Consensus Numbers of Transactional Objects

Eric Ruppert

University of Toronto
ruppert@cs.utoronto.ca

Abstract. This paper describes the ability of asynchronous shared-memory distributed systems to solve the consensus problem in a wait-free manner if processes are permitted to perform transactions on the shared memory in a single atomic action. It will be shown that transactional memory is often extremely powerful, even if weak types of shared objects are used and the transactions are short. Suppose `T` is a type of shared object. For any positive integer m, the transactional type trans$(\texttt{T}, m)$ allows processes to perform up to m accesses to a collection of objects of type `T` in a transaction. The transaction may also include internal process actions that do not affect the shared memory. For any non-trivial type `T`, trans$(\texttt{T}, m)$ can solve consensus among $\Omega(2^{m/2})$ processes. A stronger lower bound of $\Omega(2^m)$ is given for a large class of objects that includes all non-trivial read-modify-write types `T`. If the type `T` is equipped with operations that allow processes to read the state of the object without altering the state, then trans$(\texttt{T}, 2)$ is capable of solving consensus among any number of processes. This paper also gives a consensus algorithm for n^{m-1} processes using trans$(n\texttt{-consensus}, m)$ and a consensus algorithm for any number of processes that uses trans$(\texttt{test\&set}, 3)$.

1 Introduction

It is often difficult to design fault-tolerant algorithms for asynchronous distributed systems and prove them correct. This difficulty arises from uncertainty about the behaviour of processes: they run at arbitrarily varying speeds and are subject to failures. One way to make the job of algorithm designers easier is to allow them to specify that certain groups of operations by one process are to be performed without interruptions by other processes [2, 3, 6, 7, 12, 15]. Here, a general model of transactional shared memory will be considered. A transaction is a programmer-specified block of an algorithm that the scheduler is required to treat as a single atomic action. Typically, distributed systems do not provide transactions as primitive operations, so it would be desirable to implement transactions in software from the more basic primitives provided. In this paper, it is shown that giving the programmer the ability to use transactions greatly increases the power of a shared-memory distributed system, where the power is measured by the system's ability to solve the consensus problem in a wait-free way. In particular, this means that one cannot hope, in general, to implement wait-free transactions in software from the corresponding primitive operations.

This paper considers asynchronous shared-memory systems. A scheduler is free to interleave the steps of processes in an arbitrary way, and processes are subject to halting failures. All algorithms are required to be *wait-free* [4]: non-faulty processes must correctly complete their executions even if other processes fail or run at varying speeds. Processes communicate by accessing shared data structures called objects. Read/write `registers`, `queues`, and `compare&swap` objects are examples of different types of shared objects. A type of shared object can be specified formally as an I/O automaton [11]. The object has a state, a set of permissible operations, and a set of possible responses to operations. An operation updates the state of the object and returns a response to the process that invoked the operation, in accordance with a transition function that is part of the object specification. All objects considered in this paper are deterministic.

For any object type $\mathtt{T}$ and any positive integer m, one can define the transactional type $\mathrm{trans}(\mathtt{T}, m)$, where m is an upper bound on the length of a transaction. It consists of a collection of objects of type $\mathtt{T}$, called *base objects*, indexed by the natural numbers. Although a potentially infinite number of base objects are permitted, the number of base objects actually used by the algorithms in this paper is polynomial in the number of processes. The type $\mathtt{T}$ is called the *base type* of the transactional type. An operation on the transactional object (called a *transaction*) can be thought of as a block of code where any execution of the code performs at most m operations on the base objects and performs no other shared-memory operations. More formally, a transaction is a collection of m (computable) functions $f_1, \ldots, f_m$, where f_i maps tuples of $i-1$ responses from objects of type $\mathtt{T}$ either to "nil" or to an operation on an object of type $\mathtt{T}$ and a natural number. Thus, $f_i(r_1, \ldots, r_{i-1})$ gives the operation to be performed and the index of the base object to be accessed during the ith shared-memory access of the transaction when the responses received from the first $i-1$ shared-memory accesses in the transaction are $r_1, \ldots, r_{i-1}$. If, during some execution of the transaction, the function f_i evaluates to "nil", this indicates that the transaction should terminate after the first $i-1$ shared-memory accesses.

The consensus problem has been a very useful tool for comparing the power of different shared-memory systems. In the consensus problem, each process begins with an input value and the processes must all output the same value. This common output value must be the input value of some process. Herlihy [4] showed that objects that solve the n-process consensus problem can be used, with `registers`, to implement any other object type in a system of n processes. Thus, the ability of a shared object type to solve the wait-free consensus problem is an important measure of the type's power to implement other types of shared data structures and to solve problems. This result led to the idea of classifying object types according to their consensus numbers [4, 8]. The *consensus number* of an object type $\mathtt{T}$, denoted $\mathrm{cons}(\mathtt{T})$, is the maximum number of processes that can solve consensus using objects of type $\mathtt{T}$ and `registers`, or infinity if no such maximum exists. In the latter case, the object type is called *universal*, since it can be used to implement any object in a system with any number of processes. This paper studies how much the consensus number of a type $\mathtt{T}$ increases when processes are permitted to perform transactions instead of individual operations.

1.1 Related Results

Afek, Merritt and Taubenfeld [1] defined another type of object, called a multi-object, where processes may perform a number of basic operations as a single atomic action. If $\mathtt{T}$ is an object type and m is a positive integer, the multi-object multi$(\mathtt{T}, m)$ consists of a collection of base objects of type $\mathtt{T}$, indexed by the natural numbers. [1] An operation on the multi-object is a set of up to m operations on the base objects. As is the case for transactional objects, a multi-object has a potentially infinite number of base objects, but algorithms designed for multi-objects typically use only a polynomial number of base objects. In contrast to the adaptive nature of transactional memory, the m accesses of a multi-object operation must be specified in advance. Thus none of the accesses to base objects may depend on the results from other accesses within the same operation on the multi-object. In addition, the m operations that make up an operation on the multi-object must be applied to distinct base objects. Although the latter restriction is not a part of the definition of transactional object types, all of the algorithms in this paper do perform their operations within a transaction on distinct base objects.

Afek, Merritt and Taubenfeld [1] determined cons(multi$(\mathtt{T}, m)$) for several base types $\mathtt{T}$. Ruppert [14] proved a lower bound on the consensus number of multi$(\mathtt{T}, m)$ that applies to any type $\mathtt{T}$ with cons$(\mathtt{T}) > 2$, and a stronger bound in the case of readable base types $\mathtt{T}$. These results are summarized in the fourth column of Table 1. Since multi$(\mathtt{T}, m)$ is a restricted form of trans$(\mathtt{T}, m)$, any lower bound on the consensus number of multi$(\mathtt{T}, m)$ is automatically a lower bound on the consensus number of trans$(\mathtt{T}, m)$. Jayanti and Khanna [9] considered a related kind of multi-object, where no bound was set on the number of base objects that could be accessed in an atomic action. They showed that the consensus number of such a multi-object is always either 1, 2 or infinity.

A number of researchers have studied transactional objects for particular base types. Most of the research has been focussed on implementing versions of transactional multi-objects from universal object types. One type of universal primitive object that has been considered is the `compare&swap` object. It is equipped with the operation *compare&swap*(old, new), which updates the state of the object to new, if and only if the current state of the object is old. Another type is the `LL/SC` object, which is equipped with two operations: the *load-linked* (LL) operation reads the value stored in the object, and the *store-conditional*(v) (SC) operation updates it to a new value v. However, a SC operation performed by a process succeeds if and only if the object has not been updated since the last LL by that process. Israeli and Rappoport [7] described a variety of implementations of versions of `LL/SC` and `compare&swap` objects where the SC and *compare&swap* operations are generalized to access several objects in a single atomic action. Shavit and Touitou described how to use `LL/SC` objects for a non-blocking implementation of atomic transactions on read/write `registers`. (A non-blocking implementation [5] has a weaker notion of fault-tolerance than

[1] The type multi$(\mathtt{T}, m)$ has previously been denoted $\mathtt{T}^m$ in the literature.

Base Type `T`	`T` readable?	cons(`T`)	cons(multi(`T`, m)) (for $m > 1$)		cons(trans(`T`, m)) (for $m > 1$)	
`trivial`	Yes	1	1		1	
`register`	Yes	1	$2m-2$	[4]	∞	Thm 3
`toggle`	Yes	1	∞	[14]	∞	
`test&set`	No	2	2	[1]	∞ $(m > 2)$	Prop. 8
`queue`	No	2	∞	[1]	∞	
any non-trivial `T`	Maybe	n	$\Omega(n\sqrt{m})$ $(n \geq 3)$	[14]	$\Omega(2^{m/2})$	Thm 9
n-`consensus`	No	n	$\Theta(n\sqrt{m})$ $(n \geq 3)$	[1]	$\Omega(n^{m-1})$	Prop. 4
any readable `T`	Yes	n	$\Omega(nm)$ $(n \geq 3)$	[14]	∞ (`T` non-trivial)	Thm 3
$\mathtt{T}_n$	Yes	$n \geq 3$	$\Theta(nm)$	[14]	∞	Thm 3
`toggle` $\vee$ $\mathtt{T}_n$	Yes	n	∞	[14]	∞	
any RMW `T`	Maybe	n	$\Omega(n\sqrt{m})$ $(n \geq 3)$	[14]	$\Omega(2^m)$ $(n \geq 2)$	Corol. 7

Table 1. Summary of results

wait-freedom: there can never be an infinite execution in which no operation is completed.) Moir [12] gave a similar wait-free implementation. Attiya and Dagan [3] discussed a non-blocking implementation of operations that can access two base objects in a single action, using `LL/SC` objects as their primitive. They also showed that *LL* and *SC* operations can be used to solve a problem more efficiently if processes may perform operations on more than one base object in a single atomic action.

1.2 New Results

An object type is called *trivial* if it can be simulated without using shared memory at all. For example, if the only operation permitted is a *read* operation, which simply returns the current state of the object, the object is trivial, since all *read* operations can be simulated by returning the initial state of the object as the response. An object type is called *readable* if it is possible to read the state of the object without altering the state. The read operation need not read the entire state in a single action: instead, it might be possible to read the state piece by piece, as in an array of `registers`, for example. Section 2 contains a proof that, for any non-trivial readable type `T`, trans(`T`, 2) is universal. Thus, in the case of readable objects, the transactional model is far more powerful than the multi-object model: a multi(`register`, m) object can only solve consensus among $2m-2$ processes [4], and there are readable types whose consensus numbers increase only by a factor of $\Theta(m)$ in a multi-object setting [14].

Section 3 studies a specific base type called n-`consensus`. The n-`consensus` object is equipped with an operation *propose*(v), which returns the first value proposed to the object. The object may be accessed at most n times and has consensus number n. Afek, Merritt and Taubenfeld showed that the consensus number of the multi-object multi(n-`consensus`, m) is $\Theta(n\sqrt{m})$. This result does not extend to transactional objects: Sect. 3 shows that cons(trans(n-**consensus**, m)) is at least n^{m-1}.

Section 4 shows that cons(trans($\mathtt{T}, m$)) $\geq 2^{m-1}$ for a very large class of base types $\mathtt{T}$, which includes all non-trivial read-modify-write (RMW) types [10]. Many commonly studied objects including `compare&swap` objects, `test&set` objects and `fetch&add` objects are RMW types. Section 4 concludes by considering the base type `test&set`, which is perhaps the simplest kind of RMW object. A `test&set` object has two states, 0 and 1. Its single operation, *test&set* sets the state to 1, and returns the old value of the state. This type has consensus number two [4]. In fact, even multi(`test&set`, m) has consensus number two, for any m [1]. However, trans(`test&set`, 3) is universal.

Section 5 gives a general lower bound of $2^{\lfloor (m-1)/2 \rfloor}$ on cons(trans($\mathtt{T}, m$)) for any non-trivial base type $\mathtt{T}$.

The results about consensus numbers of transactional objects are summarized in the last column of Table 1. The type $\mathtt{T}_n$ is a readable version of the n-`consensus` type, and the `toggle` type is a simple three-state readable object [14]. The $\vee$ operator used in the table takes two object types as operands. The type $\mathtt{T}_1 \vee \mathtt{T}_2$ describes an object that behaves either as an object of type $\mathtt{T}_1$ or as an object of type $\mathtt{T}_2$, depending on its initial state. Using $\mathtt{T}_1 \vee \mathtt{T}_2$ as the base type in a transactional setting effectively allows transactions that can access objects of both types. Thus, there is no real loss of generality in studying transactional objects that use a single base type instead of a collection of different base types.

2 Readable Base Objects

An object is readable if processes may read the object's state without altering it. It is not necessary that processes be able to read the entire state in a single operation; instead they may be able to read it "piecewise". Formally, a *readable* object O has a state set that is a Cartesian product $Q = \bigtimes_{k \in \Gamma} Q_k$, where Γ is an index set and Q_k is a set for each $k \in \Gamma$. The sets Γ and Q_k need not be finite. For each $k \in \Gamma$, processes may execute the operation *read*(O, k), which returns component k of the current state of O without changing the state of O. In addition to the *read* operations, the object may be equipped with an arbitrary set of other operations. These other operations are called *update* operations. It is assumed that each update operation can change only a finite number of components of the state.

Any object type with a *read* operation that returns the entire state of the object is readable; in this case, $|\Gamma| = 1$. An array of `registers` whose elements can be read, copied or swapped atomically is another example of a readable type.

For readable objects, it is possible to give a combinatorial characterization of the object types that are capable of solving consensus among n processes. A readable object is called *n-universal* [13] if a set of n processes can be partitioned into two non-empty teams and a single operation can be assigned to each process so that for any group of processes, if each process performs its own operation on an appropriately initialized object X of type T, then each process in the group could determine which team accessed X first, provided that it could see the final state of X. A more formal definition is given below.

Definition 1. *A readable type* T *is n-universal if there exist*
- *a state $q_0 \in Q$,*
- *a partition of the set of processes $\{P_1, \ldots, P_n\}$ into two non-empty teams A and B, and*
- *an update operation op_i, for $1 \leq i \leq n$,*

such that

for $1 \leq j \leq n$, $R_{A,j} \cap R_{B,j} = \emptyset$,

where $R_{A,j}$ is the set of pairs (r, q) for which there exist distinct process indices $i_1, \ldots, i_\alpha$ including j with $P_{i_1} \in A$ such that if $P_{i_1}, \ldots, P_{i_\alpha}$ each perform their operations (in that order) on an object of type T *that is initially in state q_0, P_j gets the result r, and the object ends in state q. The set $R_{B,j}$ is defined similarly.*

The property of being n-universal characterizes the readable object types that have consensus number at least n.

Theorem 2. [13] *A readable type can be used with* `registers` *to solve consensus among n processes if and only if it is n-universal.*

The universality of non-trivial readable types may now be proved. Recall that an object is trivial if it can be simulated without using shared memory.

Theorem 3. *For any non-trivial readable type* T, $\mathrm{cons}(\mathrm{trans}(\mathtt{T}, 2)) = \infty$.

Proof. Since T is non-trivial, there must be some update operation *op* that changes the state of the object from some state q to some other state r.

First it is shown that $\mathrm{trans}(\mathtt{T}, m)$ is, itself, a readable type. It is possible to read each of the base objects (possibly in a piecewise manner), so it is possible to read any component of the transactional object's state without altering the state. In addition, a transaction will satisfy the technical requirement of updating only a finite number of components of the state of the transactional object, since it accesses at most m base objects, and updates a finite number of components of the state of each of them.

By Theorem 2, it suffices to show that $\mathrm{trans}(\mathtt{T}, m)$ is n-universal (as specified in Definition 1) for every natural number n. The initial state q_0 of the transactional object has every base object in state q. Partition a collection of n processes into two teams $A = \{P_1\}$ and $B = \{P_2, \ldots, P_n\}$. Let the base objects of the transactional object be denoted by $O_1, O_2, \ldots$. The operation op_1 assigned to process P_1 is simply an application of the update operation *op* to O_1. For $i > 1$, assign to process P_i a transaction as its op_i: it first performs a *read* of O_1 and, if the state returned is q, it then applies *op* to O_i.

Consider any execution where some group of processes each perform their assigned operations. If the process P_1 on team A takes the first step, all of the objects $O_2, \ldots, O_n$ will always remain in state q. However, if a process P_i

on team B takes the first step, it will change the state of O_i to r, and O_i will remain in state r for the rest of the execution. Thus, one can determine whether a process on team A took the first step from the state of the transactional object by checking whether all of the objects $O_2, \ldots, O_n$ are in state q. The object trans($\mathtt{T}, 2$) is therefore n-universal for any n. $\square$

It follows from this theorem that a readable type `T` can be used to implement trans($\mathtt{T}, 2$) if and only if `T` is either trivial or universal. Furthermore, transactional objects can be much more powerful than ordinary multi-objects. For example, for any integer m greater than one, the multi-object multi($\mathtt{register}, m$), which has consensus number $2m - 2$ [4], is incapable of implementing even trans($\mathtt{register}, 2$).

3 Consensus Base Objects

This section considers transactional memory that uses base objects of the type n-`consensus` (defined in Sect. 1.2). The algorithm presented here shows that the ability to use transactions causes an exponential increase in the consensus number of this type. This contrasts with the multi-object setting, where Afek, Merritt and Taubenfeld showed that if the base objects of type `T` may be accessed by at most n processes, cons(multi($\mathtt{T}, m$)) is $O(n\sqrt{m})$.

The algorithm uses a collection of tree data structures. Processes access these trees by starting at a leaf and working towards the root. Each node contains an n-`consensus` object. The nodes at one level of the tree act as a filter to control access to the nodes at the next higher level. The first value proposed to the first tree's root becomes the output of all processes. However, since each base object in the transactional object can be accessed at most n times, information about the output value must be carefully distributed to all processes using a series of other trees that use similar filtering mechanisms to control access to the nodes. The algorithm given here will be adapted in Sections 4 and 5 to prove more general results.

Proposition 4. *The transactional object* trans(n-$\mathtt{consensus}, m$) *can be used to solve consensus among* n^{m-1} *processes.*

Proof. When $n = 1$, the result is trivial, so assume that $n \geq 2$. A consensus protocol for $N = n^{m-1}$ processes, $P_1, \ldots, P_N$, will be constructed using a transactional object of type trans(n-$\mathtt{consensus}, m$). Arrange the base objects into trees as follows. The first tree, T, is a complete n-ary tree of height $m-2$. Each of the remaining trees, T_k^j for $1 \leq k \leq n$ and $1 \leq j < n^{m-2}$, consists of a root with one child which is, itself, the root of a complete n-ary tree of height $m-3$. The tree T will be used to determine the output of the consensus protocol. The remaining trees will be used to distribute this information to all of the processes. A part of the data structure is shown in Fig. 1. The triangles represent complete n-ary trees of height $m-3$.

Divide the processes into n^{m-2} groups of size n. Associate each group with a different leaf of the tree T. Let G_k be the set of n^{m-2} processes that are

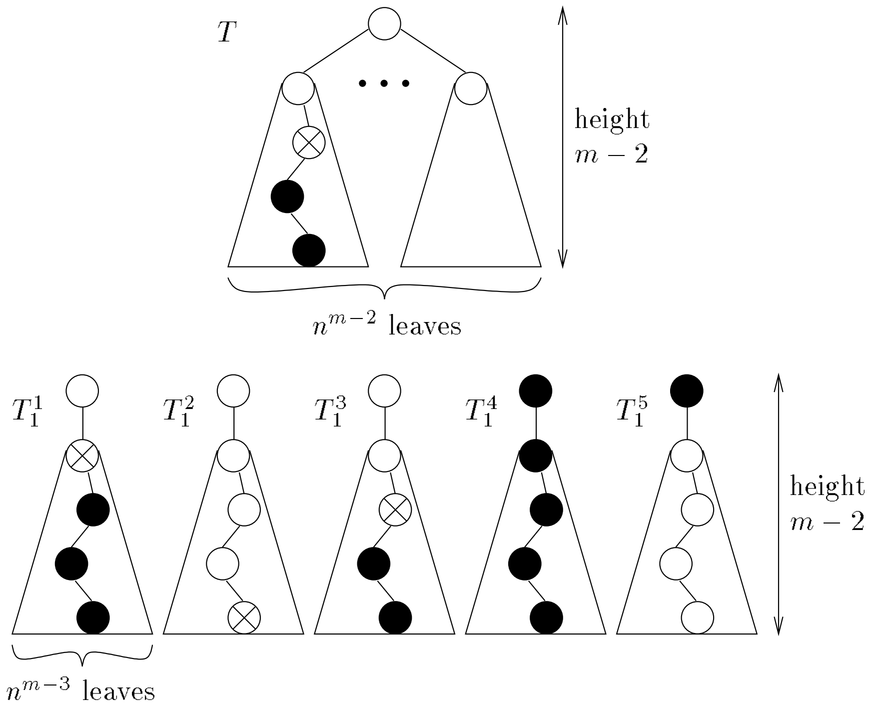

Fig. 1. Accesses to the transactional object by one process

associated with leaf descendants of the kth child of T's root. The trees T_k^j, for $1 \leq j < n^{m-2}$, will be used to distribute information to the processes in G_k. Associate each of the n-process groups in G_k with one of the n^{m-3} leaves of T_k^1. The group is also associated with the corresponding leaf in each of the trees T_k^j, for $2 \leq j < n^{m-2}$.

Process P_i in group G_k begins a transaction in which it will access base objects of the tree T. The process will access several nodes in the tree, each time proposing its own process identifier, i, and receiving a response. The process is said to "win" at a node if the response it receives is the value it proposed, and it is said to "lose" at the node if it receives some other response. First, it accesses the leaf of T with which its group of processes has been associated. If it wins at the leaf, process P_i goes on to access the parent of the leaf, again proposing the value i. It continues in this way, traversing the path from the leaf to the root until it loses. If the process loses at some node below the root, it ends its transaction. If it does reach the root, P_i proposes its own input value (instead of i) to the object located there. Let v be the value that the process P_i receives as a response from the root. The process then proposes the value v to the root of T_k^1, terminates its transaction, outputs v and halts. The access to the root of

T_k^1 has the effect of copying v into that object: any future access to that object will return the value v.

So far, only the processes that accessed the root of T know the outcome. However, this outcome has also been stored in the roots of the trees T_k^1. In an attempt to retrieve this information, each active process in group G_k performs its second transaction on T_k^1. It executes the same algorithm on this tree as it performed on T. (If the process does reach the root of T_k^1, it copies the output value into the root of T_k^2.) The process continues in this way, performing transactions on T_k^2, T_k^3, and so on, until it successfully reaches the root of a tree. Whenever a process accesses the root of a tree T_k^j, it immediately copies the value stored there to the root of the next tree, T_k^{j+1}, as part of the same transaction, outputs the value and then halts. All outputs of this protocol will be the input value of the first process to perform a transaction.

Figure 1 shows the accesses to the transactional object by some process of group G_1 in one execution. The solid circles represent nodes where the process wins. The circles marked with an "X" represent the nodes where it loses. In this example, the process performs five transactions. In the first, it attempts to gain access to the root of T by working its way up from a leaf, but fails. The process then tries unsuccessfully to gain access to the roots of T_1^1, T_1^2 and T_1^3 in its next three transactions. In the final transaction, it successfully reaches the root of T_1^4 and then accesses the root of T_1^5 directly.

It must be shown that the algorithm terminates. Suppose the process P_i attempts to access one of the trees T_k^j, but loses before reaching the root. Then some other process must have performed its transaction on the tree before P_i. The first such process will successfully reach the root and terminate. Thus, each time process P_i fails to access the root of a tree, some other process in G_k terminates. There are $n^{m-2} - 1$ processes in G_k that attempt to access the information from the trees in the set $\{T_k^j : 1 \leq j < n^{m-2}\}$, since one process in G_k gets the information directly from the root of T. Thus, process P_i must successfully access the root of one of the $n^{m-2} - 1$ trees T_k^j.

Since any path from a leaf to a root of one of the trees has $m - 1$ nodes, no transaction contains more than m operations on the shared memory. It must also be checked that no more than n processes access any base object, since the behaviour of the base objects is well-defined only under this restriction. For $1 \leq j < n^{m-2}$, at most one process will access the root of T_k^j during its $(j+1)$th transaction, since only one process will win at the child of the root. At most one process will access the root of T_k^1 during its first transaction, since only one of G_k's processes can reach the root of T. For $1 < j < n^{m-2}$, at most one process will access the root of T_k^j during its jth transaction, since only one process can reach the root of T_k^{j-1} in its jth transaction. Thus, at most two processes access the root of any tree T_k^j. The leaf of any tree is accessed only by the n processes in the group associated with that leaf. Any other internal node is accessed only by those processes that win at the children of the node, and there are at most n such processes. $\square$

4 Read-Modify-Write Base Types

Consider an object type whose set of possible states is Q. Any function $f : Q \to Q$ defines a *RMW* operation [10] that updates the state of the object by applying the function f and then returns the previous state of the object. If all operations on the object have this form, the object type is called a RMW type. This class of types includes many important types of objects, such as `compare&swap`, `test&set` and `fetch&add` objects.

To study the consensus power of transactional objects built from RMW types, it is helpful to use a special case of the consensus problem. The $\mathrm{team}(n_1, n_2)$ problem is defined to be a restricted version of the general consensus problem among $n_1 + n_2$ processes where the processes are divided (in advance) into two non-empty teams of sizes n_1 and n_2 and all processes on a team receive the same input value. A tournament algorithm was used to prove the following lemma, which says that this restricted version of the consensus problem is just as hard as the general consensus problem.

Lemma 5. [13] *Suppose objects of type* `T` *and* `registers` *can be used to solve the* $\mathrm{team}(n_1, n_2)$ *problem for some positive integers n_1 and n_2. Then objects of type* `T` *can be used, with* `registers`*, to solve consensus among $n_1 + n_2$ processes.*

Herlihy showed that any non-trivial RMW object type has consensus number at least two [4]. An algorithm similar to the one given in Proposition 4 for the case where $n = 2$ can be used to establish an exponential lower bound on the consensus number of transactional memory built from any non-trivial RMW object type. In fact, the lower bound applies to an even more general class of objects. It is applicable whenever two processes can each apply a single operation and immediately know which operation occurred first. (It will be shown below that any non-trivial RMW type has this property.)

Theorem 6. *Let* `T` *be any type. Suppose there exist two (not necessarily different) operations op_0 and op_1, and a state q of type* `T` *so that op_0 and op_1 each return different responses depending on the order that the two operations are performed on an object initially in state q. Then,* $\mathrm{cons}(\mathrm{trans}(\mathtt{T}, m)) \geq 2^{m-1}$.

Proof. For $i = 0, 1$, let R_i be the response returned by op_i when an object of type `T` is in state q, and R'_i be the response returned by op_i when it is preceded by the other operation, op_{1-i}. The hypothesis requires that $R_0 \neq R'_0$ and $R_1 \neq R'_1$.

Consider a system with 2^{m-1} processes. Partition the set of processes into two non-empty teams, A and B, each with 2^{m-2} processes. By Lemma 5, it suffices to show that $\mathrm{trans}(\mathtt{T}, m)$ can solve the $\mathrm{team}(2^{m-2}, 2^{m-2})$ problem.

Arrange the base objects into trees and assign groups of processes to each leaf as in the proof of Proposition 4 for the case where $n = 2$. Processes on team A should be assigned to the left subtree of T, and processes on team B should be assigned to the right subtree of T. Initialize all base objects to the state q.

The team consensus protocol will mimic the operation of the algorithm given in Proposition 4 with $n = 2$. However, instead of agreeing on an input value, the algorithm will be used to agree on the team of the first process to access the transactional object. Once this can be accomplished, it is easy to solve the team$(2^{m-2}, 2^{m-2})$ problem: each process first writes its input value into a register belonging to its team, and when the identity of the winning team is known, the value stored in that team's register is returned as the output.

To simulate the filtering action of a node below the root of a tree, one process that accesses the node performs op_0 and the other process performs op_1. If the process that performed op_i receives the response R_i, then it has won at that node. If it receives the response R'_i, then it has lost at that node. The two processes that access the root of T determine which team accessed the root first in exactly the same way.

To simulate the operation that stores a value into the root of T_k^j (during the jth transaction performed by a process), the process applies the operation op_0 to the object if and only if it wants to record the fact that a process from team B took the first step. If it wants to indicate that a process from team A took the first step, it does nothing. To simulate the operation that reads the contents of the root of T_k^j (during the $(j+1)$th transaction performed by a process), the process applies the operation op_1 to the base object located at the root. It interprets the response R_1 as indicating that a process from team A went first, and the response R'_1 as indicating that a process from team B went first.

Clearly, each transaction contains at most m operations on base objects. The correctness of the algorithm can be shown in exactly the same way as in the proof of Proposition 4. □

Corollary 7. *For any non-trivial RMW type* $\mathtt{T}$*,* $\mathrm{cons}(\mathrm{trans}(\mathtt{T}, m)) \geq 2^{m-1}$.

Proof. Since `T` is non-trivial, there exists some operation, op, that applies a function f with $f(q) \neq q$ for some state q. Otherwise, one could trivially simulate `T` without using shared memory by returning the initial state of the object as the response to every operation. The operation op can be used as both operations op_0 and op_1 of Theorem 6. If op is performed when the object is in state q, it returns the response q. However, if op is performed when another op has already been performed on an object in state q, the second op returns a different response, $f(q)$. The lower bound follows from Theorem 6. □

The `test&set` object defined in Sect. 1.2 is perhaps the most basic non-trivial RMW type. The following proposition demonstrates that this very simple type becomes even more powerful in the transactional setting than Corollary 7 suggests. The algorithm used in the proof has a different flavour from the algorithm in the previous proof. Once a base object has been accessed, further accesses never change its state, so it is not necessary to carefully control access to the base objects to avoid erasing information stored in them.

Proposition 8. *The transactional object* $\mathrm{trans}(\mathtt{test\&set}, 3)$ *is universal.*

Proof. Let n be any integer greater than one. An algorithm will be given that uses $\mathrm{trans}(\mathtt{test\&set}, 3)$ to solve the team$(1, n-1)$ problem. It follows from Lemma 5 that the consensus number of $\mathrm{trans}(\mathtt{test\&set}, 3)$ is infinity. Here, it is described how every process can determine which team first accesses the transactional object. Once this can be done, it is easy to solve team consensus using two additional registers, as in the preceding proof.

Divide the n processes into two teams $A = \{P_1\}$ and $B = \{P_2, \ldots, P_n\}$. The algorithm uses $2n+1$ base objects, labelled $A_1, \ldots, A_n, B_1, \ldots, B_n$ and C, all of which are initially in state 0.

The object C is used to determine which team wins: every process will eventually discover which team accessed C first. The other objects are used to distribute the information about the winning team to all of the processes. First, the method of distributing this information will be described informally. Processes access the objects $A_1, \ldots, A_n$ in order so that A_{i+1} is never set to 1 while A_i still has value 0. The same comment applies to the objects $B_1, \ldots, B_n$. Consider some moment in the computation. Let a be the largest index such that A_a has been set to 1. Let b be the largest index such that B_b has been set to 1. The information about the first team to access object C is stored in the other base objects by maintaining the following invariant after each complete transaction.

Invariant: If a process from team A accessed the transactional object first, then $a = b + 1$. If a process from team B accessed the transactional object first, then $b = a + 1$.

Each process retrieves information about which team first accessed the base object C by accessing pairs (A_i, B_i) for increasing values of i until it finds a pair where only one of the two base objects is set to 1.

The algorithm will now be described in detail. Process P_1 performs a single transaction. It first performs a *test&set* operation on object C. If it receives the response 1, P_1 knows that some other process has already accessed C, so it can conclude that a process on team B accessed the transactional object first, and it need not perform any further actions. On the other hand, if it receives the response 0, then it knows it is the first process to access the transactional object. It then performs a *test&set* operation on object A_1 to ensure that the invariant holds and performs no further actions.

The algorithm for a process on team B is more complicated. In the first transaction, the process performs a *test&set* operation on the base object C. If the result is 0, it knows that it is the first process to access the transactional object. It then performs the operation *test&set* on B_1 as part of the same transaction in order to satisfy the invariant. If the process receives the result 1 from C, it knows that some other process has already accessed the transactional object. The process must then use the other base objects to determine which team made the first access. The process performs a number of transactions, accessing A_1 and B_1 in the first transaction, then A_2 and B_2 in the second transaction, and so on, until it gets different responses from two objects A_i and B_i. Suppose that A_i returns 1 and B_i returns 0, indicating that a process from team A accessed the transactional object first. To ensure that the invariant remains true, the process performs a *test&set* operation on A_{i+1} as part of the same transaction. It can then halt, knowing that team A performed the first transaction. The case where A_i returns 0 and B_i returns 1 is symmetric.

It must be checked that this algorithm does terminate. The first process to perform a transaction that accesses the pair of objects A_i and B_i will halt at the completion of that transaction. Thus, at most $n - i$ processes on team B will perform the transaction that accesses A_i and B_i. Thus, every process will perform at most n transactions. It is easy to see that the invariant is true after each complete transaction, and the correctness of the algorithm follows. □

The algorithm given in this proof can easily be adapted to work for any base type that is equipped with an operation whose first invocation returns a response different from any subsequent invocation's response.

5 A General Bound

Theorem 6 gives a lower bound on the consensus numbers of transactional objects that applies to a very large class of base types. In this section it will be shown that a property which is weaker than the hypothesis of Theorem 6 and is satisfied by *any* non-trivial base type can be used to prove a weaker, but still exponential, lower bound on consensus numbers of transactional objects. The hypothesis of Theorem 6 assumed that two processes can each perform a single operation so that both processes can immediately tell which of the two performed its operation first. The following theorem shows that a similar algorithm will solve consensus even if only one of the two processes can determine which process accessed the base object first. Because of the weaker assumption, processes will have to access two base objects at each node, and this has the effect of halving the tree heights and producing a corresponding reduction in the number of processes that can be assigned to the leaves.

Theorem 9. *For any non-trivial base type* $\mathtt{T}$*,* $\mathrm{cons}(\mathrm{trans}(\mathtt{T}, m)) \geq 2^{\lfloor (m-1)/2 \rfloor}$.

Proof. First, the non-triviality of $\mathtt{T}$ will be used to prove that $\mathtt{T}$ has a useful property, similar to the hypothesis of Theorem 6. Since $\mathtt{T}$ is non-trivial, the response to some operation, op_0 cannot depend entirely on the state in which the object is initialized. Thus, there is some (shortest) sequence of operations that takes the object from a state where it would return the response R to op_0 into a state where it would return a different response R' to op_0. Let op_1 be the last operation in this sequence, and let q be the state of the object just before op_1 is applied. If an object begins in state q and two processes apply the operations op_0 and op_1, the process applying op_0 can tell whether the other process has already taken its step, by checking whether it gets the response R or R'.

Now, consider a system with $2^{\lfloor (m-1)/2 \rfloor}$ processes, partitioned into two teams A and B of equal size. As in the proof of Theorem 6, it suffices to give an algorithm that allows every process to determine whether a process from team A or from team B accesses the transactional object first.

The base objects are organized into trees as in the proof of Proposition 4 for the case when $n = 2$, except the height of the trees are reduced to $\lfloor (m-3)/2 \rfloor$, and nodes of the trees have two base objects instead of one. (However, the roots of the trees T_k^j need only one base object.) The two objects at a node will be referred to as the left and right objects of that node. All base objects are initialized to the state q. Associate groups of two processes with the leaves of the trees as in Proposition 4.

The algorithm to determine the team containing the process that performed the first transaction will mimic the algorithm of Theorem 6. To simulate the filtering action of the nodes below the roots of trees, one process performs op_0 on the left object and op_1 on the right object, and the other process performs op_1 on the left object and op_0 on the right object. If a process receives the response R from op_0, then the process wins at the node, and if the response is R', then the process loses at the node. The winning team at the root of T is determined in the same way.

To simulate an operation that writes the winning team to the root of T_k^j, a process performs op_1 on the object at that node if and only if it wants to indicate that a process from team B took the first step. Otherwise it does nothing. To simulate the operation that reads the contents of the root of T_k^j, the process performs op_0 on the object in the node. The response R indicates that team A took the first step and the response R' indicates that team B took the first step.

In the worst case, a transaction accesses all base objects on some path from a leaf to a root of a tree plus one base object in another tree. Each transaction therefore contains at most $2(\lfloor (m-3)/2 \rfloor + 1) + 1 \leq m$ memory accesses, as required. The proof of correctness is the same as in Proposition 4. □

Corollary 10. *If* $\mathtt{T}$ *is neither trivial nor universal, there is some* m *such that* $\mathtt{T}$ *cannot provide a wait-free implementation of* $\mathrm{trans}(\mathtt{T}, m)$.

Proof. If $m = 2\log_2(\mathrm{cons}(\mathtt{T}))+3$, then $\mathrm{cons}(\mathrm{trans}(\mathtt{T}, m)) > \mathrm{cons}(\mathtt{T})$, so $\mathrm{trans}(\mathtt{T}, m)$ is strictly more powerful than $\mathtt{T}$. □

6 Open Questions

It has been shown that the ability to perform even short transactions can greatly increase the consensus numbers of object types. Can corresponding upper bounds on consensus numbers of transactional objects be proved?

In the multi-object model, memory accesses within an atomic operation must be performed on different base objects. In the transactional model, processes are free to access base objects repeatedly during an atomic action, but this ability was not used in the algorithms given here. Is there any case where the ability to access the same base object repeatedly in a transaction really does help?

Another possible setting for transactional objects would be one where each transaction must perform all of the shared-memory accesses on a single base object. How does such a restriction affect the consensus power of objects? In the case of `test&set` base objects, transactions on a single base object give no additional power, since any such transaction has the same effect as doing a single *test&set* operation. Therefore, this single-object version of transactional memory is strictly weaker than general transactional objects.

Acknowledgements I thank my advisor, Faith Fich, for suggesting many improvements. I also thank the anonymous reviewers for their helpful comments.

References

1. Yehuda Afek, Michael Merritt, and Gadi Taubenfeld. The power of multi-objects. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 213–222, 1996.
2. James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.
3. Hagit Attiya and Eyal Dagan. Universal operations: Unary versus binary. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 223–232, 1996.
4. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1), pages 124–149, January 1991.
5. Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5), pages 745–770, November 1993.
6. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th International Symposium on Computer Architecture*, pages 289–300, 1993.
7. Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, 1994.
8. Prasad Jayanti. Robust wait-free hierarchies. *Journal of the ACM*, 44(4), pages 592–614, July 1997.
9. Prasad Jayanti and Sanjay Khanna. On the power of multi-objects. In *Distributed Algorithms, 11th International Workshop*, volume 1320 of *LNCS*, pages 320–332, 1997.
10. Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4), pages 579–601, October 1988.
11. Nancy A. Lynch. *Distributed Algorithms*, chapter 8. Morgan Kaufmann, 1996.
12. Mark Moir. Transparent support for wait-free transactions. In *Distributed Algorithms, 11th International Workshop*, volume 1320 of *LNCS*, pages 305–319, 1997.
13. Eric Ruppert. Determining consensus numbers. In *Proc. 16th ACM Symposium on Principles of Distributed Computing*, pages 93–99, 1997.
14. Eric Ruppert. Consensus numbers of multi-objects. In *Proc. 17th ACM Symposium on Principles of Distributed Computing*, pages 211–217, 1998.
15. Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2), pages 99–116, February 1997.