On the Use of Registers in Achieving Wait-Free Consensus

Rida A. Bazzi*†
Gil Neiger*
Georgia Institute of Technology

Gary L. Peterson‡
Spelman College

Abstract

The computational power of concurrent data types has been the focus of much recent research. Herlihy showed that such power may be measured by examining the type's ability to implement wait-free consensus. Jayanti argued that this "ability" could be measured in different ways, depending, for example, on whether or not read/write registers could be used in an implementation. He demonstrated the significance of this distinction by exhibiting a nondeterministic type whose ability to implement consensus was increased with the availability of registers. We show that registers cannot increase the computational power (to implement consensus) of any deterministic type or of any type that can implement 2-process consensus. These results significantly impact upon the study of the wait-free hierarchies of concurrent data types. In particular, the combination of these results with other recent works shows that Jayanti's h_m hierarchy is *robust* for deterministic types.

*This author was supported in part by the National Science Foundation under grants CCR-9106627 and CCR-9301454. Author's address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280.

†This author was supported in part by a scholarship from the Hariri Foundation.

‡This author was supported in part by the W. F. Kellogg Foundation under a grant to the Center for Scientific Applications of Mathematics at Spelman College. Author's address: Computer and Information Science Program, Spelman College, 350 Spelman Lane SW, Post Office Box 333, Atlanta Georgia 30314-0339.

PODC 94 - 8/94 Los Angeles CA USA

1 Introduction

Achieving consensus in the presence of processor failures is of fundamental importance in distributed computing. A large body of research has studied algorithms for achieving consensus in three domains: (1) synchronous message-passing systems, (2) asynchronous message-passing systems, and (3) asynchronous read/write memory systems. While the first domain has produced a large number of deterministic algorithms, it has been shown that such algorithms do not exist in the other two [4–7,14]. Because of these results, researchers also consider algorithms for consensus in asynchronous systems with primitives more powerful than simple reads and writes [1,2,7,9–12,14,17,19].

Another reason for taking this approach stems from the study of wait-free implementations of concurrent data types. Here, researchers ask questions such as the following: "is there a wait-free implementation of type T_1 using objects of type T_2?" A concurrent implementation of a data type is *wait-free* if any process can complete any operation on the type in a finite number of its own steps, regardless of the behavior of other processes. Wait-free implementations are desirable in asynchronous systems because they prevent slow processes from slowing down faster ones; in addition, they tolerate any number of stopping failures. Herlihy [7] showed a direct connection between a type's ability to implement wait-free consensus and its ability to provide wait-free implementations of other types. In particular, he showed that consensus is *universal*: for any $n > 0$, if type T can implement wait-free consensus in systems with n processes, then T can provide a wait-free implementation of any type in such systems. In light of this result, Herlihy evaluated the power of a data type by assigning it a *consensus number*; this is the maximum number of processes for which the type can implement wait-free consensus. Herlihy thus cast the universe of concurrent data types into a hierarchy, each level of which contains types with a particular consensus number.

Jayanti [9] refined this study by asking the following

question: what does it mean to say that a type can implement wait-free consensus? He argued that an answer required addressing the following questions:

1. Can more than one object of the type be used in the implementation?
2. Can read/write registers also be used in the implementation?

Because these questions can be answered together in four different ways, Jayanti identified four possible hierarchies of types, one of which corresponds to Herlihy's assignment of consensus numbers (answering "no" to question 1 and "yes" to 2). He called these h_1, h_1^r, h_m, and h_m^r. A subscript "1" indicates that only one object of a type can be used, while a subscript "m" indicates that many can be used. A superscript "r" indicates that registers may be used, while its absence indicates that they may not. Using this notation, Herlihy's hierarchy is h_1^r.

Given these four hierarchies, Jayanti naturally asked if they were distinct and, if so, which best measured the computational power of different data types. He argued that it is desirable for a hierarchy to be *robust*. Informally, a hierarchy is robust if no collection of types at low levels can implement a type at a higher level. Jayanti showed that none of h_1, h_1^r, and h_m could be robust if it were not equal to h_m^r. He then showed that both h_1^r and h_m were different from h_m^r, proving that only h_m^r might be robust (h_1 cannot equal h_m^r if either h_m or h_1^r does not). Jayanti left the robustness of h_m^r as an open question.

Recall that Jayanti's hierarchy h_m was defined by answering "yes" to question 1 above and "no" to question 2; it differs from h_m^r on whether or not registers may be used in implementations of consensus. Jayanti proved $h_m \neq h_m^r$ (and h_m to not be robust) by exhibiting a type that was at different levels in the two hierarchies. This type was specified *nondeterministically*; that is, there is (at least) one sequence of operations on the type for which more than one behavior is possible. This raises an obvious question: can the same result be shown with a deterministic type? Since most commonly used concurrent data types are deterministic, a positive answer to this question would imply that the non-robustness of h_m would hold even for the class of deterministic types.

We answer this question in the negative. That is, we show that the two hierarchies give equal values for any deterministic type. Thus, the nondeterminism used by Jayanti is necessary. It is thus possible that h_m is robust for the class of deterministic types. The above result is first proven for types that are *oblivious*; objects of such types are not aware of the identities of processes accessing them. Most research in this area has concentrated on such types. Our results also hold for types that are not oblivious; such types are aware of (and may use) the identities of processes accessing them.

We also demonstrate other results relevant to the use of registers in implementing wait-free consensus. For all types (even nondeterministic ones), the two hierarchies can differ only at the first level: if either assigns a type a value greater than 1, then the other assigns it the same value.

These results confirm that, in most cases, registers do not play a special role in achieving wait-free consensus. In another paper [17], we show (as a corollary) that Jayanti's hierarchy h_m^r is robust for deterministic types. Combined with the results of this paper, we conclude that h_m is robust for these types.

Our results are proven through the introduction of a new concurrent data type called the *one-use bit*. An object of this type is a bit, initially 0, that can be read at most once and set to 1 at most once. Our main results stem from the following:

1. A finite number of one-use bits can implement a read/write register in a wait-free implementation of consensus (this is shown in Section 4).
2. Almost any type can be used to implement a one-use bit (this is shown in Section 5).

These results show that almost any type can be used to implement one-use bits. Thus, the availability of registers does not increase the ability of such a type to do consensus (if one is allowed multiple objects of the type). The existence of types that cannot implement one-use bits do not invalidate the results mentioned above because, as will be seen, they are too weak to implement 2-process consensus with or without registers.

2 Background

This section presents the definitions and background material necessary to present and interpret the results of this paper.

2.1 Types

A *type* is a 5-tuple $T = \langle n, Q, I, R, \delta \rangle$. The components are (1) n, the number of "ports" the type has (this limits the number of processes that may access the type), (2) Q, a (possibly infinite) set of *states*; (3) I, a set of access *invocations*; (4) R, a set of access *responses*; and (5) δ, a transition function. T may be either *deterministic*, in which case $\delta : Q \times N_n \times I \mapsto Q \times R$, or *nondeterministic*, in which case $\delta : Q \times N_n \times I \mapsto 2^{Q \times R}$.[1] This specification of a type indicates how processes may access an object of type T (using invocations in I), how the object communicates to processes (using responses in R), and what are the legal sequential histories of the

[1] $N_n = \{1, 2, \ldots, n\}$.

type (specified by δ). If an object of type T is in state q when invocation $i \in I$ appears on port $j \in N_n$, then the object changes to state q' and returns response r over port j if and only if $\langle q', r\rangle = \delta(q, j, i)$ (if T is deterministic) or $\langle q', r\rangle \in \delta(q, j, i)$ (if T is nondeterministic). A type is *oblivious* if, for all $q \in Q$, $j_1, j_2 \in N_n$, and $i \in I$, $\delta(q, j_1, i) = \delta(q, j_2, i)$. An oblivious type does not distinguish identical accesses by different processes. For oblivious types, we often abuse notation and omit the second (port number) input to the transition function.

A *sequential history* of T from a state q_0 is a sequence of alternating states and port-invocation-response triples (starting with q_0) such that certain conditions are met. In particular, consider the sequence

$$H = q_0; \langle j_1, i_1, r_1\rangle; q_1; \langle j_2, i_2, r_2\rangle; q_2; \cdots$$

(for all k, $q_k \in Q$, $j_k \in N_n$, $i_k \in I$, and $r_k \in R$). It must be the case that, for all k, $\langle q_k, r_k\rangle = \delta(q_k, j_k, i_k)$ (if T is deterministic) and $\langle q_k, r_k\rangle \in \delta(q_k, j_k, i_k)$ (if T is nondeterministic). We say that state q' is *reachable* from q if q' appears in some serial history from q. If H ends after k port-invocation-response triples, then the *length* of H, denoted $|H|$, is k.

An instantiation or *object* of type T in a system must specify, for each port, which process (if any) accesses the object through that port. At most one process may use a port. If T is oblivious and there are at most n processes in the system, then the assignment of "port numbers" is irrelevant.

The ability of a type to solve consensus is central to this paper. We define consensus as a type and then consider the ability of different types to implement an object of the consensus type. The *n-process binary consensus type* $T_{c,n}$ is an oblivious type defined to be $\langle n, Q, I, R, \delta\rangle$, where $Q = \{\perp, 0, 1\}$, $I = \{0, 1\}$, $R = \{0, 1\}$, and

$$\begin{aligned} \delta(\perp, 0) &= \langle 0, 0\rangle \\ \delta(\perp, 1) &= \langle 1, 1\rangle \\ \delta(a, b) &= \langle a, a\rangle \quad \text{for any } a, b \in \{0, 1\} \end{aligned}$$

Usually, objects of the type are chosen to have state $\perp$ initially. If a process's initial value is 0 (or 1, respectively), it performs invocation 0 (or 1, respectively). Note that the first invocation on the object determines all future responses. This response is sometimes called the *consensus value* of the object.

2.2 Implementations

This section defines what it means for one type to be implemented by others. Informally, an implementation is a set of objects (appropriately initialized) and deterministic programs that operate on these objects. There is one program for each process and for each invocation for the type being implemented. More formally, let $T = \langle n, Q, I, R, \delta\rangle$ and let $S = \{O_1, O_2, \ldots, O_m\}$ be a set of objects such that O_j is of type $T_j = \langle n_j, Q_j, I_j, R_j, \delta_j\rangle$. An *implementation of T in state $q \in Q$ from S for n processes* is a tuple of initial states $\langle q_1, q_2, \ldots, q_m\rangle$ ($q_j \in Q_j$) and a deterministic program P_{jk} for each $i_j \in I$ and each $k \in N_n$. The implementation should specify, for each object O_j, the port number of each process in the system that accesses O_j; at most m_j processes can do so. Each program of the implementation specifies how the implementing objects are to be accessed and what response should be returned to the invocation associated with that program.

An implementation is correct if all resulting histories are *linearizable* with respect to the specification of T starting from state q [8].[2] It is *wait-free* if, in all fair histories, all invocations of the deterministic programs terminate in a finite number of steps.[3] (For further details of these definitions, consult Herlihy [7] or Jayanti [9].) We say that there is *an implementation of T from S for n processes* if such an implementation exists for all $q \in Q$.

If there is an implementation of $T_{c,n}$ from S, we say that *S implements n-process consensus.*

2.3 The Universality of Consensus and Wait-Free Hierarchies

Herlihy [7] demonstrated that the consensus types $T_{c,n}$ are *universal* in the following sense. There is a wait-free implementation of any type T using registers and objects of type $T_{c,n}$ for systems of n processes. Because of this, Herlihy proposed evaluating different types by assigning them consensus numbers. The *consensus number* of type T is the largest number n for which some number of registers and a single object of type T can implement $T_{c,n}$.

Jayanti [9] questioned two of Herlihy's assumptions in assigning consensus numbers: whether or not registers should be used in the implementations of $T_{c,n}$ and whether or not multiple objects of a type can be used. To explore the impact of different choices here, he defined four *wait-free hierarchies*:

- $h_1(T) \geq n$ if one object of type T can implement n-process consensus.
- $h_1^r(T) \geq n$ if some number of registers and one object of type T can implement n-process consensus.
- $h_m(T) \geq n$ if some number of objects of type T can implement n-process consensus.

[2] A recent paper considers an alternative to linearizability that is appropriate for certain aspects of the study of asynchronous computability [15].

[3] A history is *fair* if each process either halts explicitly or performs an infinite number of operations.

- $h_m^r(T) \geq n$ if some number of registers and objects of type T can implement n-process consensus.

Herlihy's assignment of consensus number corresponds to Jayanti's hierarchy h_1^r. It is clear from these definitions that, for all types T, $1 \leq h_1(T) \leq h_1^r(T) \leq h_m^r(T)$ and $1 \leq h_1(T) \leq h_m(T) \leq h_m^r(T)$. In addition, standard techniques can be used to show that, if $T = \langle n, Q, I, R, \delta \rangle$, then $h(T) \leq n$ (where h is any of the hierarchies given above).

Ideally, the assignment of a consensus (or hierarchy) number to a type should be a good measure of the type's computational power. The larger the number assigned, the more power the type has to implement other types. Indeed, Herlihy's result on the universality of consensus shows that, if $h(T) = n \geq h(T')$ (where h is any of the hierarchies given above) and T' has at most n ports, then there is an implementation of T' using some number of registers and objects of type T.

Given four different ways of assigning these values, it makes sense to consider which is best. Jayanti identified a desirable property of hierarchies that he called *robustness*. Hierarchy h is *robust* if, for every choice of n, T, and $S = \{T_1, T_2, \ldots, T_m\}$, the relations $h(T) \geq n$ and $h(T_j) < n$ (for $1 \leq j \leq m$) imply that there is no implementation of T using objects of types in S. Robustness implies that there is no way to combine "weak" types of implement a "strong" type.

Jayanti showed that none of h_1, h_1^r, and h_m could be robust if it were not equal to h_m^r. He then showed that both h_1^r and h_m were different from h_m^r, proving that only h_m^r might be robust (h_1 cannot equal h_m^r if either h_m or h_1^r does not). Jayanti left the robustness of h_m^r as an open question.

Jayanti's proof that $h_m \neq h_m^r$ was based on the existence of a type T with $h_m(T) = 1$ and $h_m^r(T) \geq 2$. This type is nondeterministic. The remainder of this paper considers restricted classes of types for which h_m is shown equal to h_m^r. For these classes, h_m is robust if and only if h_m^r is.

3 One-Use Bits

The main results of this paper stem from the implementation and use of a new concurrent date type called the *one-use bit*. Objects of this type are one-bit registers that can be read only once and written only once. Section 4 shows that objects of this type can be used to implement general multi-reader, multi-writer, multi-value registers, while Section 5 shows that it is easy to implement this type.

The one-use bit type $T_{1u} = \langle 2, Q_{1u}, I_{1u}, R_{1u}, \delta_{1u} \rangle$ is defined as follows:

$$
\begin{aligned}
Q_{1u} &= \{\text{UNSET}, \text{SET}, \text{DEAD}\} \\
I_{1u} &= \{read, write\} \\
R_{1u} &= \{0, 1, ok\}
\end{aligned}
$$

$$
\begin{aligned}
\delta_{1u}(\text{UNSET}, read) &= \{\langle \text{DEAD}, 0 \rangle\} \\
\delta_{1u}(\text{SET}, read) &= \{\langle \text{DEAD}, 1 \rangle\} \\
\delta_{1u}(\text{DEAD}, read) &= \{\langle \text{DEAD}, 0 \rangle, \langle \text{DEAD}, 1 \rangle\}
\end{aligned}
$$

$$
\begin{aligned}
\delta_{1u}(\text{UNSET}, write) &= \{\langle \text{SET}, ok \rangle\} \\
\delta_{1u}(\text{SET}, write) &= \{\langle \text{DEAD}, ok \rangle\} \\
\delta_{1u}(\text{DEAD}, write) &= \{\langle \text{DEAD}, ok \rangle\}.
\end{aligned}
$$

Only *read* operations return informational responses; any such operation sends the object to the state DEAD. The object can never leave this state and, because of the nondeterminism, no further information about the bit can be obtained at this point. After two *write* operations, the object also goes to the state DEAD.

Note that, although the specification of this type is nondeterministic, this nondeterminism will play no role in our use of the type (Section 4); a *read* will never be invoked when the object is in state DEAD. Note also that, as specified, this type is oblivious. In all our uses of the type, only one process performs *write* and only one performs *read*. Thus, the object could have been specified as a 2-port, non-oblivious type without loss of applicability.

4 Using One-Use Bits

Although one-use bits are apparently weaker than general multi-use multi-value registers, we can show that, within the context of wait-free implementations of consensus, they are equally powerful. This can be shown through three observations:

1. General read/write registers can be simulated using single-reader single-writer multi-use bits.
2. For any n, any wait-free implementation of n-process consensus, and any single-reader single-writer bit b, there are bounds r_b and w_b such that b is read and written no more than r_b and w_b times, respectively, in any execution of the implementation.
3. If there are bounds on the number of times that a multi-use bit b can be read and written, then b can be simulated by a finite number of single-use bits.

These facts are shown in Sections 4.1–4.3 below.

4.1 Simulating General Read/Write Registers

A large body of literature has considered the definition and implementation of a variety of different kinds of read/write registers and the relationships between these

kinds. The registers required by Herlihy [7] and Jayanti [9] are atomic, multi-reader, multi-writer, and multi-value. Researchers have also considered registers that are regular (weaker than atomic), single-reader, single-writer, and one-bit. The following paragraph gives a very incomplete account of the large volume of results that that have been produced, mentioning only those that are necessary for the results of this paper.

Lamport [13] showed that there is a wait-free implementation of multi-reader, single-writer, regular bits from single-reader, single-writer, regular bits. Burns and Peterson [3] showed that there is a wait-free implementation of multi-reader, single-writer, atomic bits from multi-reader, single-writer, regular bits. Peterson [16] showed that there is a wait-free implementation of multi-reader, single-writer, atomic, multi-value registers from multi-reader, single-writer, atomic bits. Peterson and Burns [18] showed that there is a wait-free implementation of multi-reader, multi-writer, atomic, multi-value registers from multi-reader, single-writer, atomic, multi-value registers. Since atomic registers are stronger than regular registers, it follows from all these results that there is a wait-free implementation of multi-reader, multi-writer, atomic, multi-value registers from single-reader, single-writer, atomic bits. (All the implementations mentioned above exist for any number of processes.)

4.2 Access Bounds in Wait-Free Consensus

Consider some deterministic type T such that $h^r_m(T) \geq n$. This means that there is a wait-free implementation of $T_{c,n}$ in a system with n processes that uses some number of registers and objects of type T. As noted in the previous section, we can assume that these registers are single-reader single-writer bits. We show that, for each bit b, there exist constants r_b and w_b such that in no execution of the implementation is the bit read more than r_b times or written more than w_b times.

We can consider the executions of the implementation (from state $\perp$) as a collection of trees. Each node of a tree corresponds to some configuration of the implementing objects (the bits and the objects of type T) and the "program counters" of the n processes in their implementing functions. The roots of the trees correspond to possible initial configurations: the initial states of the implementing objects and the vector of invocations that the n processes will use to first access the $T_{c,n}$ object (each may be 0 or 1); that is, each process is at the "entry point" of one of its two implementing functions. (Notice that two root configurations can differ only with respect to the "entry points" selected by the processes; the implementing objects have the same states in all such configurations, as the implementation must specify a unique initial state for each object.) A configuration C_1 is the parent of C_2 if C_2 results from C_1 through the execution of one low-level operation by one process *in its first invocation on* $T_{c,n}$. (If a configuration can be reached via multiple paths, it appears multiple times.) Any configuration in which some process accesses the $T_{c,n}$ object a second time does not appear in a tree. Thus, a configuration in which all n processes have completed their first invocations is a leaf node.

We consider only first invocations because any later invocations by a process must return the same response as first (see Section 2.1). Thus, the process can store the first response locally and need not access any of the implementing objects after its first invocation on the $T_{c,n}$ object.

Consider any one of these trees. We will show by contradiction that it is finite. Assume that it is not. This means that a form of König's Lemma applies:

Lemma 1 (König): *If an infinite rooted tree has a bound on the fan-out of its nodes, then there is an infinite path from the root.*

The fan-out of our trees is bounded by n. Any node has at most n children, one for each process. This is because, as noted above, the processes are deterministic, as are registers and the type T.

König's Lemma now implies that there is an infinite path from the root. This path corresponds to some execution of the implementation. This means that there is an execution in which some process never completes its first invocation on $T_{c,n}$. This contradicts the fact that the implementation is wait-free.

The tree described is thus finite; let d be its depth, the maximum length of a path from the root. Consider now all the trees defined above. There are 2^n such trees. This is because the initial states of the implementing objects are the same in all trees, and only the choice of the entry points of the n processes can vary. Let D be the maximum d over all the trees. This means that, in each execution in which each process accesses the $T_{c,n}$ object at most once, at most D steps are executed. By the argument given above, this means that at most D accesses are invoked on any implementing object in any execution of the implementation. This means that, by choosing $r_b = w_b = D$, we know that, in no execution of the implementation, does any process read bit b more than r_b times or write it more than w_b times.

4.3 Implementing Multi-Use Bits

This section shows how any single-reader, single-writer bit that is accessed a bounded number of times can be implemented with a finite number of one-use bits. Suppose that bit b is initialized to v, is read at most r_b times, and is written at most w_b times. Because b is written by only one process, we assume that it is only written when its value is being changed.

The implementation uses $r_b(w_b + 1)$ one-use bits. These form an $(w_b + 1) \times r_b$ array

$$bits[1 \ldots w_b + 1, 1 \ldots r_b],$$

all elements of which are initially 0. Each row corresponds to a write and each column to a read. (This means that the last row is not actually necessary. It is included here to simplify the presentation of the read routine.) The reader maintains two local integer variables i_r and j_r, while the writer maintains local i_w and j_w; these are all initially 1. A write is performed by flipping all the bits in the row corresponding to the write:

for $j_w := 1$ **to** r_b **do**
    $bits[i_w, j_w] := 1$
$i_w := i_w + 1$
return(ok)

A read is performed by looking for a row that contains an unflipped bit:

while $bits[i_r, j_r] = 1$ **do**
    $i_r := i_r + 1$
$j_r := j_r + 1$
return$((v + (i_r - 1)) \bmod 2)$

Each read examines a different column to ensure that no one-use bit is read more than once. After an execution of a read, i_r contains the index of the first row that has not been completely flipped. This means that bit b has been written $i_r - 1$ times, so the returned value is $(v + (i_r - 1)) \bmod 2$ (recall that v is b's initial value).

A formal proof of the correctness of this implementation is deferred to the full paper.

5 Implementing One-Use Bits

This section illustrates two cases in which one-use bits can be implemented. These are non-trivial deterministic types and types above level 1 in hierarchy h_m. Section 5.1 first shows how non-trivial oblivious deterministic types can implement one-use bits. Section 5.2 extends this to the general case of any type. Section 5.3 shows how higher-level types in h_m can implement one-use bits.

5.1 Non-Trivial Oblivious Deterministic Types

Most, but not all, deterministic oblivious types can implement one-use bits. Some types, however, are so weak as to be *trivial* and are incapable of implementing any interesting type. Consider, for example, a type $T = \langle n, Q, I, R, \delta \rangle$ such that $|R| = 1$. Because the type must return the same response to every invocation, there is no way that it can supply any useful information. Formally, an oblivious type $T = \langle n, Q, I, R, \delta \rangle$ is *trivial* if, for every state $q \in Q$ and every invocation $i \in I$, there is a response $r_{qi} \in R$ and state q' such that $\delta(q, i) = \langle q', r_{qi} \rangle$ and, for every state p reachable from q, there is a state p' such that $\delta(p, i) = \langle p', r_{qi} \rangle$. A trivial type, once initialized, returns the same response to each occurrence of a given invocation; processes can gain no information by accessing an object of the type. A type that is not trivial is *non-trivial*. We now show that any non-trivial oblivious deterministic type can implement a one-use bit.

Let $T = \langle n, Q, I, R, \delta \rangle$ be a non-trivial oblivious deterministic type. This means that there are states $q, p, q', p' \in Q$, invocation $i \in I$, and responses $r_q, r_p \in R$ such that p is reachable from q, $\delta(q, i) = \langle q', r_q \rangle$, $\delta(p, i) = \langle p', r_p \rangle$, and $r_p \neq r_q$. It is not hard to see that p and q can be chosen such that p is reachable from q in one step, that is, so that there is some invocation i' and response r' such that $\delta(q, i') = \langle p, r' \rangle$.

We can now give an implementation of a one-use bit. We use one object O of type T, initialized to state q. A read of the bit is performed as follows:

invoke i on O
if response is r_q **then**
    /* O is still in state q */
    return(0)
else
    /* O was in state p */
    return(1)

A write is performed as follows:

invoke i' on O
return(ok)

Intuitively, state q corresponds to UNSET, p to SET, and any other state to DEAD. It is not hard to see that the above procedures correctly implement a one-use bit (note that, after the first read, any value can be properly returned by subsequent reads).

5.2 Non-Trivial Deterministic Types in General

The previous section showed that any non-trivial oblivious deterministic type can implement one-use bits. The proof given there depended on the obliviousness of the type being used. This section generalizes that result for types that are not necessarily oblivious.

A type is *trivial* if, for all start states and all ports, all sequences of invocations on that port always return the same sequence of results regardless of any invocations performed (and the order in which they are performed) on other ports. Thus, for any *non-trivial* type $T = \langle n, Q, I, R, \delta \rangle$, there is at least one start state q, a sequence of invocations on one port (without loss of generality port 1), and two sequential histories H_1 and H_2 from q that contain the same sequence of invocations on

port 1 such that one of the invocations in the sequence returns different values in the two schedules. Call H_1 and H_2 a *non-trivial pair*. Without loss of generality, we assume that the invocation returning different values is the last invocation on port 1. Let $\vec{\imath} = \langle i_1, i_2, \ldots, i_k \rangle$ be the sequence of invocations on port 1 in the two histories. Note that different sequences of operations may be invoked on ports other than port 1 in H_1 and H_2.

Consider any sequential history H from q in which the invocations on port 1 are $\vec{\imath}$. The history's *return value* is the result returned by i_k.

For the remainder of this section, we will assume that H_1, H_2, and q are such that $|H_1| + |H_2|$ is minimal among the non-trivial pairs. The following sequence of lemmas demonstrate certain properties of H_1 and H_2. These properties allow T to be used to implement one-use bits.

Lemma 2: *One of H_1 and H_2 has length k; that is, it consists only of invocations on port 1.*

Proof: Let H be the history from q consisting only of the invocations in $\vec{\imath}$ on port 1. Because the return values of H_1 and H_2 differ, the return value of H must differ from at least one of them, say H_2's. In this case, H and H_2 are also a non-trivial pair. Since $|H_1| + |H_2|$ is minimal, $|H| + |H_2| = k + |H_2| \geq |H_1| + |H_2|$, so $|H_1| \geq k$. Since H_1 must contain the k operations on port 1, $|H_1| = k$. □

From now on assume that H_1 contains only the k invocations on port 1 and that H_2 contains at least one invocation on another port (the latter because otherwise $H_2 = H_1$ and they have the same return value).

Lemma 3: *The last k invocations in H_2 are all on port 1.*

Proof: Let $m = |H_2|$ $(m > k)$ and assume that one of the last k invocations in H_2 in on some port other than port 1. The last invocation in H_2 must be i_k on port 1; otherwise, this invocation can eliminated, resulting in a shorter H_2', with H_1 and H_2' being a shorter non-trivial pair. Suppose that the last j invocations in H_2 are on port 1 $(k > j > 1)$ and that the $(m-j)$th is on some other port, say 2. Define a sequence of histories, $H_2^0, H_2^1, \ldots, H_2^j$ as follows: $H_2^0 = H_2$, and H_2^{i+1} results from inverting the order of $(m-(j-i))$th and $(m-(j-i)+1)$st operations in H_2^j. That is, each history successively moves the operation on port 2 (which is $(m-j)$th in H_2^0) one step later.

Let $\ell \geq 0$ be least such that H^ℓ and $H^{\ell+1}$ have different return values (if no such ℓ exists, then H_2^j has an invocation on port 2 as its last and forms a non-trivial pair with H_1, contradicting the above observation that both histories must end with invocations on port 1). Let q' be the state of the object after the first $m-j-1$ invocations of H_2 (note that $m-j-1 > k-j-1 = (k-1)-j \geq 0$). Let H_1' and H_2' be the last $j+1$ operations of H^ℓ and $H^{\ell+1}$. Then H_1' and H_2' have different return values from q' and thus form a non-trivial pair. Note that $|H_1'| + |H_2'| = 2(j+1) \leq 2k = k+k < |H_1| + |H_2|$. This contradicts the minimality of H_1 and H_2. We conclude that the last k invocations in H_2 are on port 1. □

Lemma 4: *The length of H_2 is $k+1$; that is, H_2 consists of one invocation (say i_w) on some port (say 2) followed by k invocations on port 1.*

Proof: Assume that $|H_2| = m > k+1$. Let H be a history from q consisting of the first $m-(k+1) > 1$ invocations of H_2 (none on port 1) followed by the last k invocations (all on port 1); $|H| = m-1$. If the return value of H is the same as that of H_2, then H_1 and H are a shorter non-trivial pair, giving a contradiction. Thus, the return value of H is different from that of H_2. Let q' be the state of the object after the first $m-(k+1)$ invocations of H_2. Let H_1' and H_2' be histories from q' containing the last k and last $k+1$ invocations in H_1 and H_2, respectively. The return value of H_1' is the same as that of H_1 and the return value of H_2' is the same as that of H_2. These differ, so H_1', and H_2' are a non-trivial pair. But $|H_1'| + |H_2'| = k+(k+1) < k+m = |H_1| + |H_2|$, again contradicting minimality. Thus, $|H_2| = k+1$. □

Lemma 4 establishes that an object O of any non-trivial deterministic type can be used by two processes to simulate a one-use bit. (Recall that, in Section 4, only two processes ever access such a bit in any of our uses of the type, one writing and one reading.) The reading process is connected to port 1 and performs a read as follows:

```
invoke ī on O
if return value is that of H1 then
    /* writer has not written */
    return(0)
else
    /* writer has written */
    return(1)
```

The writing process is connected to port 2 and simply performs the one invocation i_w from H_2 on that port:

```
invoke i_w on O
return(ok)
```

Note that the reader may get a return value that is neither H_1's nor H_2's. However, this still indicates that the writer has written, so 1 can be returned.

5.3 High-Level Types in h_m

Let T be any type such that $h_m(T) \geq 2$. This means that there is an implementation of 2-process consensus

using only objects of type T (without registers). We now show that, even if T is nondeterministic, T can implement one-use bits.

Let O be an object of type $T_{c,2}$, initialized to state $\perp$, as implemented by objects of type T. A read of a one-use bit is performed as follows:

invoke 0 on O
let r be response
return(r)

A write is performed as follows:

invoke 1 on O
return(ok)

Basically, the reader proposes 0, meaning "read precedes write," while the writer proposes 1, meaning "write precedes read." If the consensus value is 0, then the write could not have completely preceded the read, so the read can be linearized before the write and return 0. If the consensus value is 1, then the read could not have completely preceded the write, so the read can be linearized after the write and return 1. (Note that this implementation returns the same response to all reads by the reader; this is permitted by the nondeterministic specification of one-use bits.)

6 Applications to Wait-Free Hierarchies

The above results have two important applications to wait-free hierarchies, given below in Theorem 5.

Theorem 5: *Suppose that one of the following holds of type T:*

- *T is deterministic;*
- *$h_m(T) \geq 2$.*

Then $h_m(T) = h_m^r(T)$.

Proof: Let T be a type with one of the above properties. Recall that $1 \leq h_m(T) \leq h_m^r(T)$ for all types T. It thus suffices to show that $h_m^r(T) \leq h_m(T)$. The proof is divided into three cases:

- T is deterministic and trivial. This means that, no matter how an object of type T is initialized, any sequence of invocations by a process always returns the same sequence of responses. The object can thus be trivially implemented locally. This means that, if $h_m^r(T) \geq n$, then registers alone can implement n-process consensus. Since registers cannot implement 2-process consensus [4,7,14], this implies that $h_m^r(T) = 1$. Since $h_m(T) \geq 1$, $h_m^r(T) \leq h_m(T)$ as desired.
- T is deterministic and non-trivial. If $h_m^r(T) \geq n$, then objects of type T and registers can implement n-process consensus. As noted in Section 4.1, the registers can be single-reader, single-writer bits. Section 4.2 showed that there is bound on the number of times each bit may be used and Section 4.3 showed that, if this is the case, each such bit may be simulated by a finite number of one-use bits. Section 5.2 showed that objects of any non-trivial deterministic type can implement one-use bits. Thus, objects of type T can implement n-process consensus without using registers. This implies that $h_m(T) \geq n$ and, therefore, $h_m^r(T) \leq h_m(T)$ as desired.
- $h_m(T) \geq 2$. As noted above, if $h_m^r(T) \geq n$, then objects of type T and one-use bits can implement n-process consensus. Section 5.3 showed that objects of type T can implement one-use bits. Thus, objects of type T can implement n-process consensus without using registers. This implies that $h_m(T) \geq n$ and, therefore, $h_m^r(T) \leq h_m(T)$ as desired.

In all cases, $h_m^r(T) \leq h_m(T)$. This implies that $h_m(T) = h_m^r(T)$. □

Theorem 5 shows that Jayanti's choice of a type T to distinguish h_m and h_m^r was not accidental: it had to a nondeterministic type with $h_m(T) = 1$ and $h_m^r(T) \geq 2$.

7 Conclusions

The results of this paper show that, for two large classes of concurrent data types, Jayanti's wait-free hierarchies h_m and h_m^r are equal. One of these is the class of deterministic types, which is of considerable interest.

These results are of more than theoretical curiosities. They show that, in most cases of interest, registers are not "special" when it comes to implementing wait-free consensus. If convenient, they can be used to simplify the reasoning process: various arguments made with the assumptions that registers are available (e.g., about the hierarchy h_m^r) apply when they are not (e.g., to the hierarchy h_m); the converse is also true.

In particular, these facts pertain to Jayanti's robustness property. His proof that h_m is not robust does not apply, for example, to deterministic types. In fact, we have shown in another paper that h_m^r is robust for deterministic types [17]. The results of this paper show that h_m is also robust for these types.

Acknowledgments

We are grateful to Scott McCrickard for discussing this work with us.

References

[1] Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared memory. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, pages 47–58. ACM Press, August 1992.

[2] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 159–170. ACM Press, August 1993.

[3] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231. ACM Press, August 1987.

[4] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97. ACM Press, August 1987.

[5] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[6] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

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

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

[9] Prasad Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 145–158. ACM Press, August 1993.

[10] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. In *Proceedings of the Thirty-Third Symposium on Foundations of Computer Science*, pages 157–166. IEEE Computer Society Press, October 1992.

[11] Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality, and decidability of consensus. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth International Workshop on Distributed Algorithms*, number 647 in Lecture Notes on Computer Science, pages 69–84. Springer-Verlag, November 1992.

[12] Jon Kleinberg and Sendhil Mullainathan. Resource bounds and combinations of consensus objects. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 133–144. ACM Press, August 1993.

[13] Leslie Lamport. On interprocess communication; part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[14] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processors. In Franco P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, 1987.

[15] Gil Neiger. Set-linearizability and obliviousness: Foundations of the study of asynchronous computability. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1994. This volume.

[16] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[17] Gary L. Peterson, Rida A. Bazzi, and Gil Neiger. A gap theorem for consensus types. In *Proceedings of the Thirteenth ACM Symposium on Principles of Distributed Computing*. ACM Press, August 1994. This volume.

[18] Gary L. Peterson and James E. Burns. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science*, pages 383–392. IEEE Computer Society Press, October 1987.

[19] Serge Plotkin. Sticky bits and the universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–175. ACM Press, August 1989.