It contains a pointer to a record of type `statrec` (lines 4-10) where the status of *op* is recorded. The status of *op* can be either *simulating*, indicating that *op* is in its simulation phase, *modifying*, if *op* is in its modifying phase, *done*, if the execution of *op* has been completed but *op* has not yet returned, or *restart*, if *op* has experienced a conflict and should re-execute its simulation phase from the beginning. Depending on which of these values *status* contains, it may additionally store another pointer or a value.

To ensure consistency, each time a data item x is accessed for the first time, q checks, before reading the value of x, whether *op* conflicts with other operations accessing x. This is done as follows: q announces *op* to x by storing a pointer *opr* to *op*'s `oprec` in $A[opr \rightarrow owner]$. This is performed by calling Announce (line 39). Announce first performs an `LL` on $var_x \rightarrow A[p]$ (line 68), where var_x is the `varrec` for x and $p = opr \rightarrow owner$. Then, it checks if the status of *op* (line 69) remains *simulating* and, if this is so, it performs an `SC` to store *op* in $var_x \rightarrow A[p]$ (line 70). These instructions are then executed one more time. This is needed because an obsolete helper of an operation, initiated by p before *op*, may successfully execute an `SC` on $var_x \rightarrow A[p]$ that stores a pointer to this operation's `oprec`. However, we prove [15] that this can happen only once, so executing the instructions on lines 68-70 twice is enough to ensure consistency.

After announcing *op* to var_x, q calls Conflicts (line 40) to detect conflicts with other operations that access x. In Conflicts, q reads the rest of the elements of $var_x \rightarrow A$ (lines 76-77). Whenever a conflict is detected (i.e., the condition of the `if` statement of line 78 evaluates to `true`) between *op* and some other operation op', Conflicts first checks if op' is in its modifying phase (line 82) and, if so, it helps op' to complete. In this way, it is ensured that, once an operation enters its modification phase, it will complete its operation successfully. Therefore, once the status of an operation becomes *modifying*, it will next become *done*, and then, henceforth, never change. If the status of op' is *simulating*, q determines which of *op* or op' has the higher priority (line 84). If op' has higher priority (line 89), then *op* helps op' by calling Help(op'). Otherwise, q first adds a pointer opr' to the `oprec` of op' into $opr \rightarrow tohelp$ (line 85), so that the owner of *op* will help op' to complete after *op* has completed. Then q attempts to notify op' to restart, using `SC` (line 87) to change the status of op' to *restart*. A pointer *opr* is also stored in the status field of op'. When op' restarts its simulation phase, it will help *op* to complete (lines 30-33), if *op* is still in its simulation phase, before it continues with the re-execution of the simulation phase of op'. This guarantees that *op* will not cause op' to restart again.

Recall that each helper q of *op* maintains a local dictionary. This dictionary contains an element of type `dictrec` (lines 18-20) for each data item that q accesses (while simulating *op*). A dictionary element corresponding to data item x consists of two fields, *key*, which is a pointer to var_x, and *newval*, which stores the value that *op* currently knows for x. Notice that only one helper of *op* will succeed in executing the `SC` on line 52, which changes the status of *op* to *modifying*. This helper records a pointer to the dictionary it maintains for *op*, as well as its output value, in *op*'s *status*, to make them public. During the modification phase, each helper q of *op* traverses this dictionary, which is recorded in the status of *op* (lines 54, 56). For each element in the dictionary, it tries to write the new value into the `varrec` of the corresponding data item (lines 57-59). This is performed twice to avoid problems with obsolete helpers in a similar way as in Announce.

Theorem 6. *The* DAP-UC *universal construction (Figures 3 and 4) produces disjoint-access parallel, wait-free, concurrent implementations when applied to objects that have a bound on the number of data items accessed by each operation they support.*

6. REFERENCES

[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, STOC '95, pages 538–547, New York, USA, 1995.

[2] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou. Disentangling multi-object operations (extended abstract). In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, PODC '97, pages 111–120, New York, USA, 1997.

[3] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 184–193, New York, USA, 1995.

[4] J. H. Anderson and M. Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999.

[5] H. Attiya and E. Dagan. Universal operations: unary versus binary. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 223–232, New York, USA, 1996.

[6] H. Attiya and E. Hillel. Built-in coloring for highly-concurrent doubly-linked lists. In *Proceedings of the 20th International Symposium on Distributed Computing*, DISC '06, volume 4167 of *Lecture Notes in Computer Science*, pages 31–45, Springer, 2006.

[7] H. Attiya and E. Hillel. Highly-concurrent multi-word synchronization. In *Proceedings of the 9th International Conference on Distributed Computing and Networking*, ICDCN'08, pages 112–123, Springer-Verlag, Berlin, Heidelberg, 2008.

[8] 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, Springer-Verlag, Berlin, Heidelberg, 2011.

[9] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 69–78, New York, USA, 2009.

[10] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[11] V. Bushkov, R. Guerraoui, and M. Kapalka. On the liveness of transactional memory. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing*, PODC '12, New York, USA, 2012, to appear.

[12] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 335–344, New York, USA, 2010.

[13] T. Crain, D. Imbs, and M. Raynal. Towards a universal construction for transaction-based multiprocess programs. In *Proceedings of the 13th International Conference on Distributed Computing and Networking*, ICDCN '12, pages 61–75, 2012.

[14] D. Dice and N. Shavit. What Really Makes Transactions Faster? In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT '06, Ottawa, Canada, 2006.

[15] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal Constructions that Ensure Disjoint-Access Parallelism and Wait-Freedom. Technical Report, `http://hal.inria.fr/hal-00697198/en/`, INRIA, 2012.

[16] P. Fatourou and N. D. Kallimanis. The redblue adaptive universal constructions. In *Proceedings of the 23rd international conference on Distributed computing*, DISC '09, pages 127–141, Berlin, Heidelberg, 2009.

[17] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the 23nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 325 – 334, San Jose, USA, 2011.

[18] K. Fraser. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[19] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 304–313, New York, USA, 2008.

[20] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, PPOPP '90, pages 197–206, New York, USA, 1990.

[21] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languanges and Systems*, 13:124–149, January 1991.

[22] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23:146–196, May 2005.

[23] 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, USA, 1993.

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

[25] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 151–160, New York, USA, 1994.

[26] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, DISC' 05, 2005. Also available as TR 868, University of Rochester Computer Science Dept., May2005.

[27] D. Pelerman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, PODC '10, pages 16–25, New York, USA, 2010.

[28] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, USA, 1995.

[29] F. Tabba, M. Moir, J. R. Goodman, A. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, August 2009.

[30] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, pages 212–222, New York, USA, 1992.

[31] J.-T. Wamhoff, T. Riegel, C. Fetzer, and P. Felber. RobuSTM: A robust software transactional system. In *Proceedings of the 12th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '10, volume 6366 of *Lecture Notes in Computer Science*, Springer-Verlag, 2010.