The Power of Multi-objects

Yehuda Afek [*] Michael Merritt [†] Gadi Taubenfeld [‡]

January 1, 1998

Abstract

We consider shared memory systems that support multi-object operations in which processes may simultaneously access several objects in one atomic operation. We provide upper and lower bounds on the synchronization power (consensus number) of multi-object systems as a function of the type and the number of objects that may be simultaneously accessed in one atomic operation. These bounds imply that known classifications of component objects fail to characterize the synchronization power of their combination. In particular, we show that in the context of multi-objects, *fetch&add* objects are less powerful than *swap* objects, which in turn are less powerful than *queue* objects. This stands in contrast to the fact that *swap* can be implemented from *fetch&add*. Herein we introduce a restricted notion of implementation, called direct implementation. We show that, if an object Y has a direct implementation from X then also the Y based multi-object can be implemented from the X based multi-object. Following the above, we derive results such as: there is no *direct* implementations of a *swap* object, and *queue* object from any collection of commutative objects (e.g., *fetch&add, test&set*).

1 Introduction

A shared memory system is a collection of objects accessed by a collection of processes. Generally, it is assumed that each process accesses the individual objects, atomically and one at a time, and that the individual accesses by different processes are interleaved. It is possible to consider alternative access methods, such as permitting processes to access multiple objects in a single atomic step and that these multiple accesses by different processes are interleaved. Each of these alternative views of shared memory can be understood as a different way of combining objects. This leads naturally to the idea of alternative object *combinators*, constructors that produce *compound* objects from sets of *component* objects.

This paper focuses on the properties of a specific parallel combinator, the *multi-object* combinator, which given a shared memory system with a set of component objects O, and a parameter m, produces a compound object O^m in which processes are allowed to simultaneously (and atomically) execute operations on up to m of the component objects in O. An example is a *register* multi-object which allows processes to read or write up to

[*]Computer Science Dept., Tel-Aviv Univ., Israel 69978, and AT&T Labs. Partial support provided by the Israel/United States Binational Science Foundation, Grant #94002.

[†]AT&T Labs, 600 Mountain Avenue, Murray Hill, NJ 07974. Partial support provided by the Israel/United States Binational Science Foundation, Grant #94002.

[‡]The Open Univ., 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel, and AT&T Labs

m registers in a single atomic operation [MT94]. This object generalizes the *m-assignment* objects discussed by Herlihy [Her91] (which support writes to m registers in a single atomic operation), and *snapshot* objects [AAD$^+$93, And94, And93] (which support reads of multiple registers in a single atomic operation). In [Pat71], Patil has defined a stronge type of semaphore, called PV-multiple, which enables to access several basic semaphores in one step.

We narrow our attention to the synchronization power (consensus number) of multi-objects, as a function of the types of the components, and of m. Although the consensus number of O^m may depend on the number of objects in O, this paper studies the impact of the *kinds* of objects in O (e.g., sets of *swap* versus sets of *queue* objects), not the *number* of such objects. Hence, we assume that each set contains sufficient copies of the component objects for the algorithms we present, and each impossibility result holds for an infinite number of component objects. For example, we show that if m different *swap* objects can be accessed in a single atomic step, then *n-consensus* (n processes consensus) can be implemented for as many as $n = \sqrt{2m+5}$ processes, but no such algorithm exists if $n \geq \frac{2}{\sqrt{3}}\sqrt{2m+4}+\frac{1}{3}$. Previous research on the synchronization power of shared-objects studied the dependence of such power on the type of the object [Her91], the total number of objects available [Jay93] and their size [AS94]. Our research studies a third dimension: how the power varies with the number of objects that are accessed atomically in one step.

Bounds on $Con(X^m)$, the consensus number of multi-objects X^m, in which up to m component objects of type X can be simultaneously accessed.		
multi-object X^m	Lower bound on $Con(X^m)$: X^m implements consensus for at least this many processes.	Upper bound on $Con(X^m)$: No implementation of consensus from X^m exists for more than this many processes.
fetch&add^m †	2	2
swap^m	$\lfloor\sqrt{2m+5}\rfloor$	$\frac{2}{\sqrt{3}}\sqrt{2m+4}+\frac{1}{3}$
c-consensus^m, $c > 2$	$\left\lfloor\frac{c}{\sqrt{8}}\sqrt{2m-\frac{4m}{c}+\frac{8}{c^2}}\right\rfloor$	$c\sqrt{2m}+1$
register^m [Herlihy91]	$2m-2$	$2m-2$
queue^2	∞	∞

† The same bound holds for any collection of commutative objects, such as *fetch&increment* and *test&set*.

Figure 1: Summary of results

Some general properties of the multi-object combinator emerge from this investigation. They imply, for example, that the consensus number of the component objects is in general inadequate to explain the properties of their multi-object combination (e.g., the consensus number of the combination). This follows from observing that the consensus number of *fetch&add* multi-objects is 2, the consensus number of *swap* multi-objects grows as the square root of the number of components that can be accessed simultaneously, and the consensus number of *queue* multi-objects is infinite, if as few as two of the queue objects can be accessed simultaneously. Yet, the three component objects each have consensus number 2.

1.1 Summary of results

The table in Figure 1, summarizes properties of some specific multi-objects.[1]

We noted above that these results imply that $Con(O^m)$, the consensus number of multi-object O^m, is not in general a function of m and of $Con(O)$, the consensus number of the component objects in O. (Though since multi-objects support strictly more operations than are supported by the components alone, $Con(O^m) \geq Con(O)$.) Hence, determining the consensus number of O^m affords a means of classifying objects that can be more discriminating than the consensus number of O alone, as is clear from the table in the distinction between $Con(\textit{fetch\&add}^m)$, $Con(\textit{swap}^m)$, and $Con(\textit{queue}^2)$, since $Con(\textit{fetch\&add}) = Con(\textit{swap}) = Con(\textit{queue}) = 2$.

Another theme that emerges from our investigation is the effect of supporting *read* operations. For many objects, the consensus number increases with the number of processes that can read the object. This is suggested in the table in the distinctions between *fetch&add*, *swap*, and *c-consensus*. Intuitively, $\textit{fetch\&add}^m$ allows two processes to reach consensus, but no other process can read the result, while we show that $\textit{swap}^2$ allows two processes to reach consensus, and a third to read the result. Finally, when $c > 2$, *c-consensus* objects can be used to allow $c - r$ processes to reach consensus, and r other processes to read the result, for any r, $1 \leq r < c$. As we discuss, the consensus number is also greatly affected by adding read operations directly to specific objects, e.g., *queues*. (Adding operations to objects is an example of another object combinator as discussed in Subsection 5.2.1.)

The standard notion of object implementation depends crucially on the fact that when one object type X implements another object type Y, (denoted as $X \rightarrow Y$), then in any memory system in which instances of Y are combined in the standard way with objects of other types, each instance of Y can be replaced by sufficiently many instances of X, of *registers*, and by algorithms run by the processes in place of direct accesses to Y, resulting in a memory system that simulates the original. Such a substitution of X for Y when $X \rightarrow Y$ is not possible in multi-objects. That is, in general $X \rightarrow Y$ does not imply $X^m \rightarrow Y^m$. For example, it is known that $\textit{fetch\&add} \rightarrow \textit{swap}$ [AWW93], but it follows from the upper bound for $\textit{fetch\&add}^m$ and the lower bound for $\textit{swap}^m$ that $\textit{fetch\&add}^m \not\rightarrow \textit{swap}^m$, for $m \geq 2$. Similarly it follows that $\textit{swap}^m \not\rightarrow \textit{queue}^m$, for $m \geq 2$.

However, it is possible to define a more constrained implementation relation, which does allow one object to replace another within multi-object constructions. We call this relation, defined in the next section, the *directly implements* relation, and write it $X \stackrel{di}{\rightarrow} Y$. This relation satisfies two key composition properties; if $X \stackrel{di}{\rightarrow} Y$, then both $X^m \stackrel{di}{\rightarrow} Y^m$ and $X \rightarrow Y$, and hence also $X^m \rightarrow Y^m$. As a consequence, $\textit{fetch\&add} \stackrel{di}{\not\rightarrow} \textit{swap}$ (cannot directly implement) and $\textit{swap} \stackrel{di}{\not\rightarrow} \textit{queue}$. Interestingly these $\stackrel{di}{\not\rightarrow}$ relations parallel the relative difficulties in constructing these objects from each other as reported in [AWW93], i.e., the construction of *fetch&add* is relatively simpler than that of *swap*, and the authors of [AWW93] were not able to give a construction of *queue* from either *swap* or *fetch&add*.

[1] **Comments and explanations.** Given a set of objects O, O^m denotes the result of applying the multi-object combinator to the objects in O. This table uses the notation T^m, where T is an object *type*, not a set of objects. Hence, by T^m we mean the class of objects that can be obtained by applying the multi-object combinator to sets of objects of type T.

We show how to use the $\stackrel{di}{\rightarrow}$ relation to obtain modular constructions of consensus algorithms using multi-objects. For example, we present modular constructions of *n-consensus* algorithms from $swap^m$, c-$consensus^m$, and $register^m$ objects. These modular constructions hinge on the definition of (f,r)-*consensus* objects, which implement consensus for f processes, and allow r other processes to read the result. We first present an *n-consensus* algorithm using (f,r)-$consensus^m$ objects, where n is at most $\left\lfloor \frac{f-r}{2} + \sqrt{(\frac{f-r}{2})^2 + f - r + 1 + (f-1)rm} \right\rfloor$. Implementations of *n-consensus* from $swap^m$, c-$consensus^m$, and $register^m$ objects are then derived by presenting direct implementations of $(2,1)$-*consensus* from $swap^2$, $(c-r,r)$-*consensus* from *c-consensus*, and $(2,r)$-*consensus* from $register^2$.

Definitions of multi-objects and different implementation relations and their properties are provided in Section 2. In Section 3 we describe several *n-consensus* algorithms using different multi-objects and in Section 4 we present impossibility results to bound the algorithmic results of Section 3. Section 5 closes with a discussion of additional object combinators (such as adding or removing operations from an object). It is possible to construct many familiar objects from a few simple initial components, and simple object combinators. We suggest that a general theory of shared memory may emerge from a better understanding of such simple components and object combinators.

2 The multi-object combinator

The notion of one object implementing another is central to any theory of distributed or concurrent computation. One formulation of a standard notion of wait-free implementation [HW90] defines objects as I/O automata [LT87] and says that object type X implements object type Y if a wait-free algorithm exists which accesses instances of X and of shared registers, resulting in a simulation of an object of type Y. We write $X \rightarrow Y$ when such a relation exists between object types X and Y. We also write $Con(X)$ to be the maximum n such that $X \rightarrow$ *n-consensus*, or ∞ if $X \rightarrow$ *n-consensus* for all n.

As we discuss in the introduction, it is important to consider whether $X \rightarrow Y$ implies $X^m \rightarrow Y^m$, and some of our results demonstrate this is not the case. For example, *fetch&add*$\rightarrow$*swap* [AWW93, Wei94], but $fetch\&add^m \not\rightarrow swap^m$. (Because $Con(fetch\&add^m) = 2$ and $Con(swap^m) = O(\sqrt{m})$.)

It is possible to define a more constrained implementation relation, X *directly implements* Y, or $X \stackrel{di}{\rightarrow} Y$, that does satisfy the property: $X \stackrel{di}{\rightarrow} Y$ implies $X^m \stackrel{di}{\rightarrow} Y^m$. In particular, X directly implements Y if X implements Y using a construction in which every operation op_y of Y is implemented by a series of operations on instances of X and of registers, and such that the linearization of the implementation of op_y can always be placed at the first access to an instance of X. We state without proof some simple properties satisfied by the direct implementation relation and the multi-object combinator (see Section 5 for detailed definitions of objects and object combinators):

Theorem 1 *Let X and Y be two object types.*

1. $X \stackrel{di}{\rightarrow} Y$ *implies* $X \rightarrow Y$.
2. $X \stackrel{di}{\rightarrow} Y$ *implies* $X^m \stackrel{di}{\rightarrow} Y^m$, *for all* $m > 0$.

3. $X \stackrel{di}{\rightarrow} Y$ *implies* $X^m \rightarrow Y^m$, *for all* $m > 0$.

4. $X^{pq} \stackrel{di}{\rightarrow} (X^p)^q$, *for all* $p, q > 0$.

5. $X^p \stackrel{di}{\rightarrow} Y$ *and* $Y^q \rightarrow Z$ *implies* $X^{pq} \rightarrow Z$, *for all* $p, q > 0$.

6. $X^p \stackrel{di}{\rightarrow} Y$ *implies* $Con(X^{pq}) \geq Con(Y^q)$, *for all* $p, q > 0$.

7. $Con(X^m) < Con(Y^m)$, *for any* $m > 0$, *implies* $X \stackrel{di}{\not\rightarrow} Y$.

3 Implementations of *consensus* using multi-objects

We prove lower bounds on the consensus number of various multi-objects X^m by presenting consensus algorithms that access only X^m objects and read/write registers. Generally, the presentations assume a fixed number of processes, n, and describe a general algorithm using simultaneous access to sufficiently many instances of X.

The algorithms for implementing *n-consensus* from *swap*^{m}, *register*^{m}, and *c-consensus*^{m} objects are presented as reductions into one basic algorithm that implements *n-consensus* from a newly defined object, called (f, r)-*consensus*. The basic algorithm implements *n-consensus* using (f, r)-*consensus*^{m} multi-objects, and is a generalization of Herlihy's implementation from *m-assignment* objects [Her91]. By part 5 of Theorem 1, the basic algorithm can be combined with direct implementations of (f, r)-*consensus* from *swap*^{2} or *register*^{2} to implement *n-consensus* from *swap*^{m} and *register*^{m}. Despite technical difficulties, a similar construction can be used to replace $(c - r, r)$-*consensus* objects with *c-consensus* objects in the basic algorithm, implementing *n-consensus* from *c-consensus*^{m}.

3.1 Implementation of *n-consensus* from (f, r)-*consensus* objects

Specifically, an (f, r)-*consensus* object is accessed by no more than $(f+r)$ distinct processes, f of which may invoke the consensus operations *propose*(0) or *propose*(1), and r of which may invoke *read* operations. A read linearized before any consensus operation returns the value $\perp$; otherwise, it returns the consensus value (0 or 1). (The consensus operations must be linearized as in an *f-consensus* object, i.e., the reads have no apparent effect on the object state.)

Theorem 2 $Con((f, r)\text{-consensus}^m) \geq \left\lfloor \frac{f-r}{2} + \sqrt{(\frac{f-r}{2})^2 + f - r + 1 + (f-1)rm} \right\rfloor$.

Proof: Given a sufficient number of (f, r)-*consensus*^{m} objects, we describe an algorithm for reaching consensus for any n processes, provided that $n \leq \left\lfloor \frac{f-r}{2} + \sqrt{(\frac{f-r}{2})^2 + f - r + 1 + (f-1)rm} \right\rfloor$. processes. To reach consensus the n processes iteratively use a basic building block that solves consensus among two groups of processes, under the restriction that the members of each group all propose the same value. (The two groups may or may not propose different values.) The basic building block is used to run a series of competitions among the processes, starting with pairs of groups of one process each. Each pair of groups reaching consensus in one round, competes together as a single group in the next round, until two final groups of at most $\lceil n/2 \rceil$ processes compete.

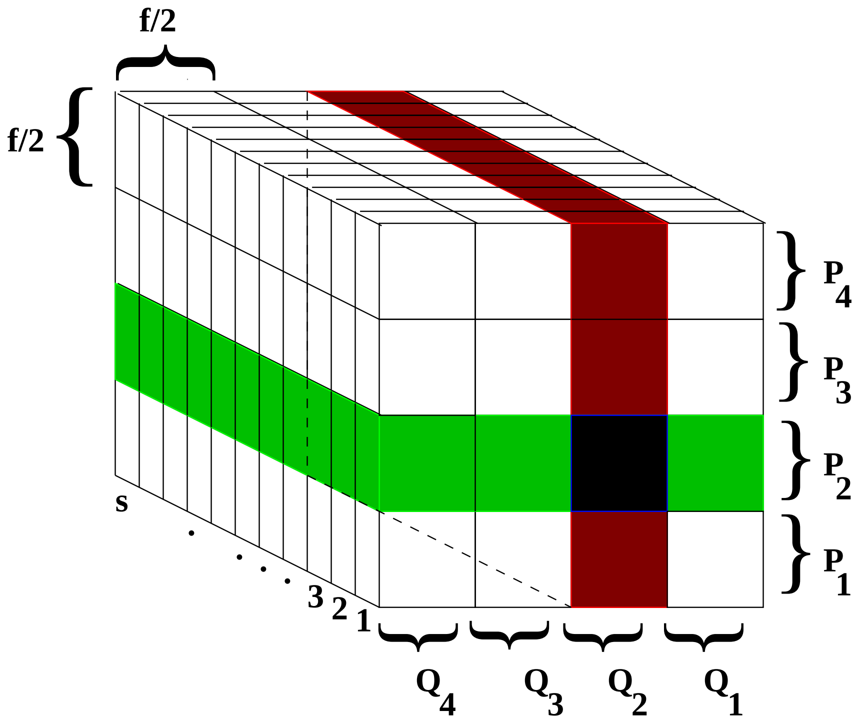

Figure 2: 3-dimensional array of (f,r)-*consensus*^{m} objects in the basic *n-consensus* algorithm.

Consensus between two groups: Let the two groups of processes competing in the basic building block be $P = \{p_1,\ldots,p_{n_p}\}$ and $Q = \{q_1,\ldots,q_{n_q}\}$, where all members of the same group propose the same value. The basic idea is that a process tries to reach consensus on the value of its group by competing against all the members of the other group, simultaneously, in one atomic step. Furthermore, this simultaneous step accesses enough (f,r)-*consensus* objects so any other processes will be able to later observe the result of the competition.

Let v_p and v_q be the values proposed by members of P and Q, respectively. Assume for simplicity that initially f is divisible by 2 and that n_p and n_q are divisible by $f/2$. Each of P and Q is partitioned into $2n_p/f$ and $2n_q/f$ subsets of $f/2$ processes each. Let $\mathbf{P} = \{P_1, \ldots, P_{2n_p/f}\}$, and $\mathbf{Q} = \{Q_1, \ldots, Q_{2n_q/f}\}$ be the corresponding collections of subsets. Each member of P competes with all members of Q by accessing (f,r)-*consensus* objects. This can be done by arranging $4n_pn_q/f^2$ (f,r)-*consensus* objects in a two-dimensional array $COMPETE(1..2n_p/f, 1..2n_q/f)$. Each process in P_i simultaneously proposes v_p to the $2n_q/f$ (f,r)-*consensus* objects $COMPETE(i,j)$, for all j, $1 \le j \le 2n_q/f$, and similarly each process in Q_i simultaneously proposes v_q to the $2n_p/f$ (f,r)-*consensus* objects $COMPETE(j,i)$, for all j, $1 \le j \le 2n_p/f$.

The outcome of each component (f,r)-*consensus* object $COMPETE(i,j)$ is apparent to the f members of $P_i \cup Q_j$, once they propose a value. The other $n_p + n_q - f$ processes in $(P \cup Q) - (P_i \cup Q_j)$ need to be able to read the outcome, but only r of them can access $COMPETE(i,j)$. Let $s = \lceil (n_p + n_q - f)/r \rceil$. We need to replicate each (f,r)-*consensus* object in $COMPETE(i,j)$ a total of s times, and partition the $n_p + n_q - f$ process indices in $(P \cup Q) - (P_i \cup Q_j)$ into a collection of subsets $\mathbf{R}^{ij} = \{R^{ij}_1,\ldots,R^{ij}_s\}$, each containing at most r processes. Each subset is assigned one of replica. Hence, the final data structure is the three-dimensional array $COMPETE(1..2n_p/f, 1..2n_q/f, 1..s)$, illustrated in Figure 2. Each process p in P_i simultaneously proposes v_p to the $s2n_q/f$ (f,r)-*consensus* objects $COMPETE(i,j,k)$, for all j, $1 \le j \le 2n_q/f$, and all k, $1 \le k \le s$, and similarly each process in Q_i simultaneously proposes v_q to the $s2n_p/f$ (f,r)-*consensus* objects $COMPETE(j,i,k)$, for all j, $1 \leq j \leq 2n_p/f$, and all k, $1 \leq k \leq s$. We call these steps the *write steps* for the building block, in this round of the algorithm.

The first member of either group to take a write step determines the output of the competition for all the processes in P and Q, by setting all copies of all (f,r)-*consensus* objects in a row or column of $COMPETE$ to v_p or v_q. Hence, if a process's write step returns the same value, v, from all component objects, it can return v as the outcome of this step of the algorithm.

If a process $p \in P_i \in \mathbf{P}$ returns different values from different component objects, it knows both that $v_p \neq v_q$ and that some of the processes in Q have already taken a write step. Specifically, if any object $COMPETE(i,j,k)$ returns v_q, process p knows that at least one process $q \in Q_j$ has taken a write step. Without loss of generality, let q be the first member of Q_j to take a write step.

Next p has to determine whether q took its write step before any member of P. It does this by reading the $2n_p/f$ (f,r)-*consensus* objects $COMPETE(k,j,t)$ such that $p \notin P_k$ and $p \in R_t^{k,j}$, one at a time. Process q took its write step before all members of P if and only if all of these reads return v_q. Hence, p can return v_q from this step if all these reads return v_q. Call these steps the *read steps* by p for Q_j.

Process p takes read steps for every Q_j such that $COMPETE(i,j,k)$ returned v_q, until it either returns v_q or reads for all such Q_j without returning v_q. In the latter case, p can conclude that no q took a write step before all the members of P, and returns v_p.

The full array contains $4sn_pn_q/f^2$ (f,r)-*consensus* objects, where $s = \lceil (n_p+n_q-f))/r \rceil$. During a write step, processes need to simultaneously access as many as $2sn'/f$ of these objects, where n' is the maximum of n_p and n_q.

We made the simplifying assumption that n_p and n_q are divisible by $f/2$. Suppose this were not the case–suppose, for example, that there were one $p \in P$ and one $q \in Q$ left over after distributing the processes in P and Q in groups of $f/2$. Then p and q would have to contend alone in a (f,r)-*consensus* object, and there would be $n_p + n_q - 2$ other processes to read the outcome, instead of $n_p + n_q - f$. With only r readers per object, there would need to be $\lceil (n_p + n_q - 2)/r \rceil$ copies of this object instead of $s = \lceil (n_p + n_q - f)/r \rceil$ copies. But observe that in the algorithm, no consensus object is read by a process unless at least one other process has taken a write step that accesses it. This means that in this algorithm, $f - 2$ processes can use the consensus operation *propose*(v) to read the object, and we can remove the restriction that n_p and n_q are divisible by $f/2$.

The final round of the competition in which each group contains at most $\lceil n/2 \rceil$ processes requires the most (f,r)-*consensus* objects to be accessed simultaneously: the full array contains as many as $4\lceil (n-f)/r \rceil \lceil n/2 \rceil \lfloor n/2 \rfloor / f^2$ (f,r)-*consensus* objects, and during their write step, processes need to simultaneously access as many as $\frac{\lceil n/2 \rceil \lceil (n-f)/r \rceil}{\lfloor f/2 \rfloor}$ of these objects. Hence:

$$m \geq \frac{\lceil n/2 \rceil \lceil (n-f)/r \rceil}{\lfloor f/2 \rfloor}. \tag{1}$$

It follows that for a fixed $m > 1$, $f > 1$, and $r > 0$, there is an implementation of n-process *consensus* via this construction, provided $n \leq \frac{f-r}{2} + \sqrt{(\frac{f-r}{2})^2 + f - r + 1 + (f-1)rm}$. ∎

3.2 Implementations of *n-consensus* from *swap*, *register* and *c-consensus* objects

Implementations from *swap* and *register*: It is quite straightforward to directly implement $(2,1)$-*consensus* from *swap*^{2}: Let p_0 and p_1 be the processes with access to the consensus operations *propose*(0) and *propose*(1), and let p_r be the reading process. The construction uses three swap objects, S_0, S_1, and S_{01}. Process p_0 proposes v_0 by swapping it into S_0 and S_{01}, and process p_1 proposes v_1 by swapping it into S_1 and S_{01}. If either sees $\perp$ as the return value from S_{01}, it returns its input value. Otherwise, it returns the value returned from S_{01}. To read, process p_r swaps $\perp$ into S_0 and S_1. If both swaps return $\perp$, p_r knows the consensus value has yet to be determined. If exactly one swap returns a value other than $\perp$, or both return the same value, the reader knows that value is the outcome of the consensus. Finally, if two different values are returned, the reader swaps $\perp$ into S_{01}, and the read operation returns the value which was *not* returned by this final swap. (Since the consensus value is stable once determined, any subsequent *propose* or *read* operations do not need to access the shared memory.)

It is fairly obvious that this algorithm implements $(2,1)$-*consensus*. Moreover, each operation can be linearized with its first *swap*^{2} operation. A very similar algorithm suffices to show that for all $r \geq 1$, *register*$^2 \overset{di}{\rightarrow} (2,r)$-*consensus*. Hence, we have:

Theorem 3

1. swap$^2 \overset{di}{\rightarrow} (2,1)$-consensus.
2. $\forall r \geq 1$, register$^2 \overset{di}{\rightarrow} (2,r)$-consensus.

Corollary 4

1. $Con(\text{swap}^m) \geq \left\lfloor \sqrt{m + \frac{5}{4}} + \frac{1}{2} \right\rfloor$.
2. $Con(\text{register}^m) \geq m - 2$

Proof: Part 1: By Theorem 3.1, and Theorem 1.6, $Con(swap^m) \geq Con((2,1)\text{-}consensus^{\lfloor m/2 \rfloor})$. Setting $f = 2$ and $r = 1$ in Formula 1, the latter term is at least $\sqrt{m + \frac{5}{4}} + \frac{1}{2}$.

Part 2: By Theorem 3.2, and Theorem 1.6, $\forall r \geq 1$, $Con(register^m) \geq Con((2,r)\text{-}consensus^{\lfloor m/2 \rfloor})$. Replacing m by $\lfloor m/2 \rfloor$ and for $f = 2$, then $\forall r \geq 1$ Formula 1 becomes $\lfloor m/2 \rfloor \geq \lceil n/2 \rceil \lceil (n-2)/r \rceil$. It follows that $Con((2,r)\text{-}consensus^{\lfloor m/2 \rfloor})$ is at least $m - 2$. ∎

These results depend on component-by-component direct implementations of $(2,1)$-*consensus* and $(2,k-2)$-*consensus* by *swap*^{2} and *register*^{2}, respectively. A more direct and optimized implementation of the *COMPETE* array by *swap* and *register* multi-objects, respectively, gives us bounds of $Con(swap^m) \geq \sqrt{2m+5}$, and $Con(register^m) \geq 2m - 2$.

Implementations from *c-consensus*: It is also possible, and seemingly quite direct, to simulate (f,r)-*consensus* objects with *c-consensus* objects. However, devising such a simulation is complicated by the fact that the *propose* operation permanently changes a *consensus* object, making it difficult to use to directly simulate a *read* operation. However, the (f,r)-*consensus*^{m} implementation in Theorem 2 satisfies the following property:

No component object is read until it has been accessed by a *propose* operation.

This leads naturally to the definition of a constrained environment, E, in which such an ordering of *propose* and *read* operations is guaranteed, and in which we can require *c-consensus*^{m} objects to simulate (f,r)-*consensus*^{m} objects. Within this constrained environment, calls to (f,r)-*consensus*^{m} objects can be replaced directly with calls to *c-consensus*^{m} objects.

It is possible to condition the notions of $\rightarrow$ and $\stackrel{di}{\rightarrow}$ on the environment E, obtaining weaker implementation relations $X \underset{E}{\rightarrow} Y$ and $X \underset{E}{\stackrel{di}{\rightarrow}} Y$ relating X and Y.

Theorem 5

1. c-consensus^m $\underset{E}{\stackrel{di}{\rightarrow}}$ $(c-r,r)$-consensus^m, *for any* r, $1 < c$, $1 \leq r < c-1$, *and any* $m > 0$.
2. *The implementation of* n-consensus *from* (f,r)-consensus^m *objects satisfies* E.
3. $Con(c\text{-consensus}^m)$
$\geq max_{1 \leq r < c-1}\ Con((c-r,r)\text{-consensus})^m$
$\geq \left\lfloor \frac{c}{\sqrt{8}} \sqrt{2m - \frac{4m}{c} + \frac{8}{c^2}} \right\rfloor$.

3.3 An implementation of *n-consensus* from *queue*^{2} objects

The component *queue* object we consider supports the operations *enqueue* and *dequeue* by every process. A *dequeue* operation on an empty queue returns a special value, $\perp$. (The algorithm presented in this section never enqueues more than $2n+1$ values in any *queue* object, so by assuming the queues can hold at least this many values, the behavior of the component queue object when an enqueue is invoked on a full *queue* is immaterial.)

Theorem 6 $Con(\text{queue}^2) = \infty$.

Proof: We show that for any $n > 1$, n-consensus can be solved by n processes $p_1, \ldots, p_n$ accessing a set of $4n$ queues, accessed 2 at a time. The algorithm is built out of n building blocks, $B_1, \ldots, B_n$, each B_i consisting of four queues, O_1^i, O_2^i, O_3^i, and O_4^i. Process p_i is the *owner* of building block B_i, and the remaining processes are *non-owners* of B_i. Owners and non-owners run different algorithms on each building block, and return either *win* or *lose*. These algorithms guarantee that the non-owners always return a value different from the owner. Moreover, if the owner runs its algorithm before any non-owner, the owner returns *win*, and similarly if a non-owner runs its algorithm before the owner, the non-owners return *win*.

We first describe the algorithms run by the owner and non-owner, and then describe how to use these building blocks to implement *n-consensus*.

The key to understanding the algorithm, given in Figure 3, is that the owner's dequeue operation on O_3^i is racing the non-owner's first enqueue operations on O_3^i and O_4^i. Clearly, dequeuing $\perp$ or an enqueued value suffices to inform the owner whether his operation has run first. It remains for the non-owner to determine whether the owner or some non-owner has moved first.

Implementation of building block B_i:
Variables r_1^k, r_2^k, r_3^k, and r_4^k are local variables of process p_k.
Code for p_i the owner of B_i:
enqueue $(i, 1)$ on O_1^i and on O_2^i ;
dequeue from O_2^i and O_3^i into r_2^i and r_3^i, respectively ;
if $r_3^i = \perp$ return(win) /* No non-owner has enqueued. */
else return($lose$)
Code for p_j a non-owner of B_i:
enqueue $(j, 1)$ on O_3^i and on O_4^i
enqueue $(j, 2)$ on O_3^i and on O_4^i
dequeue from O_3^i and O_4^i into r_3^j and r_4^j, respectively
if $r_3^j \neq r_4^j$ return(win) /* The owner dequeued a queued value from O_3^i. */
else { /* The owner either dequeued $\perp$ from O_3^i or hasn't yet dequeued. */
    enqueue $(j, 3)$ on O_1^i and on O_2^i
    dequeue from O_1^i and O_2^i into r_1^j and r_2^j, respectively
    if $r_1^j = r_2^j$ return(win) /* The owner hasn't dequeued. */
    else /* The owner has dequeued.
        It yet remains to detect if before any enqueue by a non-owner*/
        {*enqueue* $(j, 4)$ on O_3^i and on O_4^i
        dequeue from O_3^i and O_4^i into r_3^j and r_4^j, respectively
        if $r_3^j \neq r_4^j$ return(win) /* The owner dequeued a queued value from O_3^i. */
        else return($lose$) /* The owner dequeued $\perp$ from O_3^i. */
        }
    }

Figure 3: Implementation of building block B_i

First, note that the non-owners enqueue sufficiently many values before dequeuing, to guarantee that no non-owner dequeues an empty queue. Moreover, the two queue's, O_3^i and O_4^i, are always enqueued and dequeued in pairs by the non-owners, with distinct values enqueued each time. So the values dequeued from each are identical until the un-paired dequeue of O_3^i by the owner removes a value. Of course, they are always identical in runs in which the dequeue from O_3^i by the owner returns $\perp$. Hence, as the comments indicate, a non-owner dequeuing distinct values from these queues is a clear indication that the owner dequeued a value (that was enqueued by some non-owner), misaligning the two queue contents for every later pair of dequeues.

Dequeuing identical values from these two queues is less definitive. It means that either the owner hasn't moved, or dequeued an empty queue. The queues O_1^i and O_2^i are used to distinguish these cases, by allowing the non-owners to determine whether the owner has moved. Also accessed in pairs except for the owner's dequeue, the owner's previous enqueue guarantees that it will dequeue a value from O_2^i, misaligning these two queues thereafter. Hence, a dequeue of these two objects suffices to tell whether the owner's dequeue has occurred. If not, the non-owner can safely return *win*. If the values dequeued are not equal, the non-owner still does not know if the owner dequeued before or after the non-owner's first dequeue of O_3^i and O_4^i. A second dequeue of O_3^i and O_4^i, this time knowing the owner has dequeued, disambiguates these cases.

Constructing *n-consensus* **from building blocks** Each process p_i first writes its input value to a shared register, $input_i$. Then, it runs its owner algorithm on building block B_i. Next, it runs its non-owner algorithms on the other building blocks. Let j be the minimum process index of the blocks in which the owner *won*. (That is, the minimum process index of a block in which p_i returned *lose*, or i if i is smaller and p_i returned *win* from B_i.) Then p_i returns the value it reads in $input_j$.

Since every process runs its owner algorithm before its non-owner algorithms, some owner returns *win*, and the building blocks guarantee that all processes will find the same minimum. ∎

4 Impossibility proofs

This section presents three impossibility proofs for multi-objects. Following Herlihy [Her91], we first define the class of *commutative* objects, which he showed have consensus number 2. We show that multi-objects constructed out of commutative components are themselves commutative, and hence have no greater consensus number.

Next, we prove a general impossibility proof for objects built from ℓ-access-limited components (objects which can only be accessed by $\ell < n$ processes). This proof applies directly to (f,r)-*consensus* and *c-consensus* objects, which are $f+r$ and c-access-limited, respectively. Finally, we close the section with an impossibility proof specific to the $swap^m$ multi-object.

4.1 Commutative objects

A shared memory object O is *commutative* if the relative order of any two operations cannot be determined by the non-participating processes. That is, let o_p and o_q be operations offered by O to processes p and q. Let xo_po_qy be a run in the sequential specification of O, where y contains no operations of p or q. The object O is commutative if for all pairs of operations o_p and o_q, and runs xo_po_qy, there exist operations o'_p and o'_q, such that $xo'_qo'_py$ is also a run of O. Examples of commutative objects include test&set, fetch&add, fetch&complement, and all 2-access-limited objects, such as 2-*consensus*. The following is a simple property of multi-objects:

Lemma 4.1 *Any multi-object constructed of commutative component objects is commutative.*

An immediate consequence of this lemma and Herlihy's observation [Her91] is:

Theorem 7 *If X is a commutative object then the consensus number of X^m is no greater than 2.*

4.2 Impossibility proof for ℓ-access-limited objects

Next, we prove a general impossibility proof for objects built from ℓ-access-limited components. Recall that these are objects which can only be accessed by $\ell < n$ processes.

Theorem 8 *If O is a set of ℓ-access-limited objects, then $Con(O^m) < \ell\sqrt{2m}+1$.*

Proof: Note first that operations on 2-access-limited objects commute. ¿From Theorem 7, for such objects, $Con(O^m) < 2$.

Assume then that $\ell > 2$. Suppose we have an algorithm for implementing consensus among n processes, using simultaneous access to up to m copies of ℓ-access-limited objects, such as *ℓ-consensus*, where $\ell < n$.

Following [FLP85], we define a prefix of run of a consensus algorithm (and the state in which it ends) to be *univalent* with value v if no process ever returns $\neg v$ in extensions of the run. If a prefix (and state) is not univalent, we say it is *bivalent.*

In any wait-free consensus algorithm, it is always possible to find a finite bivalent run x from which any step by any process moves to a univalent state [FLP85]. Then steps by processes from the set P=$\{p_1,\ldots,p_{n_0}\}$ make x 0-valent, and steps by processes from Q=$\{q_1,\ldots,q_{n_1}\}$ make x 1-valent, and $n = n_0+n_1$. For process p_i or q_i, let P_i and Q_i respectively denote the set of objects accessed in their next step after x. Without loss of generality, assume $n_1 \geq n/2$.

Lemma 4.2

1. $|P_1 \cap Q_i| \geq (n-2)/(\ell-2)$.
2. $|P_1| \geq n(n-2)/2(\ell-1)(\ell-2)$.

Proof: Using each processor name to denote it's actions if run a single step, we have that xp_1q_i and xq_ip_1 are runs of opposite valencies, which leave every object r in the same state unless r is in $P_1 \cap Q_i$. In each of these runs there are $n-2$ processes who have not yet taken steps after x, specifically the members of $\{p_2,\ldots,p_{n_0}, q_1,\ldots,q_{i-1}q_{i+1},\ldots,q_{n_1}\}$. Denote these processes by $S=\{s_1,\ldots,s_{n-2}\}$, and let s be an arbitrary process in S.

Consider what happens when s runs after xp_1q_i or xq_ip_1. The actions taken by s will be the same in each case unless s accesses some object in $(P_1 \cap Q_i)$; since xp_1q_i or xq_ip_1 have different valencies, s must eventually access such an object. It follows that all $n-2$ members of S can access at least one object in $(P_1 \cap Q_i)$. Since each object can be accessed by at most ℓ processes, and p_1 and q_i both access each object in $(P_1 \cap Q_i)$, at least $(n-2)/(\ell-2)$ objects are each accessed by both p_1 and q_i in a single step.

There are at least $n/2$ members of Q, and each accesses at least $(n-2)/(\ell-2)$ different objects in their next step after x, making a total of at least $n(n-2)/2(\ell-2)$ distinct accesses by members of Q to objects in P_1. Each object r_j in P_1 is accessed by p_1 and at most $\ell-1$ members of Q, so there must be at least $n(n-2)/2(\ell-1)(\ell-2)$ members of P_1 to accommodate all $n(n-2)/2(\ell-2)$ distinct accesses by members of Q.

That is, $|P_1| \geq n(n-2)/2(\ell-1)(\ell-2)$. ∎

Since $m \geq |P_1|$, we have $m \geq n(n-2)/2(\ell-1)(\ell-2)$. In turn, this implies $n \leq \sqrt{2m(\ell-1)(\ell-2)+1}+1 < \ell\sqrt{2m}+1$. The Theorem follows. ∎

We notice that *c-consensus* objects are c-access-limited, and hence the upper bound on $Con(\textit{c-consensus}^m)$ follows from Theorem 8.

Corollary 9 $Con(c\text{-consensus}^m) < c\sqrt{2m}+1$.

4.3 Impossibility proof for $swap^m$

Any number of processes may invoke operations on *swap* objects. Hence, the result of the previous subsection, bounding the consensus value of ℓ-access-limited objects, cannot be usefully applied to *swap.* Nevertheless, a direct argument obtains a bound within a small constant of the $\left\lfloor\sqrt{m+\frac{5}{4}}+\frac{1}{2}\right\rfloor$ bound in Corollary 4.

Theorem 10 $Con(\text{swap}^m) \leq \frac{2}{\sqrt{3}}\sqrt{2m+4}+\frac{1}{3}$.

Proof: As in the previous theorem, to find a finite bivalent run x from which any step by any process moves to a univalent state. Then from the set P of processes $p_1,\ldots,p_{n_0}$ make x 0-valent, and a set Q of processes $q_1,\ldots,q_{n_1}$ make x 1-valent, and $n = n_0 + n_1$.

For process p_i or q_i, let P_i and Q_i respectively denote the set of objects swapped in their next step after x.

Lemma 4.3

1. $|(P_1 \cap Q_i) - \bigcup_{j=1}^{i-1} Q_j| \geq n - (i+1)$.
2. $|P_1| \geq \min_{\lceil n/2\rceil \leq n_1 < n}\ n + nn_1 - n_1^2/2 - 5n_1/2 - 1$.

Proof: Using each processor name to denote it's actions if run a single step, we have that $xp_1q_iq_{i-1}\ldots q_1$ and $xq_ip_1q_{i-1}\ldots q_1$ are runs of opposite valencies, which leave every object r in the same state unless r is in $P_1 \cap Q_i$ and not in $\bigcup_{j=1}^{i-1} Q_j$. In each of these runs there are $n-(i+1)$ processes who have not yet taken steps after x, specifically the members of $\{p_2, \ldots, p_{n_0}, q_{i+1}, \ldots, q_{n_1}\}$. Denote these processes by $s_1, \ldots, s_{n-(i+1)}$.

Consider what happens when s_1 runs after $xp_1q_iq_{i-1}\ldots q_1$ or $xq_ip_1q_{i-1}\ldots q_1$. The actions taken by s_1 will be the same in each case until s_1 first swaps some object, r_1, in $(P_1 \cap Q_i) - \bigcup_{j=1}^{i-1} Q_j$; this swap in turn overwrites the value returned to s_1. Suppose we run s_1 until just after this swap, and use $xp_1q_iq_{i-1}\ldots q_1s_1$ and $xq_ip_1q_{i-1}\ldots q_1s_1$ to denote these two runs. These runs are now indistinguishable by the processes $s_2,\ldots,s_{n-(i+1)}$ until each reads an object other than r_1 in $(P_1 \cap Q_i) - \bigcup_{j=1}^{i-1} Q_j$. By an obvious induction, each process s_{j+1} must eventually swap a distinct object $r_{j+1} \notin \{r_1, .., r_j\}$, but in $(P_1 \cap Q_i) - \bigcup_{j=1}^{i-1} Q_j$, in order to distinguish $xp_1q_iq_{i-1}\ldots q_1s_1\ldots s_j$ from $xq_ip_1q_{i-1}\ldots q_1s_1\ldots s_j$. (As above, using s_j to denote the steps taken by s_j through its first swap of r_j.) This suffices to prove the first part of the Lemma. Moreover, we have that $|P_1 \cap \bigcup_{j=1}^{n_1} Q_j| \geq \sum_{j=1}^{n_1} n-(i+1) = nn_1 - n_1 - \sum_{j=1}^{n_1} i = nn_1 - n_1^2/2 - 3n_1/2$.

Now consider the runs $xp_1q_{n_1}\ldots q_1$ and $xq_{n_1}\ldots q_1$. These runs are now indistinguishable to p_2 until it first swaps an object r_2 in $P_1 - \bigcup_{j=1}^{n_1} Q_j$. As above, suppose we run p_2 until just after this swap, and use $xp_1q_{n_1}\ldots q_1p_2$ and $xq_{n_1}\ldots q_1p_2$ to denote these two runs. These runs are now indistinguishable by the processes $p_3,\ldots,p_{n_0}$ until each reads an object other than r_2 in $[P_1 - \bigcup_{j=1}^{n_1} Q_j]$. As with the simple induction above, each process p_j, $j > 2$, must eventually swap a distinct object $r_j \notin \{r_2, \ldots, r_{j-1}\}$, but in $[P_1 - \bigcup_{j=1}^{n_1} Q_j]$, in order to distinguish $xp_1q_{n_1}\ldots q_1p_2\ldots p_{j-1}$ and $xq_{n_1}\ldots q_1p_2\ldots p_{j-1}$. Thus, there must be at least $n_0 - 1$ objects $\{r_2,\ldots,r_{n_0}\}$ in $[P_1 - \bigcup_{j=1}^{n_1} Q_j]$.

It follows that $|P_1| \geq n_0 + nn_1 - n_1^2/2 - 3n_1/2 - 1$. Using $n_0 + n_1 = n$ and $\lceil n/2 \rceil \leq n_1 < n$, we have $|P_1| \geq min_{\lceil n/2 \rceil \leq n_1 < n}\ n + nn_1 - n_1^2/2 - 5n_1/2 - 1$. ∎

Since $m \geq |P_1|$, we have $m \geq min_{\lceil n/2 \rceil \leq n_1 < n}\ n + nn_1 - n_1^2/2 - 5n_1/2 - 1$. In turn, this implies $n \leq \frac{2}{\sqrt{3}}\sqrt{2(m + \frac{25}{24})} + \frac{1}{3} < \frac{2}{\sqrt{3}}\sqrt{2m+4} + \frac{1}{3}$. The Theorem follows. ∎

5 Other combinators and objects

Other interesting object combinators, in addition to multi-object and sequential combinators, can be naturally defined as functions of canonical *sequential specifications* of the component objects as automata.

5.1 Sequential specifications via automata

Herlihy and Wing describe a method for defining shared memory objects via *sequential specifications*, sets of strings describing the object behavior when there is no interleaving of operations by different processes. These sets of strings can be specified via (Mealy) state machines, in which the state transitions are labeled by operation invocations and responses. The following definition is taken from [AGMT92]:

Definition 1 *A* sequential specification of an object *is a quintuple* $\{Q, S, I, R, \delta\}$ *where:*

Q *is a (finite or infinite) set of* states.

$S \subseteq Q$ *is a set of* initial states,

$I = (Inv_1, ..., Inv_n)$ *is an n-tuple of sets, where each* Inv_i *is a set of symbols denoting the* operation invocations *by process i. Let* $Inv = \cup_i Inv_i$.

$R = (Res_1, ..., Res_n)$ *is an n-tuple of sets, where each* Res_i *is a set of symbols denoting the* operation responses *for process i. Let* $Res = \cup_i Res_i$.

Define the set of operations by process i on $\mathcal{O}$ *to be* $Op_i = Inv_i \times Res_i$*, all the two-character strings of invocations and responses by i, and let* $Op = \cup_i Op_i$*. Then* $\delta \subset Q \times Op \times Q$ *is the* transition relation.

As an example, a *swap* object over a data domain V and accessed by n processes can be specified by a state machine with state set V, and the transition relation $\{(v, swap_i(w)return_i(v), w) \mid 1 \leq i \leq n\}$. Similarly, n-process *register* objects over V, $register_V$, can be specified with the same state set and transition relation $\{(v, write_i(w)return_i, w) \mid 1 \leq i \leq n\}, \cup \{(v, read_i return_i(v), v) \mid 1 \leq i \leq n\}$.

5.2 Examples of object combinators

The multi-object combinator can now be defined as a function of the sequential specifications of the component objects. We call two such specifications $X = \{Q_X, S_X, I_X, R_X, \delta_X\}$ and $Y = \{Q_Y, S_Y, I_Y, R_Y, \delta_Y\}$ *compatible* if $I_X \cap I_Y = \emptyset$ and $R_X \cap R_Y = \emptyset$.

Formal definition of the multi-object combinator: Let $\{X_j | j \in J\}$ be a set of compatible shared objects with index set J, where $X_j = \{Q_j, S_j, I_j, R_j, \delta_j\}$. Let I^m and R^m be the collections of subsets of $\cup_{j \in J} I_j$ and $\cup_{j \in J} R_j$, respectively, containing at most m operation invocations or responses, no more than one from each I_j or R_j, and let Op^m denote the set of operations IR, $I \in I^m$ and $R \in R^m$, such that I and R contain invocations and responses from the same subset of J. Then
$\{X_j | j \in J\}^m = \{Q^J, S^J, I^m, R^m, \delta^m\}$, where
$\delta^m = \{(\overline{u}, IR, \overline{v}) | IR \in Op^m$, and $\forall j \in J$ if I and R contain invocations and responses of X_j, then $(u_j, I_j R_j, v_j) \in \delta_j$, and $u_j = v_j$, otherwise$\}$.

Definitions of other combinators: Other operators can now be defined in terms of sequential specifications. Let $X = \{Q_X, S_X, I_X, R_X, \delta_X\}$ and $Y = \{Q_Y, S_Y, I_Y, R_Y, \delta_Y\}$ be compatible shared object specifications:

1. Standard object composition:
   $XY = \{Q_X \times Q_Y, S_X \times S_Y, I_X \cup I_Y, R_X \cup R_Y, \delta_{XY}\}$, where
   $\delta_{XY} = \{((u_X, u_Y), Op, (v_X, v_Y)) | ((u_X, Op, v_X) \in \delta_X \textit{ and } u_Y = v_Y) \textit{ or }$
   $((u_Y, Op, v_Y) \in \delta_Y \textit{ and } u_X = v_X)\}$,

2. the sequential product,
   If $Q_X = Q_Y = Q$ and $S_X = S_Y = S$, then
   $X;Y = \{Q, S, I_X \times I_Y, R_X \times R_Y, \delta_{X;Y}\}$, where
   $\delta_{X;Y} = \{(u, (inv_X, inv_Y)(res_X, res_Y), w) | (u, inv_X\, res_X, v) \in \delta_X \textit{ and } (v, inv_Y\, res_Y, w) \in \delta_Y\}$, and

3. a union combinator, $+$
   If $Q_X = Q_Y = Q$ and $S_X = S_Y = S$, then
   $X+Y = \{Q, S, I_X \cup I_Y, R_X \cup R_Y, \delta_X \cup \delta_Y\}$.

For example, for a given state space V with initial state v_0, it is natural to define the simple n-process *read* and *write* objects R_V and W_V, where the transition relation for R_V is $\{(v,\ read_i return_i(v),\ v) \mid v \in V,\ 1 \leq i \leq n\}$, and the transition relation for W_V is $\{(v,\ write_i(w) return_i,\ w) \mid v, w \in V,\ 1 \leq i \leq n\}$. Then we can construct the n-process *register* object over V, $register_V$, from R_V and W_V: $register_V = R_V + W_V$. Similarly, the *swap* object over V, $swap_V$, can be constructed from read and write objects via the sequential combinator: $R_V; W_V = swap_V$.

5.2.1 The + combinator

The + combinator is a very simple but powerful object combinator. To illustrate some of its properties, we briefly discuss some specific objects constructed with this combinator.

For example, applying the + combinator to *swap* (over integers) and *fetch&add* results in an object, *swap+fetch&add*, which supports both the *swap* and *fetch&add* operations on the same memory location.

The consensus number of *enqueue+read* is ∞: A process first enqueues its input value and then reads the queue contents and decides on the value at the head of the queue. This solves consensus for any number of processes.

An obvious upper bound of $2m-2$ on the consensus number of $(swap + read)^m$ and of $(swap{+}fetch\&add)^m$ follows from Herlihy's consensus algorithm from *m-assignment*, which shows that $Con(register^m) \geq 2m-2$ [Her91, MT94]. Since a *swap* operation is more powerful than a *write* operation, there is a direct implementation of *register* from *swap+read*, *swap+read* $\stackrel{di}{\to}$ *register*. Hence, $\{swap{+}read\}^m \rightarrow register^m$; we can simply replace any *write* operation by the corresponding *swap* operation in Herlihy's algorithm and get a consensus algorithm for $2m-2$ processes using $\{swap{+}read\}^m$ objects. A similar argument works for $\{swap{+}fetch\&add\}^m$ objects.

The above discussion gives a lower bound on $Con(swap + read)^m$, however the question of finding a bound b such that there is no consensus algorithm for more than b processes from $(swap + read)^m$ is left open. We note that the result reported in Section 3.2 ($Con(swap^m)$ $\geq \sqrt{2m+5}$) shows that $Con(\{swap + read\}^2) \geq 3$. Hence, $\{swap + read\}^2$ is strictly more powerful than $register^2$.

5.3 The multi-object *safe*^{m}

Hitherto, we have considered the multi-object combinator as applied to atomic objects. This leaves open the question of how to extend the definition to objects which do not have linearizable specifications, such as safe registers[Lam86]. Saying that a multi-safe object permits atomic access to m safe registers does not make much sense, since safe registers are not themselves atomic objects. (And in particular, $safe^1$ would be an atomic, not a safe, register.)

In the case of a single safe register, when two processes are trying to write to it at the same time, the end result is that one of the values is eventually written. A read event that happens after these two writes terminate, should return the current register value, while a read that is concurrent with a write may return any value.

Now, what should happen in the case of a multi-safe register, when two processes are trying to write concurrently to the same set of registers, say registers $r_1,\ldots,r_m$? The weakest sensible requirement seems to be that, once the write operations terminate, the value of each register r_i $(1 \leq i \leq m)$ is one of the two values that the processes tried to write into it. A stronger requirement would be that the values of all the m registers are written by the same process.

Thus, for example, if the first process tries to write "1" to all the registers and the other process tries to write "2", with the weaker requirement the value of each register after the writes is either "1" or "2", while with the stronger requirement either the values of *all* the registers are "1" or all are "2".

A simple observation is that it is possible to implement multi-objects that satisfy the weaker requirement by simply accessing the m safe registers sequentially. Thus, since the consensus number of safe registers is one, the consensus number of these trivial multi-objects is also one.

Since the weaker requirement is trivial, we choose to define multi-safe registers as satisfying the stronger requirement, and denoted them by $safe^m$, where m is the maximum number of registers that a process can access in one step. Interestingly, it can be shown that the consensus number of a $safe^m$ register is 1. (i.e., $Con(\mathrm{safe}^m) = 1$, for any $m \geq 1$.)

A *multi-regular register* is defined similarly to a multi-safe register, except that a read which overlaps with writes should return either the old value or one of the values of the concurrent writes. By similar arguments, it can be shown that the consensus number of a multi-regular register is also 1.

More precisely, consider a particular object, *regular*, which supports read and write operations, and in which pending write operations can "flicker". That is, for write operations, an internal update action takes provisional effect, may be read, and then is cancelled, by another internal abort operation that restores the appropriate value. Similarly, an internal action may provisionally read the current value. Eventually, each operation must return, making its last provisional action permanent. As in the discussion above, in operations of *regular*^{m} registers, all addresses are provisionally updated or read by a single internal action.

Using standard arguments, it is possible to show that *regular*^{m} objects have consensus number 1, by considering each of the individual atomic actions of the particular implementation just described.

Theorem 11 $Con(\text{safe}^m) = Con(\text{regular}^m) = 1$.

Proof: Briefly, consider a 2-process consensus algorithm using *regular* ^{m} objects. The standard argument shows a bivalent configuration must be reached from which any atomic action by either process reaches a univalent state. A case analysis on the possible pairs of actions concludes the proof. In particular, because any update is provisional, and could be erased by a succeeding abort, an update cannot be one of the critical steps taken by either process. ∎

This observation has an interesting consequence, given that (atomic) registers can be implemented from regular registers:

Corollary 12 *For all* $m \geq 1$, $\text{regular}^m \overset{di}{\not\to} \text{register}$.

6 Conclusions and Discussion

It is interesting to note that the number of processes that are able to read a component object X can have a larger effect on the consensus number of X^m than does the consensus number of X. For example, *register* and *register*^{m} have consensus numbers 1 and $2m-2$, respectively, while both *fetch&add* and *fetch&add*^{m} have consensus number 2. Being commutative, there is no way for two competing *fetch&add* operations to be read, while *register* objects can be read by any process.

There are several interesting questions about multi-objects that have not yet been explored. For example, what are their properties under other failure types and under failure models weaker than wait freedom? What is the impact of restricting the total number of multi-objects of a given type or the number of component objects? Does a characterization of specific simple objects and combinators, in a topological (or other) general setting, lead naturally to a characterization of their various combinations?

Although we have focused mainly on the multi-object combinator, there are other interesting combinators which could be investigated in order to better understand the nature of shared memory objects.

Finally, the ability to access multiple memory locations in one atomic step is useful in implementing concurrent lock-free data structures. For that reason, others have proposed lock-free implementations of atomic multi-word operations[AD96, AM95, HM93, IR94, ST95]. Some of this work assumes the existence of special hardware[HM93], while others[AD96, AM95, IR94, ST95], assume the existence of a strong (universal) single word operation. Our work demonstrates that some multi-objects necessarily imply the availability of such universal objects. For example, it follows from our results that implementation of an object which supports atomic operations on two queues implies the existence of a universal single word operation, while atomic access to several test-and-set objects does not.

Acknowledgment

We wish to thank Eric Ruppert, who discovered several errors in an earlier version of this paper. His comments were of great help.

References

[AAD+93] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[AD96] H. Attiya, and E. Dagan. Universal Operations: Unary versus Binary. In *Proc. of the 15th ACM Symp. on Principals of Distributed Computing*, pages 223-232, August 1996.

[AGMT92] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. *Journal of the ACM*, 42(6):1231–1274, November 1995. Also in *Proc. 11th Annual ACM Symp. on Principles of Distributed Computing*, pages 47–58, August 1992.

[AM95] J.H. Anderson, and M. Moir. Universal Constructions for Multi-Object Operations. In *Proc. of the 14th ACM Symp. on Principals of Distributed Computing*, pages 184–193, August 1995.

[And93] J. Anderson. Composite registers. *Distributed Computing*, 6, 1993.

[And94] J. Anderson. Multi-writer composite registers. *Distributed Computing*, 7, 1994.

[AS94] Y. Afek and G. Stupp. Delimiting the power of bounded size synchronization objects. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, pages 42–51, August 1994.

[AWW93] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 159–170, August 1993.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HM93] M. Herlihy and J.E B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *20th Annual Symp. on Computer Architecture*, pages 289–300, May 1993.

[HW90] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IR94] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory *Proc. of the 13th ACM Symp. on Principles of Distributed Computing*, pages 151–160, August 1994.

[Jay93] P. Jayanti. On the robustness of Herlihy's hierarchy. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 145–158, August 1993.

[Lam86] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of 6th ACM Symp. on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, April 1987.

[MT94] M. Merritt and G. Taubenfeld. Atomic *m*-register operations. *Distributed Computing*, 7:213–221, 1994. Also in *Proc. 5th Int. Workshop on Distributed Algorithms*, 1991, pages 289-294.

[Pat71] S. S. Patil. Limitations of capabilities of Dijkstra's semaphore primitives for coordination among processes. In *Project MAC computational structures group, MIT, memo 57*, 1971.

[ST95] N. Shavit, and D. Touitou. Software Transactional Memory. In *Proc. of the 14th ACM Symp. on Principals of Distributed Computing*, pages 204–213, August 1995.

[Wei94] H. Weisman. Implementing shared memory overwriting objects. Master's thesis, Tel-Aviv University, 1994.