On the robustness of Herlihy's hierarchy

Prasad Jayanti
Department of Computer Science
Cornell University
Ithaca, NY 14853

E-mail: prasad@cs.cornell.edu

Abstract

A *wait-free hierarchy* maps object types to levels in $\{1, 2, 3, \ldots\} \cup \{\infty\}$, and has the following property: if a type T is at level N, then, for all types T', there is a wait-free implementation of an object of type T', for N processes, using only registers and objects of type T. The infinite hierarchy defined by Herlihy is an example of a wait-free hierarchy. A wait-free hierarchy is *robust* if it has the following property: if a type T is at level N, and $\mathcal{S}$ is a finite set of types belonging to levels $N-1$ or lower, then there is no wait-free implementation of an object of type T, for N processes, using any number and any combination of objects belonging to the types in $\mathcal{S}$. Robustness implies that there are no clever ways of combining weak shared objects to obtain stronger ones.

Contrary to what many researchers believe, we prove that Herlihy's hierarchy is not robust. We then define some natural variants of Herlihy's hierarchy, which are also infinite wait-free hierarchies. With the exception of one, which is still open, these are not robust either. We conclude with the open question of whether a non-trivial robust wait-free hierarchies exists.

1 Introduction

A concurrent system consists of processes and shared objects. Processes are asynchronous: the rate at which a process makes progress may vary with time and is independent of the rates at which other processes make progress. Processes communicate with each other through shared objects. A process interacts with a shared object by invoking an operation on the object and receiving a response from the object. Shared objects are typed. The type of an object specifies the operations that may be invoked on the object, and, more importantly, specifies the behavior of the object in the special case when operations are applied without overlap (*i.e.*, an operation is invoked on the object only after a response is returned to the previous invocation). The latter specification is often referred to as the *sequential specification* of the type. For example, the type `binary register` specifies that an object of this type supports the operations *read*, *write 0*, and *write 1*, and has the following sequential specification: a read operation returns the most recent value written. As a second example, the type `consensus` specifies that an object of this type supports the operations *propose 0* and *propose 1*, and has the following sequential specification: every operation returns the value proposed by the first operation. `queue`, `stack`, `test&set`, and `compare&swap` are a few other examples of object types. (We will use the type-writer font for object types.)

In a concurrent system, it is possible that operations applied by different processes on the same object overlap. As noted above, the type of an object does not specify the behavior of the object in the presence of such overlapping operations. Thus, it is necessary to resort to some additional criterion in order to fully specify the behavior of an object in the presence of overlapping operations. A common criterion, and the one used in this work, is *linearizability* [HW90]. By this criterion, each operation, spanning over an interval of time from the invocation of the operation to its response, must appear to take effect at some instant in this interval.

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG-2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp.

12th ACM Symposium on Principles on Distributed Computing, Ithaca NY
© 1993 ACM 0-89791-613-1/93/0008/0145....$1.50

In most systems, simple shared objects, such as registers and test&set objects, are supported in hardware, but more complex objects, such as queues, stacks, and sets, are not. Thus, complex shared objects must be implemented in software. Traditionally, in order to ensure the linearizability of these software objects, implementations have employed semaphores and critical sections [CHP71]. However, a shared object implemented using critical-sections is not fault-tolerant: If any process crashes in a critical-section, the other processes are effectively prevented from accessing that object. To overcome this problem, Lamport advocated wait-free implementations [Lam77]. An implementation is *wait-free* if every process can complete every operation on the implemented object in a finite number of its own steps, regardless of the execution speeds of the remaining processes. Our focus in this paper is on wait-free implementations. Hereafter we write "implement" and "implementation" as shorthands for "implement in a wait-free manner" and "wait-free implementation", respectively.

How feasible are wait-free implementations? It is known that registers are too weak to implement a consensus object for even two processes [LAA87, CIL87]. Test&set objects can implement a consensus object for two processes, but not for three processes [LAA87]. A compare&swap object, on the other hand, can implement a consensus object for any number of processes [Her91b]. These results indicate that object types differ in their ability to support wait-free implementations, and that there may be a way of ordering them accordingly. This issue was addressed in a seminal paper by Herlihy [Her91b]. Listed below are an important result and a definition from [Her91b].

1. For all object types T, an object of type T can be implemented for N processes using registers and consensus objects that can be shared by N processes. This is the universality result of Herlihy.

2. The *consensus number* of a shared object $\mathcal{O}$ is the maximum number N such that a consensus object can be implemented for N processes using just $\mathcal{O}$ and (any number of) registers. If there is no such maximum, then the consensus number of $\mathcal{O}$ is ∞. Define a hierarchy of shared object types as follows: a type T is at level N if and only if the consensus number of an object of type T is N. We will refer to this hierarchy as Herlihy's hierarchy.

   (Notice that the level of a type in Herlihy's hierarchy is based on what a *single* object of this type can achieve. This will turn out to be important when we analyze the robustness of this hierarchy.)

As an obvious consequence of the universality result, Herlihy's hierarchy has the following important property: if a type T is at level N, then for all types T', an object of type T' can be implemented for N processes using just registers and objects of type T. We will call any hierarchy with this property a *wait-free hierarchy*. Notice that, by this definition, a type T can be at level N in a wait-free hierarchy even if registers and objects of type T suffice to implement objects of all types for more than N processes. In particular, if h is a wait-free hierarchy, and h' is a hierarchy in which every type is at a lower level or at the same level as in h, then h' is also a wait-free hierarchy. Thus, the level of a type in a wait-free hierarchy does not reflect the full potential of that type. This observation motivates us to define a *tight* wait-free hierarchy: it is a wait-free hierarchy with the property that if any type is elevated to a higher level, then the resulting hierarchy is not a wait-free hierarchy.

What other properties are important in a hierarchy? We argue below that robustness is one. A hierarchy is *robust* if for every type T and every finite set $\mathcal{S}$ of types, the following holds: if T is at level N and each type in $\mathcal{S}$ is at level $N-1$ or lower, then it is impossible to implement an object of type T, for N processes, using any number and any combination of objects belonging to the types in $\mathcal{S}$. Robustness guarantees that there are no clever ways of putting weak objects together to implement a strong one. The following example illustrates the significance of robustness in analyzing the power of multi-processor systems. Consider two systems $\mathcal{S}_1$ and $\mathcal{S}_2$. Suppose that $\mathcal{S}_1$ supports registers and test&set objects, and $\mathcal{S}_2$ supports registers with 3-register assignment (a process can write to any three registers in one atomic operation). Herlihy showed that arbitrary wait-free synchronization is impossible for three or more processes in $\mathcal{S}_1$, and for five or more processes in $\mathcal{S}_2$. What implications do these results have on a third system $\mathcal{S}_3$ which supports both test&set objects and registers with 3-register assignment? In particular, can we conclude, based on just the above results, that arbitrary wait-free synchronization among five processes is impossible in $\mathcal{S}_3$? We can, provided that Herlihy's hierarchy is robust. Otherwise

we cannot. More generally, if Herlihy's hierarchy is robust, the consensus number of a set of objects, belonging (possibly) to different types, is just the maximum of the consensus numbers of the individual objects in the set. Thus, if Herlihy's hierarchy is robust, the difficult problem of analyzing the combined power of a set of shared objects reduces to the simpler problem of analyzing the power of each individual object in the set.

Is Herlihy's hierarchy robust? While there is no evidence to believe one way or the other, the following facts make it plausible that the hierarchy is robust.

- Consider common types such as `register`, `test&set`, `fetch&add`, `queue`, `stack`, and `compare&swap`. The consensus number of any collection of objects of these types is the maximum of the consensus numbers of the individual objects in the collection. Thus, commonly encountered types do not provide any reason to believe in the non-robustness of Herlihy's hierarchy.

- Define a *k-process consensus object* as one which behaves like a consensus object if no more than k processes access it and behaves arbitrarily if more than k processes access it. It is impossible to implement a $(k+1)$-process consensus object using any number of registers and any number of k-process consensus objects [Her91b, JT92]. Thus, achieving consensus among $k+1$ processes is strictly harder than achieving consensus among k processes. This fact however does not imply that objects, each of which is capable of implementing only a k-process consensus object, cannot be combined to implement a $(k+1)$-process consensus object. In particular, it does not imply that Herlihy's hierarchy is robust. But it points to the plausibility of robustness.

In fact, many researchers seem to believe that Herlihy's hierarchy is robust [AGTV92, AR92, Her91a][1]. We prove that it is not. More specifically, we present a type `weak-sticky` with the property that k objects of this type, together with registers, can implement a consensus object for $k+1$ processes, but not for $k+2$ processes. In particular, one weak-sticky object, with registers, can implement a consensus object for two processes, but not for three processes. Thus, by definition, a weak-sticky object has a consensus number of 2, and consequently, the type `weak-sticky` is at level 2 in Herlihy's hierarchy. However, since multiple weak-sticky objects, with registers, can implement a consensus object for arbitrarily large number of processes, it follows from Herlihy's universality result that, for all types T and for all N, an object of type T can be implemented for N processes using just registers and weak-sticky objects. Together with the fact that `weak-sticky` is at level 2, this implies that Herlihy's wait-free hierarchy is not robust.

Does there exist a robust wait-free hierarchy? We do not know the answer yet. However, we define three natural variants of Herlihy's hierarchy, which are also infinite wait-free hierarchies. We prove that two of these are not robust.[2] The third hierarchy, whose robustness is still open, has the following property: if it is not robust, then there is no robust wait-free hierarchy. We believe that resolving the robustness of this hierarchy is an important open problem in wait-free synchronization. If it is robust, then, as already noted, the power of a system supporting multiple types of shared objects can be inferred simply from the power of each individual type. If, on the other hand, there is no robust wait-free hierarchy, then it will be possible to combine weak objects to implement strong ones. In particular, it opens up the possibility of implementing universal objects from non-universal objects!

This paper is the first to formalize and study robustness. The technical arguments involved in proving the impossibility result that k weak-sticky objects cannot implement a consensus object for $k+2$ processes are

[1] [AGTV92] states "An object has a consensus number k if k is the maximum number of processes for which the object can be used to solve the consensus problem. Thus objects with higher consensus number cannot be deterministically implemented by employing objects with lower consensus numbers."

[AR92] states "In fact, Herlihy [Her88] describes a full hierarchy of atomicity assumptions, and proves that atoms of a higher class cannot be implemented by those of a lower class, in a wait-free fashion in the deterministic setting."

[Her91a] states "Elsewhere [17, 15], we have shown that any object X can be assigned a *consensus number*, which is the largest number of processes (possibly infinite) that can achieve consensus asynchronously [13] by applying operations to a shared X. It is impossible to construct a non-blocking implementation of any object with consensus number n from objects with lower consensus numbers in a system of n or more processes, although any object with consensus number n is universal (it supports a wait-free implementation of any other object) in a system of n or fewer processes."

[2] In proving this, we show the following result which is interesting in its own right. There exist two types such that consensus among even two processes cannot be achieved using objects of either type, but consensus among any number of processes can be achieved using the two types of objects together.

novel. Traditional bivalency arguments are inadequate to prove such lower bounds.

(Due to space limitations, most proofs are omitted or only sketched. They are in [Jay93].)

2 Informal model

A concurrent system consists of asynchronous processes and shared objects. Besides a unique name, every object has two attributes: a type and a positive integer which denotes the maximum number of processes which may apply operations on that object. We say that $\mathcal{O}$ is an N-process object if N is the maximum number of processes which may apply operations on $\mathcal{O}$. The type specifies the behavior of the object when operations are applied sequentially, without overlap. More precisely, an *object type* T is a tuple (OP, RES, G), where OP and RES are sets of operations and responses respectively, and G is a directed finite or infinite multi-graph in which each edge has a label of the form (op, res) where $op \in OP$ and $res \in RES$. We refer to G as the *sequential specification* of T, and the vertices of G as the *states* of T. Intuitively, if there is an edge, labeled (op, res), from state σ to state σ', it means that applying the operation op to an object in state σ may change the state to σ' and return the response res.

An N-process object $\mathcal{O}$ of type T supports the set of procedures $\texttt{Apply}(P_i, op, \mathcal{O})$, for all $1 \leq i \leq N$ and $op \in OP(T)$. A process P *invokes* operation op on object $\mathcal{O}$ by calling $\texttt{Apply}(P, op, \mathcal{O})$, and *executes* the operation by executing this procedure. The operation *completes* when the procedure terminates. The *response* for an operation is the value returned by the procedure.

The type of an object, by itself, is not sufficient to characterize the behavior of the object in the presence of concurrent operations. To characterize such behavior, we use the concept of *linearizability* [HW90]. Roughly speaking, linearizability requires every operation execution to appear to take effect instantaneously at some point in time between its invocation and response.

Let T be an object type and $\mathcal{L} = (T_1, T_2, \ldots)$ be a (possibly infinite) list of (not necessarily distinct) object types. Let $\Sigma = (\sigma_1, \sigma_2, \ldots)$ be a list where σ_i is a state of type T_i. An *implementation of T, initialized to state σ, from $(\mathcal{L}, \Sigma)$ for N processes* is a function $\mathcal{I}(O_1, O_2, \ldots)$ such that if $O_1, O_2, \ldots$ are N-process objects of type $T_1, T_2, \ldots$, initialized to states $\sigma_1, \sigma_2, \ldots$, respectively, then $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots)$ is an N-process object of type T, initialized to σ. Intuitively, $\mathcal{I}(O_1, O_2, \ldots)$ returns a set of procedures $\texttt{Apply}(P_i, op, \mathcal{O})$, for $1 \leq i \leq N$ and $op \in OP(T)$. $\texttt{Apply}(P_i, op, \mathcal{O})$ specifies how process P_i should "simulate" the operation op on $\mathcal{O}$ in terms of operations on $O_1, O_2, \ldots$. We say $\mathcal{O}$ is a *derived object* of the implementation $\mathcal{I}$, and $O_1, O_2, \ldots, O_n$ are the *base objects* of $\mathcal{O}$.

We say that *$\mathcal{I}$ is an implementation of T, initialized to state σ, from a set $\mathcal{S}$ of types for N processes* if there is a list $\mathcal{L} = (T_1, T_2, \ldots)$ of types and a list $\Sigma = (\sigma_1, \sigma_2, \ldots)$ of states such that $T_i \in \mathcal{S}$, σ_i is a state of T_i, and $\mathcal{I}$ is an implementation of T, initialized to σ, from $(\mathcal{L}, \Sigma)$ for N processes. We say that *a type T has an implementation from a set $\mathcal{S}$ of types for N processes* if, for all states σ of T, there is an implementation of T, initialized to σ, from $\mathcal{S}$ for N processes.

An *implementation is wait-free* if every process completes every operation on a derived object in a finite number of operations on the base objects, regardless of the execution speeds of the remaining processes in the system. Hereafter we write "implementation" as a shorthand for "wait-free implementation".

3 Hierarchy Preliminaries

A *hierarchy of shared types* is a function that maps object types to levels in $\{1, 2, 3, \ldots\} \cup \{\infty\}$. An object type T is at level l in hierarchy h if $h(T) = l$. A hierarchy is *non-trivial* if it has at least two non-empty levels. An object type T is *universal for N processes* if for every type T', there is an implementation of T' from $\{T, \texttt{register}\}$ for N processes. T *is universal* if, for all N, T is universal for N processes. A hierarchy h is a *wait-free hierarchy* if, for all T, $h(T) = N$ implies that T is universal for N processes. The following proposition is immediate from the definition.

Proposition 3.1 *If h is a wait-free hierarchy, and h' is a hierarchy such that $\forall T : h'(T) \leq h(T)$, then h' is a wait-free hierarchy.*

Proposition 3.2 *If h is a wait-free hierarchy, then $h(\texttt{register}) = 1$. Thus, level 1 of any wait-free hierarchy is non-empty.*

From Proposition 3.1, it is clear that there can be "slack" in a wait-free hierarchy: a type T can be at some level N although T is universal for $M > N$ processes. This motivates us to define tightness. A wait-free hierarchy h is *tight* if, for all wait-free hierarchies h' and for all types T, $h(T) \geq h'(T)$. A wait-free hierarchy h is *fully-refined* if, for all levels $k \in \{1, 2, 3, \ldots\} \cup \{\infty\}$, there is some type at level k of the hierarchy h. A wait-free hierarchy h is *robust* if, for all types T and all finite sets $\mathcal{S}$ of types, the following holds: if $h(T) = N$ and $\forall T' \in \mathcal{S} : h(T') < N$, then there is no implementation of T from $\mathcal{S}$ for N processes. The reader should note the difference between tightness and robustness. The trivial wait-free hierarchy which maps every object type to level 1 is obviously robust, but not tight. The wait-free hierarchy $\mathtt{h}_\mathtt{m}^\mathtt{r}$ (to be defined soon) is tight, but it is not known whether it is robust.

In the remainder of this section, we define some natural wait-free hierarchies, and highlight some simple properties of these hierarchies. In the following definitions, the subscript indicates whether the definition allows just 1 or many objects of the argument type. The superscript r indicates that the definition allows the use of registers.

1. $\mathtt{h}_\mathtt{1}(T)$ = maximum number of processes for which a consensus object can be implemented using just a single object of type T. If there is no such maximum, then $\mathtt{h}_\mathtt{1}(T) = \infty$.

2. $\mathtt{h}_\mathtt{1}^\mathtt{r}(T)$ = maximum number of processes for which a consensus object can be implemented using just a single object of type T and any number of registers. If there is no such maximum, then $\mathtt{h}_\mathtt{1}^\mathtt{r}(T) = \infty$.

   Notice that this is Herlihy's hierarchy [Her91b].

3. $\mathtt{h}_\mathtt{m}(T)$ = maximum number of processes for which a consensus object can be implemented using any number of objects of type T. If there is no such maximum, then $\mathtt{h}_\mathtt{m}(T) = \infty$.

4. $\mathtt{h}_\mathtt{m}^\mathtt{r}(T)$ = maximum number of processes for which a consensus object can be implemented using any number of objects of type T and any number of registers. If there is no such maximum, then $\mathtt{h}_\mathtt{m}^\mathtt{r}(T) = \infty$.

Proposition 3.3 *Each of* $\mathtt{h}_\mathtt{1}, \mathtt{h}_\mathtt{1}^\mathtt{r}, \mathtt{h}_\mathtt{m}, \mathtt{h}_\mathtt{m}^\mathtt{r}$ *is a fully-refined wait-free hierarchy.*

Proposition 3.4 $\mathtt{h}_\mathtt{m}^\mathtt{r}(T) = N < \infty$ *if and only if* T *is universal for* N *processes, but not for* $N+1$ *processes.* $\mathtt{h}_\mathtt{m}^\mathtt{r}(T) = \infty$ *if and only if* T *is universal.*

Proposition 3.5 *If* h *is a tight wait-free hierarchy, then* $h = \mathtt{h}_\mathtt{m}^\mathtt{r}$. *In other words,* $\mathtt{h}_\mathtt{m}^\mathtt{r}$ *is the unique wait-free hierarchy which is tight.*

The hierarchy $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is uniquely important in the study of robust wait-free hierarchies. To formally state this, we need a definition. Let $\sigma = (l_1, l_2, \ldots)$ be a finite/infinite sequence such that $1 = l_1 < l_2 < l_3 \ldots$ and $l_i \in \{1, 2, 3, \ldots\} \cup \{\infty\}$. We say that a hierarchy g is a *coarsening of hierarchy* h *with respect to* σ if, for all object types T, we have:

1. If $l_i \leq h(T) < l_{i+1}$, then $g(T) = l_i$.
2. If $l_i \leq h(T)$ and l_i is the last element of σ, then $g(T) = l_i$.
3. If $h(T) = \infty$ and σ is infinite, then $g(T) = \infty$.

Intuitively, levels $l_i \ldots (l_{i+1} - 1)$ in h are lumped into level l_i of g, causing levels $(l_i + 1) \ldots (l_{i+1} - 1)$ to be empty in g. We say that a hierarchy g is a *coarsening* of hierarchy h if there is a σ of the form $1 = l_1 < l_2 < l_3 \ldots$ such that g is a coarsening of h with respect to σ. It is obvious that if h is a wait-free hierarchy, so is every coarsening of h.

Theorem 3.1 *If* h *is a robust wait-free hierarchy, then* h *is a coarsening of* $\mathtt{h}_\mathtt{m}^\mathtt{r}$.

Proof Assume that h is a robust wait-free hierarchy, and is not a coarsening of $\mathtt{h}_\mathtt{m}^\mathtt{r}$. Let $\sigma = (l_1, l_2, \ldots)$, where $1 = l_1 < l_2 < l_3 \ldots$ are all the non-empty levels of h. Define g to be the coarsening of $\mathtt{h}_\mathtt{m}^\mathtt{r}$ with respect to σ. From our assumption that h is not a coarsening of $\mathtt{h}_\mathtt{m}^\mathtt{r}$, it follows that $h \neq g$. Thus, there is a type T such that $h(T) \neq g(T)$. Let $m = h(T)$ and $n = g(T)$. By definition of g, a level k of g is non-empty if and only if level k of h is non-empty. Together with $m \neq n$, this implies that there exist types T' and T'', each different from T, such that $g(T') = m$ and $h(T'') = n$. Since $m \neq n$, we are left with two cases to consider.

1. $m < n$.

   Since g is a coarsening of $\mathtt{h}_\mathtt{m}^\mathtt{r}$ and $g(T) = n$, it follows that $\mathtt{h}_\mathtt{m}^\mathtt{r}(T) \geq n$. Thus, by Proposition 3.4, T is universal for n processes. In particular, there is an implementation of T'' from $\{T, \mathtt{register}\}$

for n processes. Since $h(T) = m < n = h(T'')$, h is not robust. This is a contradiction.

2. $m > n$.

   From the above, $g(T') = m$. Thus, level m of g is not empty. This, together with $m > n$, implies that $n \leq \mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(T) < m$. This implies, by Proposition 3.4, that T is not universal for m processes. Since $h(T) = m$, it follows that h is not a wait-free hierarchy. This is a contradiction. □

What can we say about the robustness of $\mathtt{h}_1, \mathtt{h}^{\mathtt{r}}_1$, and $\mathtt{h}_\mathtt{m}$? This question is addressed by the following proposition, which follows from Theorem 3.1 and Propositions 3.5 and 3.3.

Proposition 3.6 *Let $h \in \{\mathtt{h}_1, \mathtt{h}^{\mathtt{r}}_1, \mathtt{h}_\mathtt{m}\}$. If $h \neq \mathtt{h}^{\mathtt{r}}_\mathtt{m}$, then h is neither tight nor robust.*

Does one of $\mathtt{h}_1, \mathtt{h}^{\mathtt{r}}_1$, and $\mathtt{h}_\mathtt{m}$ define the same hierarchy as $\mathtt{h}^{\mathtt{r}}_\mathtt{m}$? The answer is not easy. For instance, $\mathtt{h}^{\mathtt{r}}_1$ differs from $\mathtt{h}^{\mathtt{r}}_\mathtt{m}$ if and only if there is a type such that multiple objects of this type (together with registers) can solve consensus among a larger number of processes than a single object (together with registers) can. Does such a type exist? No common object type exhibits such a property and, hence, it is a non-trivial question. Similarly, $\mathtt{h}_\mathtt{m}$ differs from $\mathtt{h}^{\mathtt{r}}_\mathtt{m}$ if and only if there is a type such that the use of registers increases the number of processes for which consensus can be solved using objects of this type. Again, common object types do not exhibit this property, making it difficult to answer whether such types exist.

In the rest of the paper, we prove that each of $\mathtt{h}_1, \mathtt{h}^{\mathtt{r}}_1$, and $\mathtt{h}_\mathtt{m}$ differs from $\mathtt{h}^{\mathtt{r}}_\mathtt{m}$. Thus, none of $\mathtt{h}_1, \mathtt{h}^{\mathtt{r}}_1$, and $\mathtt{h}_\mathtt{m}$ is robust. In particular, $\mathtt{h}^{\mathtt{r}}_1$, which is the same as Herlihy's wait-free hierarchy, is not robust. Unfortunately, we do not yet know whether $\mathtt{h}^{\mathtt{r}}_\mathtt{m}$ or some coarsening of it is robust. This is an important open question. We hope that the ideas employed in this paper would provide useful insights.

4 On the robustness of Herlihy's hierarchy ($\mathtt{h}^{\mathtt{r}}_1$)

The main result of this section is that $\mathtt{h}^{\mathtt{r}}_1$ is not robust. We prove this result by presenting an object type `weak-sticky` with the following property: n weak-sticky objects, together with registers, can implement a consensus object for $n+1$ processes, but not for $n+2$ processes. This implies $\mathtt{h}^{\mathtt{r}}_1(\mathtt{weak\text{-}sticky}) = 2$ and $\mathtt{h}^{\mathtt{r}}_\mathtt{m}(\mathtt{weak\text{-}sticky}) = \infty$. Thus, $\mathtt{h}^{\mathtt{r}}_1 \neq \mathtt{h}^{\mathtt{r}}_\mathtt{m}$ and, by Proposition 3.6, $\mathtt{h}^{\mathtt{r}}_1$ is not robust.

Consider the object type `sticky` in Figure 1. It supports two operations, *L-op* and *R-op*, and responds with either *L-first* or *R-first*. If *L-op* is applied on a sticky object $\mathcal{O}$, initialized to state $S_\perp$, $\mathcal{O}$ changes state to S_L and returns *L-first* as the response. Furthermore, $\mathcal{O}$ returns *L-first* to all subsequent operations, reflecting the fact that *L-op* was the first operation applied on $\mathcal{O}$. The behavior is symmetric if, instead of *L-op*, *R-op* was the first operation applied on $\mathcal{O}$. In essence, the first operation "sticks" to $\mathcal{O}$ and determines the response for all operations. Notice that `sticky` is similar to the `consensus` [Her91b] and `sticky-bit` [Plo89] object types.

Now consider the type `weak-sticky`, a variant of `sticky`, shown in Figure 2. `weak-sticky` lacks the symmetry of `sticky`: If *R-op* is applied to a weak-sticky object $\mathcal{O}$, initialized to $S_\perp$, *R-op* sticks to $\mathcal{O}$ as before. However, as soon as *R-op* is applied for the second time, it "unsticks" and $\mathcal{O}$ starts behaving as though it had been stuck with *L-op* all along.

Let $\mathcal{I}_j$ denote the implementation of `consensus` from {`weak-sticky`, `register`} for processes P_1,P_2, ..., P_j. The implementation is recursive. The base case is to derive $\mathcal{I}_1$, implementation of `consensus` for the single process P_1, and is trivial: if $\mathcal{O}_1$ is a derived object of $\mathcal{I}_1$, `Apply`(P_1, *propose* v_1, $\mathcal{O}_1$) simply returns v_1. The recursive step of deriving $\mathcal{I}_n$ from $\mathcal{I}_{n-1}$ is presented in Figure 3.

Lemma 4.1 *The implementation $\mathcal{I}_n$ in Figure 3 is a correct implementation of* `consensus` *from* {`weak-sticky`,`register`} *for processes $P_1, P_2, \ldots, P_n$. $\mathcal{I}_n$ requires $(n-1)$ weak-sticky objects and $2(n-1)$ registers.*

Corollary 4.1 $\mathtt{h}^{\mathtt{r}}_\mathtt{m}(\mathtt{weak\text{-}sticky}) = \infty$.

Next we prove a lower bound: any solution to n-process wait-free consensus using weak-sticky objects and registers requires at least $n-1$ weak-sticky objects, regardless of how many registers are available. We prove this result by reducing the problem of "achieving consensus among n processes (not necessarily in a wait-free manner) when processes may communicate only via registers and at most one of the processes

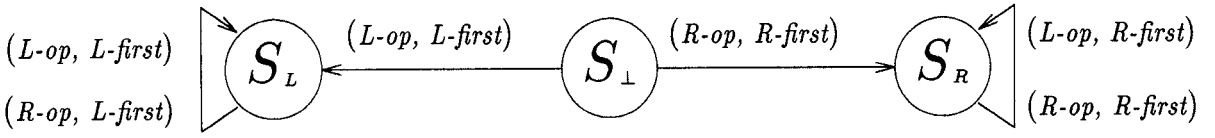

Figure 1: Object type `sticky`

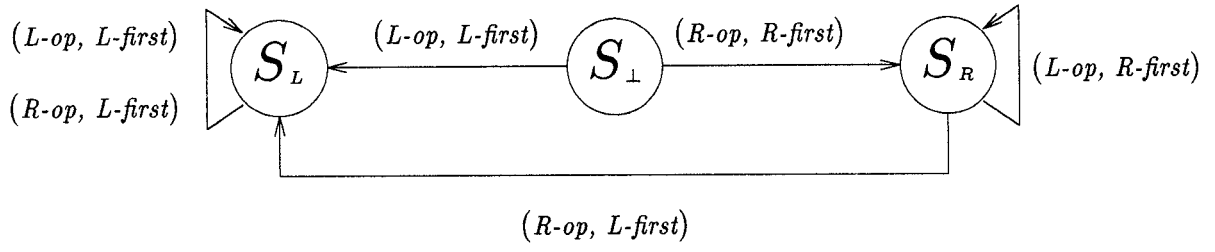

Figure 2: Object type `weak-sticky`

$\mathcal{O}_{n-1}$: consensus object for $P_1, P_2, \ldots, P_{n-1}$, derived from $\mathcal{I}_{n-1}$
O_{ws}: weak-sticky object, initialized to $S_\perp$
L, R: binary registers

Apply(P_i, *propose* v_i, $\mathcal{O}_n$) (for $1 \leq i \leq n-1$)

1. $L :=$ Apply(P_i, *propose* v_i, $\mathcal{O}_{n-1}$)
2. if Apply(P_i, *L-op*, O_{ws}) = *L-first*
3. return(L)
4. else return(R)

Apply(P_n, *propose* v_n, $\mathcal{O}_n$)

$R := v_n$
if Apply(P_n, *R-op*, O_{ws}) = *L-first*
return(L)
else return(R)

Figure 3: Implementing `consensus` from {`weak-sticky`, `register`}

may crash" to the problem of "achieving wait-free consensus among n processes communicating via registers and $(n-2)$ weak-sticky objects". The former problem is impossible to solve [LAA87, DDS87]. Hence the impossibility of the latter. The reduction is based on the novel concept of *k-trap implementations.*

An implementation for processes $P_1, P_2, \ldots, P_n$ is a *k-trap implementation* if every derived object $\mathcal{O}$ of the implementation has the following property: in any execution of $(P_1, P_2, \ldots, P_n; \mathcal{O})$, regardless of the relative execution speeds of processes, all but up to k correct processes will be able to eventually complete their operations on $\mathcal{O}$. In other words, at most k correct processes get blocked while accessing $\mathcal{O}$. Notice that a 0-trap implementation is the same as a wait-free implementation. The following lemma establishes the utility of k-trap implementations in proving lower-bounds.

Lemma 4.2 *Let* T *be any object type such that for every state σ of* T*, there is a 1-trap implementation $\mathcal{I}_\sigma$ of* T*, initialized to σ, from* `register` *for n processes. Then, any wait-free implementation of* `consensus` *from* {T,`register`} *for n processes requires at least $n-1$ objects of type* T *(regardless of how many registers it uses).*

Proof Suppose that the lemma is false, and there is a wait-free implementation $\mathcal{J}$ of `consensus` from {T,`register`} for n processes such that $\mathcal{J}$ requires only $n-2$ objects of type T, initialized to some states $\sigma_1, \sigma_2, \ldots, \sigma_{n-2}$ of T, and m registers (for some $m \geq 0$). Consider the protocol $\mathcal{P}$ in Figure 4. Clearly, processes communicate exclusively via registers in protocol $\mathcal{P}$. We argue below that $\mathcal{P}$ solves the consensus problem for processes $P_1, P_2, \ldots, P_n$ even if at most one of the processes may crash. By the impossibility result in [LAA87, DDS87], such a protocol does not exist. Hence the lemma.

We claim that at most $(n-2)$ processes block on $\mathcal{O}$. This follows from the following facts:

1. $n-2$ base objects of $\mathcal{O}$ are 1-trap. So at most one process blocks on each of these.
2. No process blocks on the remaining base objects of $\mathcal{O}$, the registers $R_1, R_2, \ldots, R_m$.
3. $\mathcal{O}$ is derived from a wait-free implementation.

---

1. For $1 \leq i \leq n-2$, use $\mathcal{I}_{\sigma_i}$ to implement an object O_i of type T initialized to state σ_i.
2. Use $\mathcal{J}$ to implement a consensus object $\mathcal{O}$ from $O_1, O_2, \ldots, O_{n-2}$ and registers $R_1, R_2, \ldots, R_m$.
3. Let D be a 3-valued register initialized to $\perp$.
4. For $1 \leq i \leq n$, let v_i be the binary value that process P_i wishes to propose for consensus. Process P_i executes the procedure `Apply`(P_i, *propose* v_i, $\mathcal{O}$) and writes the return value in register D. As P_i executes this procedure, after each step of the procedure, P_i reads the value in D and if it is not $\perp$, decides this value and terminates.

Figure 4: 1-resilient consensus protocol $\mathcal{P}$ for n processes

---

Therefore, if at most one of $P_1, P_2, \ldots, P_n$ crashes, there is still one process, call it P_k, that neither crashes nor blocks on $\mathcal{O}$. This process P_k eventually writes the response, call it V, returned by `Apply`(P_k, *propose* v_k, $\mathcal{O}$) in register D. Since $\mathcal{O}$ is a consensus object, it follows that $V \in \{v_1, v_2, \ldots, v_n\}$ and no process ever writes a value different from V in register D. The protocol in Figure 4 ensures that every non-crashing process eventually reads V and decides V. In other words, $\mathcal{P}$ solves the consensus problem for $P_1, P_2, \ldots, P_n$ even if at most a single process may crash. □

Recall that `weak-sticky` has three states - $S_\perp, S_L$, and S_R. We now present a 1-trap implementation of `weak-sticky` initialized to $S_\perp$, and 0-trap implementations of `weak-sticky` initialized to S_L or S_R. These implementations use only registers as base objects. Thus, by Lemma 4.2, we have the desired lower bound.

Lemma 4.3 *Figure 5 presents a 1-trap implementation of* `weak-sticky`*, initialized to $S_\perp$, from* `register` *for processes $P_1, P_2, \ldots, P_n$.*

Proof Sketch This implementation is subtle and is based on the observation that if the first *R-op* operation is blocked, then all other (*R-op* and *L-op*) operations can legitimately return *L-first.* An informal proof of correctness is given below.

Consider $\mathcal{O}$, a weak-sticky object derived from this implementation. Let H be a history of $\mathcal{O}$, and let *first-op* denote the first operation to complete in H.

$R[1 \ldots n]$: binary (1-writer, n-reader) registers initialized to 0

Apply(P_i, *L-op*, $\mathcal{O}$)

return(*L-first*)

Apply(P_i, *R-op*, $\mathcal{O}$)

1. if $(\forall k : R[k]= 0)$ then
2. $\quad R[i] := 1$
3. $\quad$ repeat until $(\exists j < i : R[j]= 1)$
4. return(*L-first*)

Figure 5: 1-trap implementation of `weak-sticky`, initialized to $S_\perp$, from `register`

There are two cases. Case (1) corresponds to *first-op* being an *L-op* operation. Consider the linearization S which includes only the complete operations in H and sequences them in the order of their completion times. Thus, *first-op*, which is an *L-op* operation, becomes the the first operation in S. Furthermore, the response of every operation in S is *L-first* (this is obvious from the implementation). From the sequential specification of `weak-sticky` in Figure 2, it is obvious that S is legal. Now consider Case (2), which corresponds to *first-op* being an *R-op* operation. The key observation is that if *first-op*, which is an *R-op* operation, completed in H, then by our implementation, there must be another *R-op* operation, call it *blocked-op*, from a different process which is concurrent with *first-op* and is blocked. Let us pretend that, although incomplete, *blocked-op* has indeed taken effect in H, and has *R-first* for its response. Consider the linearization S which sequences *blocked-op* first, *first-op* second, and the remaining complete operations in H in the order of their completion times. (*blocked-op* can be linearized before *first-op* since these two operations are concurrent.) Thus the first operation in the linearization S is a *R-op* operation with *R-first* as the associated response. The second operation in the linearization is also an *R-op* operation, and has *L-first* as the associated response. The remaining operations in the linearization have *L-first* as their response. From the sequential specification of `weak-sticky` in Figure 2, it is obvious that this linearization S is legal. Hence the correctness of our implementation. Furthermore, it is obvious from the implementation that at most one process gets blocked. Thus, the implementation is 1-trap. Hence the lemma. □

Lemma 4.4 *Figure 6 presents a 0-trap (wait-free) implementation of* `weak-sticky`*, initialized to S_R, from* `register` *for processes $P_1, P_2, \ldots, P_n$.*

Lemma 4.5 *There is a 0-trap (wait-free) implementation of* `weak-sticky`*, initialized to S_L, from* `register` *for processes $P_1, P_2, \ldots, P_n$. (This implementation is trivial and is therefore omitted.)*

Lemma 4.6 *Any wait-free implementation of* `consensus` *from* {`weak-sticky`, `register`} *for n processes requires at least $n - 1$ objects of type* `weak-sticky`.

Proof Follows from Lemmas 4.2, 4.3, 4.4, and 4.5. □

Corollary 4.2 $\mathtt{h}_1^{\mathtt{r}}(\mathtt{weak\text{-}sticky}) = 2$.

Proof Follows from Lemmas 4.1 and 4.6. □

Theorem 4.1 $\mathtt{h}_1^{\mathtt{r}}$ *is neither tight nor robust.*

Proof Follows from Proposition 3.6 and Corollaries 4.1 and 4.2. □

Corollary 4.3 $\mathtt{h}_1$ *is neither tight nor robust.*

Proof From the definitions of $\mathtt{h}_1$ and $\mathtt{h}_1^{\mathtt{r}}$, it is obvious that, for all types T, $\mathtt{h}_1(T) \leq \mathtt{h}_1^{\mathtt{r}}(T)$. In particular, $\mathtt{h}_1(\mathtt{weak\text{-}sticky}) \leq \mathtt{h}_1^{\mathtt{r}}(\mathtt{weak\text{-}sticky}) = 2$. Thus, by Corollary 4.1, $\mathtt{h}_1 \neq \mathtt{h}_m^{\mathtt{r}}$. It follows from Proposition 3.6 that $\mathtt{h}_1$ is neither tight nor robust. □

Independently, Kleinberg and Mullainathan provide a direct proof that $\mathtt{h}_1$ is not robust [KM].

R: binary register, initialized to 0

Apply(P_i, L-*op*, $\mathcal{O}$)

if ($R = 0$) then
    return(R-*first*)
else return(L-*first*)

Apply(P_i, R-*op*, $\mathcal{O}$)

$R := 1$
return(L-*first*)

Figure 6: 0-trap implementation of `weak-sticky`, initialized to S_R, from `register`

5 On the robustness of h_m

The main result of this section is that h_m is not robust. We prove this result by presenting an infinite family of object types, named DAD(k), $k \in \{2, 3, 4, \ldots\} \cup \{\infty\}$, with the following properties[3]:

1. There is an implementation of `consensus` from {DAD(k), `register`} for k processes, but not for $k + 1$ processes.
2. There is no implementation of `consensus` from DAD(k) for two processes.

Property (1) implies that $h_m^r(\text{DAD}(k)) = k$. Property (2) implies that $h_m(\text{DAD}(k)) = 1$. Thus, $h_m \neq h_m^r$ and, by Proposition 3.6, h_m is not robust.[4] This result is significant in the following sense. Registers by themselves are too weak to solve consensus even between two processes. So are DAD(∞) objects. Using these two types together, however, lets us solve consensus among any number of processes!

The object type DAD(k) is specified in Figure 7. In this specification, *choose*(S) chooses an element from set S non-deterministically and returns it. Notice that *upset* and *ahead*[i] are stable: once true, they remain true. Similarly, once *decision* $\in \{0, 1\}$, it does not change.

We first show, for $k \in \{2, 3, \ldots\} \cup \{\infty\}$, how to implement a consensus object for k processes using only DAD(k) objects and registers. Our implementation is recursive. Let $\mathcal{I}_n^k$ denote the implementation of `consensus` from {DAD(k), `register`} for processes $P_1, P_2, \ldots, P_n$. The base case is to derive $\mathcal{I}_0^k$, implementation of `consensus` for an empty set of processes, and is vacuous. The recursive step of deriving $\mathcal{I}_n^k$ from $\mathcal{I}_{n-1}^k$ is presented in Figure 8.

The implementation $\mathcal{I}_n^k$ works as follows. Processes $P_1 \ldots P_n$ split into two groups, G_0 and G_1. Group G_0 has $P_1 \ldots P_{n-1}$, and group G_1 has just P_n. Processes $P_1 \ldots P_{n-1}$ do consensus among themselves (recursively) and announce the outcome in $R[0]$. Process P_n announces its proposal in $R[1]$. The rest of the protocol resolves which of the two groups is the winner. If G_0 wins, every process decides the value in $R[0]$. Similarly, if G_1 wins, every process decides the value in $R[1]$. The object O_{dad} is used to determine the winner of the two groups. Processes $P_1 \ldots P_{n-1}$ perform the operation op(0) on O_{dad}. Then they set the register $R'[0]$ to inform process P_n that op(0) has been executed on O_{dad}. Process P_n, on the other hand, performs op(1) on O_{dad}, and then sets $R'[1]$ to inform processes in G_0 that op(1) has been executed. Processes then perform the `give-decision` operation. The return value determines the winning group. For this strategy to work correctly, the arguments of the `give-decision` operation must be such that the object O_{dad} does not get upset. We urge the reader to understand how the registers $R'[0..1]$ are used to ensure that O_{dad} does not get upset. Finally, if O_{dad} returns v, a process assumes that the group G_v won and decides the value in $R[v]$.

Lemma 5.1 *For $1 \leq n \leq k$, the implementation $\mathcal{I}_n^k$ in Figure 8 is a correct implementation of* `consensus` *from* {DAD(k), `register`} *for processes $P_1, P_2, \ldots, P_n$.*

[3] DAD is an abbreviation for `disciplined-access demanding`, the object type specified in Figure 7.

[4] A single member of the DAD(k) family is sufficient to establish that h_m is not robust. The existence of an entire family shows that there is not even a coarsening of h_m which is non-trivial and robust.

S1. DAD(k) supports operations in $\{\texttt{op}(i)|i = \{0,1\}\} \cup \{\texttt{give-decision}(i,b)|i \in \{0,1\}, b \in \{true, false\}\}$.

S2. The response for op(0) or op(1) is always *ack*. The response for give-decision$(-,-)$ is either 0 or 1.

S3. The state of DAD(k) is represented by the variables n_0, n_1, n_{gd} : integer; *decision* $\in \{\perp, 0, 1\}$; *ahead*[0..1], *upset* : boolean. Informally, n_0, n_1, n_{gd} count the number of executions of op(0), op(1), and give-decision, respectively. The variable *ahead*[i] is set to *true* if $n_i > 0$ and $n_{\bar{i}} = 0$ when give-decision$(i,-)$ is executed. The variable *upset* is set to *true* if one of the following happens: (i) op(1) is executed more than once (op(0) may be executed any number of times without upsetting a DAD(k) object); (ii) give-decision is executed more than k times; (iii) give-decision$(i,-)$ is executed with no prior execution of op(i); (iv) give-decision$(i, true)$ is executed with no prior execution of op($\bar{i}$); (v) give-decision$(i, false)$ is executed and *ahead* $[\bar{i}] = true$. If upset, a DAD(k) object returns 0 or 1 non-deterministically to an invocation of give-decision. If not upset, it sets *decision* irrevocably and non-deterministically (if not already set) to 0 or 1 such that $n_{decision} > 0$, and returns *decision*. See **S5** below for a formal sequential specification of DAD(k).

S4. The state of DAD(k) corresponding to ($n_0 = n_1 = n_{gd} = 0$; *decision* $= \perp$; *ahead*[0..1] $=$ *upset* $= false$) is known as the *fresh state*. The states of DAD(k) are *only* those that are reachable from the fresh state by the following specification.

S5. The sequential specification of DAD(k) is as follows:

```
op(i)                                   /* i ∈ {0,1} */
    n_i := n_i + 1
    if n_1 > 1 then upset := true
    return(ack)

give-decision(i, other-is-ahead)        /* i ∈ {0,1}, other-is-ahead: boolean */
    n_gd := n_gd + 1
    if (n_i > 0 ∧ n_ī = 0) then ahead[i] := true
    if (n_gd > k) ∨ (n_i = 0) ∨ (ahead[ī] ∧ ¬other-is-ahead) ∨ (n_ī = 0 ∧ other-is-ahead) then
        upset := true
    if upset then
        return(choose({0,1}))
    else if decision = ⊥ then
        decision := choose({j|n_j > 0})
    return(decision)
```

Figure 7: Object type DAD(k)

base objects of the implementation $\mathcal{I}_n^k$

$\mathcal{O}_{n-1}$: consensus object for $P_1, P_2, \ldots, P_{n-1}$, derived from $\mathcal{I}_{n-1}^k$
O_{dad}: DAD(k) object, initialized to the fresh state
R[0..1]: binary registers
R'[0..1]: boolean registers, initialized to *false*

local variables of process P_i

$d_i, winner_i \in \{0,1\}$
$other\text{-}ahead_i$: boolean

Apply(P_i, *propose* v_i, $\mathcal{O}_n$) (for $1 \leq i \leq n-1$)

1. $d_i :=$ Apply(P_i, *propose* v_i, $\mathcal{O}_{n-1}$)
2. $R[0] := d_i$
3. Apply(P_i, op(0), O_{dad})
4. $R'[0] := true$
5. $other\text{-}ahead_i := R'[1]$
6. $winner_i :=$ Apply(P_i, give-decision(0, $other\text{-}ahead_i$), O_{dad})
7. return($R[winner_i]$)

Apply(P_n, *propose* v_n, $\mathcal{O}_n$)

$d_n := v_n$
$R[1] := d_n$
Apply(P_n, op(1), O_{dad})
$R'[1] := true$
$other\text{-}ahead_n := R'[0]$
$winner_n :=$ Apply(P_n, give-decision(1, $other\text{-}ahead_n$), O_{dad})
return($R[winner_n]$)

Figure 8: Implementing consensus from {DAD(k), register}

Our next result is that DAD(k) objects and registers do not suffice to implement a consensus object for $k+1$ processes. This impossibility result follows from a straight forward bivalency argument.

Lemma 5.2 *There is no implementation of* consensus *from* {DAD(k), register} *for* $k+1$ *processes.*

Corollary 5.1 $h_m^r(\text{DAD}(k)) = k$.

The next lemma states that it is impossible to implement a consensus object for two processes using just DAD(k) objects. Intuitively, DAD(k) objects are so weak that a process cannot use these objects to leave its "foot marks" behind. Thus, if a process P_0 runs to completion and decides before a second process P_1 starts to run, P_1 cannot know that P_0 ran before it started. This can cause P_1 to decide a value which is not consistent with the decision of P_0. The proof formalizes this argument and is omitted. The details are subtle due to the non-determinism of the DAD(k) objects.

Lemma 5.3 $h_m(\text{DAD}(k)) = 1$.

Theorem 5.1 h_m *is neither tight nor robust.*

Proof Follows from Proposition 3.6, Corollary 5.1, and Lemma 5.3. □

6 Conclusion

It is well known that shared objects, depending on their type, vary widely in their ability to support wait-free implementations. Recent research focussed on analyzing the power of individual objects. In this paper, we ask whether, from our understanding of the power of the individual objects, we can infer the combined power of a set of objects. For instance, is it impossible to implement a universal object from some combination of non-universal objects? The answer is not clear. It is conceivable that clever protocols for such implementations exist. Besides being of theoretical interest, these issues have implications to multi-processor architectures. To make a systematic study of these issues possible, we define the property of robustness for wait-free hierarchies. Contrary to popular belief, we show that Herlihy's wait-free hierarchy is not robust. We also show that some natural variants of Herlihy's hierarchy are not robust. This raises the obvious question of whether there is a non-trivial robust wait-free hierarchy at all. We do not know the answer. However, we

observe that if such a hierarchy exists, it is either h_m^r or some coarsening of h_m^r. Thus, further research on the structure of h_m^r is essential to resolving this open question. As explained in the paper, the answer to this question, regardless of whether it is affirmative or negative, has useful implications. We close with the conjecture that h_m^r is not robust. Our conjecture is motivated by the results of this paper which show how certain clever combinations of objects result in an increased ability to do consensus. Furthermore, we see no fundamental reason for h_m^r to be robust.

Acknowledgement

I had innumerable discussions with my advisor Sam Toueg on this subject. They were very helpful in crystalizing my ideas, and in discovering some of these results. The "swap object" that Jon Kleinberg and Sendhil Mullainathan showed me helped me discover the type `weak-sticky`. I am grateful to Sam, Jon, and Sendhil for sharing their insights with me. I thank Tushar Chandra and Cynthia Dwork for reading parts of this paper and providing helpful comments. Aparna helped me with typing. Little Sucharita never complained the travel between home and school, no matter what time of the night and how cold.

References

[AGTV92] Y. Afek, E. Gafni, J. Tromp, and P. Vitanyi. Wait-free test&set. In *Proceedings of the 6th Workshop on Distributed Algorithms, Haifa, Israel*, November 1992. (Appeared in Lecture Notes in Computer Science, Springer-Verlag, No: 647).

[AR92] Y. Aumann and M.O. Rabin. Clock construction in fully asynchronous parallel systems and pram simulation. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, October 1992.

[CHP71] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

[CIL87] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, August 1987.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[Her91a] M.P. Herlihy. Impossibility results for asynchronous pram. In *Proceedings of the 3rd ACM Symposium on Parallel Architectures and Algorithms*, July 1991.

[Her91b] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[Jay93] P. Jayanti. On the robustness of herlihy's hierarchy. Technical Report TR 93-1332, Cornell University, Dept. of Computer Science, Cornell University, Ithaca, NY 14853, March 1993.

[JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms, Haifa, Israel*, November 1992. (To appear in Lecture Notes in Computer Science, Springer-Verlag).

[KM] J. Kleinberg and S. Mullainathan. Resource bounds and combinations of consensus objects. In *this proceedings.*

[LAA87] M.C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[Plo89] S. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 159–175, August 1989.