August 1989 Report No. STAN-CS-89-1280

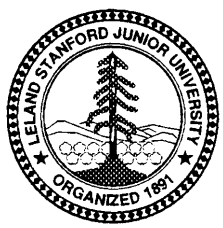
Sticky Bits and Universality of Consensus

by

Serge A. Plotkin

Department of Computer Science

Stanford University
Stanford, California 94305



Sticky Bits and Universality of Consensus

*Serge A. Plotkin**

Department of Computer Science
Stanford University
Stanford, CA 94305

August 1989

Abstract

In this paper we consider implementation of atomic wait-free objects in the context of a shared-memory multiprocessor. We introduce a new primitive object, the "Sticky-Bit", and show its universality by proving that any safe implementation of a sequential object can be transformed into a wait-free atomic one using only Sticky Bits and safe registers.

The Sticky Bit may be viewed as a memory-oriented version of consensus. In particular, the results of this paper imply "universality of consensus" in the sense that given an algorithm to achieve n-processor consensus, we can transform any safe implementation of a sequential object into a wait-free atomic one using polynomial number of additional safe bits.

The presented results also imply that the Read-Modify-Write (RMW) hierarchy "collapses". More precisely, we show that although an object that supports a l-bit atomic wait-free RMW is strictly more powerful than safe register and an object that supports 3-valued atomic wait-free RMW is strictly more powerful than l-bit RMW, the 3-value RMW is universal in the sense that any RMW can be atomically implemented from a 3-value atomic RMW in a wait-free fashion.

*Supported by ONR Contract N00014-88-K-0166 and a grant of Stanford's Center for Integrated Systems. Part of the work was done while the author was at M.I.T., Laboratory for Computer Science, Cambridge, MA 02139, supported by DARPA Contract N00014-87-K-825 and by ONR Contract N00014-86-K-0593.

1 Introduction

Consider an implementation of a concurrent object for a shared memory multiprocessor, where by "object" we mean a data structure together with a set of operations defined on this data structure. Analogously with Lamport's definition of a safe register, we will call an implementation safe if it is based on the assumption that no two accesses are concurrent. One way to transform a safe implementation so that it will work in a concurrent environment is to use mutual exclusion to "lock" the object before each access and to "unlock" it after the access is completed. The main advantage of this approach is simplicity, while the main disadvantage is that it causes one processor to wait for another, essentially reducing the speed of the system to the speed of the slowest component, which can be zero if this component has failed.

Informally, we say that an object is ***atomic*** if the accessing processors see as if no two accesses overlap and the order of the accesses is consistent with the partial order induced by the actual order of accesses (the actual order is partial since concurrent accesses are incomparable). This notion was formalized by Herlihy and Wing [7], who called it *linearizability*. We use the term ***atomicity*** in order to stress that it is a generalization of the notion of an ***atomic register*** [9]. Using this definition of atomicity, one can see that if we take a safe implementation and use "locks" to insure that the data is being accessed by at most one processor at a time, we get an atomic implementation.

Informally, we say that an implementation is ***wait-free*** if it guarantees that any processor that wants to access the object will complete the access in a bounded number of steps, independent of the speeds of the other processors concurrently accessing the object. In this paper we consider techniques that can be used to transform a ***safe*** implementation into an ***atomic wait-free*** one.

In order to answer whether it is possible to transform a safe implementation into a wait-free atomic one, it is important to specify precisely which memory model we are working with, i.e. which primitive atomic objects are supported "in hardware". The simplest case is when we require the atomic wait-free implementation to use safe registers only. Unfortunately, this model is "weak". Dolev, Dwork, and Stockmeyer [5], and Chor, Israeli, and Li [4], have proved that safe registers are not sufficient in order to reach even 2-processor wait-free consensus. From this Herlihy [7] has concluded that safe registers are not sufficient to implement wait-free versions of simple data objects like queues, stacks, etc.

Herlihy [7] defined the notion of ***universal object.*** Intuitively, an object is universal if we can convert any safe implementation into a wait-free atomic one that uses only safe registers and these objects. He showed that, assuming ***unbounded memory,*** an object that supports consensus is universal. Also, his results imply universality of an atomic swap operation, i.e. an operation that exchanges the context of a given register with a context of another one.

An important question is whether there exists a simple consensus-like object whose universality does not depend on availability of unbounded memory. We would like this object to be simple enough to be easily implemented in hardware. Note, for example, that implementation of a ***swap*** operation requires the hardware to impose a ***complete order*** on the processors concurrently accessing the object, which might be non trivial.

In this paper we introduce a new primitive object, the ***Sticky*** Bit (SB), and prove its universality by showing how to use $O(n^2 \log n)$ Sticky Bits to transform any serial implementation

into a wait-free atomic one, where n is the number of participating processors. Informally, an atomic Sticky Bit (ASB) is a register which can hold 0,1, or "undefined". If several processors are concurrently trying to write into the same ASB, only one of them succeeds. A processor returns "fail" if the value it was trying to write disagrees with the already written value, and "success" otherwise. ASB also supports a safe operation that resets the value to "undefined", where "safe" means that concurrent execution of this operation by 2 processors leads to unpredictable results.

It can be seen that ASB is a special case of a 3-valued register that supports a restricted variant of an atomic Read-Modify-Write (RMW). Hence, universality of ASB implies that the RMW hierarchy "collapses". More precisely, our results imply that although there is no wait-free implementation of 2-value atomic Read-Modify-Write (RMW) from safe bits [5, 7] and there is no wait-free implementation of S-value atomic RMW from 2-value atomic RMW [7, 10], the 3-value RMW is universal in the sense that any RMW can be atomically implemented from a 3-value atomic RMW in a wait-free fashion using bounded memory.

The Sticky Bit can be viewed as a version of consensus, which has proven to be a valuable tool in understanding the limitations of asynchronous distributed systems [5, 6]. On the other hand, the definition of the Sticky Bit is memory-oriented, which makes it a convenient alternative to consensus in the context of shared-memory systems. A Sticky Bit object can be easily constructed from two safe bits and a single initializable object that implements a wait-free single- bit consensus, where "initializable" means that the object can be initialized by one of the participating processors so that it can be used again to reach consensus, as long as the initialization does not overlap any other operation.

The construction presented in this paper indicates that reaching consensus is the fundamental problem in wait-free synchronization. In particular, randomized consensus algorithms of Chor, Israeli, and Li [4], Abrahamson [1], Aspnes and Herlihy [2], and Attiya, Dolev, and Shavit [3], together with our construction imply that polynomial number of safe bits is sufficient to convert a safe implementation into a (randomized) wait-free one.

The paper is organized as follows. Section 2 presents the model and Section 3 formalizes the notions of wait-freeness and atomicity. Section 4 presents definition of the Sticky-Bit object and illustrates its use by presenting a wait-free leader election algorithm. In Sections 5 and 6 we describe how to transform any safe implementation into a wait-free one using only Sticky Bits and safe registers, which proves that Sticky-Bit data object is universal. Conclusions and open problems are presented in Section 7.

2 Model

In this section we sketch the model and the main definitions used in the subsequent sections. Our model is similar to the one proposed by Herlihy [7]. The main difference is that we have eliminated the explicit scheduler from the model and use I/O automata to describe implementations of objects. The use of I/O automata provides a convenient way of formalizing the notion of "one object simulating another".

Informally, we regard a shared-memory multiprocessor executing a number of sequential

threads as a set of asynchronous processes communicating through shared data objects, where each process corresponds to an execution of a procedure. Each sequential thread is executing a single procedure at a time, where execution of the "call" instruction causes it to suspend the current procedure and start executing the one that was invoked. The suspended procedure is not continued until the invoked one executes the "return" instruction. This corresponds to a system that does not support instructions that create new processes, i.e. all threads exist from the beginning.

We use I/O Automata of Lynch and Tuttle [12] with the addition of ports, as described in [11]. We view all communication as if it is done through ***I/O channels,*** where each channel has two endpoints called ports. For each channel, one port is called the ***master*** and another one is called the slave, where slave ports correspond to entry-points of procedures and master ports correspond to "call" instructions. Messages sent from master ports are called ***commands*** and messages sent from slave ports are called responses.

Processors and data objects are modeled as non deterministic automata with possibly a countably infinite number of configurations and possibly a countably infinite fan-out from every configuration. An Input/Output Automaton is a tuple M = (E, Q, Q^0), where $\mathcal{E} = \mathcal{E}^{out} \cup \mathcal{E}^{in} \cup \mathcal{E}^{int}$ is the set of actions ($\mathcal{E}^{out}, \mathcal{E}^{in}, \mathcal{E}^{int}$ are output, input, and internal actions, respectively), Q is the set of states, and $Q^0 \subseteq$ Q is a distinguished set of start states. Each action corresponds to a transition of the automaton from one state to another; we say that the actions which correspond to transitions out of a given state are ***enabled*** in this state. An execution of an automaton is represented by a sequence of actions, which we call a ***schedule.***

A ***port automaton*** is a tuple (Val, ***II, CH, M),*** where

- Val is the set of values that can be sent as messages.
- II is the set of ports. Each port has a ***type,*** which is either ***master*** or ***slave.***
- CH is the set of channels. Each channel is a pair of ports, one of type master and the other one of type slave. A port can belong to at most one channel; a port which does not belong to a channel ***is*** called ***external,*** and the rest ***are*** called ***internal.***
- M is an input/output automaton with the property that every action is a tuple (m, π); m $\in Val$, $\pi \in$ II. We consider only schedules where any action (m, π_1) is followed by (m, π_2) ***if ch*** = $(\pi_1, \pi_2) \in$ CH. To simplify notation, we write (m, ***ch) in*** this case. All external actions are associated with external ports; all internal actions are associated with internal ports.

An output action associated with a master port or an input action associated with a slave port is called a ***command;*** an input action associated with a master port or an output action associated with a slave port is called a ***response.*** A port automaton is ***well formed*** if any schedule H restricted to any port π starts from a command and consists of alternating commands and responses. We say that port π is ***input enabled*** in a state C if there exists an input action associated with this port, that is enabled in C. A schedule ***H*** is ***balanced*** if it brings the system to a state in which a port is input-enabled if and only if it is of type slave.

We say that an automaton has a ***procedure signature*** if it has at most one slave port; an automaton has ***an object signature*** if it has no master ports. In order to abbreviate, we refer to these automata as procedures and objects, respectively.

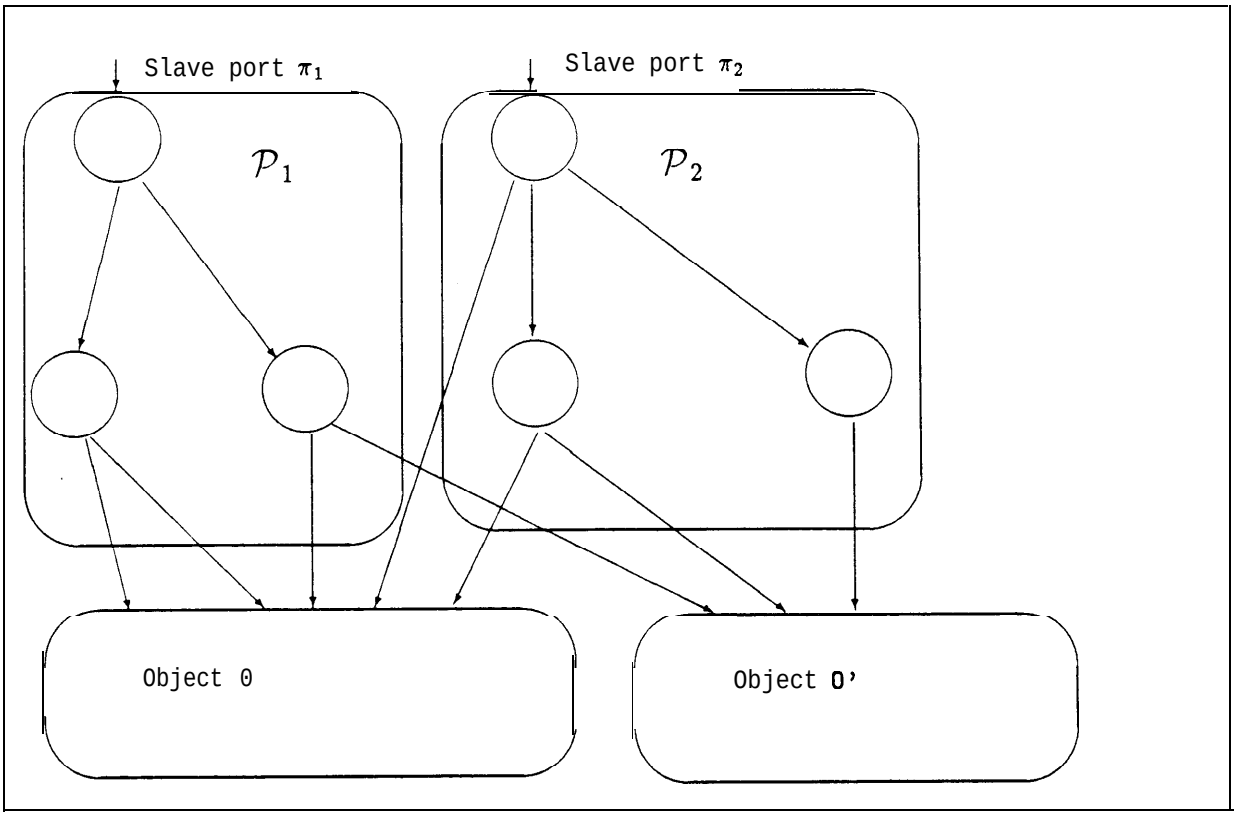


Figure **1:** Implementation of an object in terms of objects 0 and 0'. Arrows correspond to channels and are directed from master to slave ports; circles and ovals correspond to procedure and object automata, respectively.

A composition of several port automata with disjoint set of ports is done exactly like the composition of input/output automata. In order to model interactions between components we can link one component to another by defining new channels. A composition of several port automata with additional channels is called a ***system.***

An object is specified by describing its external ports and stating all of its legal external schedules. We say that an automaton ***A*** implements an object 0 if it has the same set of external ports and its external schedules are a subset of the schedules specifying 0. Moreover, we require that the set of sequential schedules of automaton ***A will*** be the same as the set of sequential scedules of 0.

It is important to formalize the notion of ***implementing*** one object ***in terms of another.*** An implementation of an object 0 in terms of objects $O_1, O_2, \ldots, O_k$ is a port automaton that implements 0 and is a system constructed by interconnecting several port automata such that each automaton in the system is either a procedure or it implements one of the objects $O_1, O_2, \ldots, O_k$.

An example of an implementation of one object in terms of other objects is given in Figure 1.

Arrows correspond to channels and are directed from master to slave ports; circles and ovals correspond to procedure and object automata, respectively. Consider the directed graph with nodes corresponding to the automata and the edges corresponding to channels. Observe that this graph is acyclic. Associate with every external port π_i the system that consists of the interconnection of all automata with procedure signatures that correspond to nodes reachable from the node that corresponds to the automaton that owns π_i, and denote the obtained automaton by $\mathcal{P}_i$. Note that $\mathcal{P}_i$ has a procedure signature. Intuitively, this corresponds to decomposing the system into "front-ends" and "representation objects", and ***we call this canonical decomposition.***

With each system ***A we*** associate a system A that hides the actions of ***A*** that correspond to internal channels. For each schedule ***H*** of ***A,*** the corresponding schedule of A is called the ***external schedule*** and is denoted by $H|\tilde{\mathcal{A}}$. The actions in $H|\tilde{\mathcal{A}}$ are divided into actions corresponding to the slave ports of A, and actions corresponding to the master ports of A, which we denote by $\boldsymbol{H}|\tilde{\mathcal{A}}^s$ and by $\boldsymbol{H}|\tilde{\mathcal{A}}^m$, respectively.

3 Wait-free Atomicity

In this section we use the formalism presented in the previous section to define the notion of wait-free atomicity. Often, it is natural to give a specification of a data object in terms of the behavior of this object when every command sent to the object is executed and acknowledged before a new command is issued. For example, in the case of a register, this corresponds to executions in which reads or writes do not overlap. More precisely, a schedule is ***sequential*** if every command action in this schedule is immediately followed by the corresponding ***response*** action on the same port.

An object is sequential if it is specified by sequential schedules only. Registers, queues, and stacks are examples of sequential objects (if we assume that a "dequeue" operation on an empty queue or "pop" of an empty stack are defined to return exceptions). On the other hand, consider a 2-port object that, given a value on one of the ports, responds with this value or with the value sent to it through the other port, whichever is larger. This object is not sequential because the response to the first command has to come after the second command, i.e. the waiting is inherent in the specification of the object.

Lamport [9] defined a register as safe if a "read not concurrent with any write obtains the most recently written value". This can be generalized to arbitrary sequential objects in a straightforward manner, i.e. an implementation of a sequential object is safe if the set of all schedules which are sequential with respect to this object is a subset of the sequential schedules specifying the object. In other words, an implementation is safe it it assumes that the accesses to the object never overlap in time.

For the case of registers, Lamport has defined the notion of ***atomicity.*** Intuitively, an object is atomic if it behaves as if each operation occurs somewhere between the command and the response. In order to generalize the notion of atomicity to arbitrary sequential objects, we first define a partial order on operations in a given schedule ***H,*** where an operation is a command action followed by a response action in the restriction of ***H to*** the external ports of the object. Let o = (e_c, e_r) and o' = (e'_c, e'_r) be two operations in ***H.*** Recall that e,, e'_c are commands and

e_r, e'_r are responses. Then o $\prec_H$ o' if both e_c and e_r appear before e'_c in ***H.***

Definition 3.1 (Herlihy-Wing [8])[1] An object is ***atomic*** if for every external schedule ***H*** of this object, there exist schedules ***H'*** and S, such that ***H'*** is a balanced extension of ***H,*** S is sequential schedule consisting of the same actions as ***H';*** and $\prec_{H'} \subseteq \prec_S$. The schedule S is called linearization of ***H*** and is denoted by ***L(H).***

It is easy to prove that under this definition of atornicity, if an object is atomic, then in order to prove the correctness of the system, it is sufficient to prove correctness under the assumption that accesses to the object do not overlap [7, 13].

Atomicity is only one of the properties required by Lamport of "atomic registers". Another, not less important property, is "wait-freeness". Intuitively, the idea is that the time it takes to access an object should depend only on the speed of the accessing processor. In particular, the object has to eventually return a response, even if all the rest of the processors "died" in the middle of an operation. Note that without this requirement atomic objects can be trivially constructed using busy- waiting.

Intuitively, one can regard an implementation of an object as if it consists of a set of representation objects and a set of "front-end" processes. Informally, an implementation of an object is wait-free if there exists N, such that each invocation of a "front-end" process returns a response after executing for at most N steps, independent of the execution of other "front-ends". An alternative definition of wait-freeness is to require that the response will be returned after a ***finite*** number of steps. Note that our definition is stronger.

Formally, wait-freeness is defined as follows. Consider a system with canonical decomposition $(P_1, P_2, \ldots, P_k, O_1, O_2, \ldots, O_l)$ where external slave port π_i corresponds to component P_i. Let $\mathcal{H}(C)$ denote the set of possible schedules that start at state C.

Definition 3.2 Let $\mathcal{H}_N^{p_i}(C) = \{H : H \in \mathcal{H}(C), ||H|(P_i, O_1, O_2, \ldots, O_l)|| > N\}$. The signature of an external slave port π_i is ***wait-free*** if there exists **N,** such that:

1. For any state C in which the port π_i is not input-enabled, there exists a schedule $H \in \mathcal{H}_N^{p_i}(C)$ such that it does not include any commands associated with external ports of ***(Pi,*** $O_1, O_2, \ldots, O_l$).
2. For any state C in which the port π_i is not input-enabled, and for any schedule $H \in \mathcal{H}_N^{p_i}(C)$, there is a prefix ***H'*** of ***H,*** s.t. π_i is input-enabled in state ***H'(C).***

Herlihy [7] introduced the notion of a ***universal object.*** A data object 0 is ***universal*** if given a ***safe*** implementation of a sequential object 0', we can construct a wait-free implementation of O' in terms of O-type objects.

4 Sticky Bit

In this section we introduce **a** new data object, the *Sticky Bit*, and illustrate the use of this object by presenting a deterministic wait-free leader-election algorithm.

[1] 'Our notion of atomicity is essentially the same as the notion of linearizability introduced by Herlihy and Wing [8]. Note, that since it refers to individual operations as opposed to sequences of operations, it is different from the standard notion of atomicity for distributed systems.

```
Procedure JAM(v_i)
    i, i' ← the ID of the processor;
    Jam-0(g_i);
    for j ← 1 to l do
        b - j-th bit of v_i';
        jam jth bit of v with b;
        if the jam failed
        then begin
          for k ← 1 to 1 do
              if g_k = 0 and ∀ 1 ≤ k' ≤ j, (k'th bit of v_k) = (k'th bit of v)
              then i' - k;
          end;
        end;
    end;
    return v;
end.
```

Figure 2: Code of $Jam(v_i)$, executed by processor i.

Definition 4.1 An atomic Sticky Bit (ASB) is a data object that holds 0,1, or $\perp$ and atomically supports the following operations:

- JAM(V) – If the value was $\perp$ or v, sets it to v and returns "success". Otherwise returns "fail".
- READ – Returns the current value of the object.

In addition, it supports a ***non-atomic*** FLUSH operation.

- FLUSH – Sets the value to $\perp$. This operation is non-atomic in the sense that any other operation that overlaps it produces unpredictable results.

A consensus protocol was defined in the seminal paper of Fisher, Lynch, and Paterson [6] to be a protocol where each processor has a l-bit input and produces a l-bit output which confirms to two conditions. First, all produced outputs are the same, and second, if the output is v then there is at least one participating processor whose input is v. The consensus protocol can be naturally represented as an n-port object. It is easy to see that it is possible to construct an atomic Sticky Bit from an initializable single-bit consensus object and two safe bits, where "initializable" means that after using the object to reach consensus it is possible to initialize it and use it again, as long as the initialization does not overlap with any other operation. In particular, this implies that the randomized consensus algorithms [1, 2, 3, 4] can be used to construct a randomized wait-free atomic Sticky Bit. Note also that ASB is a special case of the write-once memory, and can be easily implemented in hardware.

The usual problem that arises when designing wait-free algorithms is that even if it is sufficient that a single processor will execute some task, we can not assign a specific processor to this task because the adversary might make this processor fail-stop. ASB objects provide a

convenient way to address this problem. In particular, they allow several processors to execute the same task concurrently, "helping" each other, without interfering one with another.

A simple example that illustrates how to use this capability, is a wait-free atomic implementation of a "Sticky-Byte" object. This object is similar to the Sticky-Bit, but holds a number of bits (say l) instead of a single one. The command that corresponds to JAM(0) or JAM(1) of ASB is JAM(V), where v is an Z-bit value. Similarly to ASB, JAM(v) returns "success" if the object holds v after JAM(V) returns, and "fail" otherwise.

Observe, that a straightforward implementation that is based on representing a Sticky-Byte by l ASB objects where each processor simply tries to jam its bits one-by-one, leads to incorrect values. For example, consider the case where l = 2 and one processor tries to jam (1,0) and the other one tries to jam $(0,1)$. A possible scenario is that the first bit is jammed by the first processor and the second bit is jammed by the second processor, leading to the incorrect value of $(1,1)$. On the other hand, if a processor "returns" immediately after it comes to the conclusion that it must return "fail", then some of the bits of the Sticky Byte might remain undefined if the processor that is supposed to return "success" is stopped by the adversary.

The main idea is to require any processor that recognizes that he must return "fail" ***to help*** the processor that might still return "success". The code executed by processor p_i in order to simulate JAM(v_i) operation is shown in Figure 2.

The algorithm begins with p_i storing its input into v_i and marking that its v_i is valid by jamming g_i to 0. Then, it executes l iterations, where l is the number of bits in the input. At iteration j, p_i tries to jam the jth bit of the decision to be the jth bit of $v_{i'}$, where initially i' is the processor's ID. If it does not succeed, it means that there exists k and some processor $p_{k'}$, such that $p_{k'}$ has succeeded in jamming the jth bit of v_k into the jth bit of v. By induction on the number of iterations, we see that there exists at least one v_k such that g_k = 0 and the first j bits of v_k correspond to the first (already jammed) j bits of v. The processor p_i finds such k, and from now on tries to jam v_k into v, essentially "helping" processor p_k. Note that this algorithm has an interesting property that even if a processor p_i stops immediately after jamming 0 into g_i, its input may still be the decision jammed into v.

Observe, that if each processor tries to jam its own ID, the above algorithm implements a wait-free leader-election in O(log n) time. This implies that an atomic Sticky Byte that holds an arbitrary number of bits can be implemented from log n atomic Sticky Bits, where an access time of such implementation is 0 (log n).

5 Atomic Implementation of an Arbitrary Object.

In order to construct an atomic simulation of an arbitrary sequential object, we must be able to impose an order on accesses that overlap in time, such that this order will be consistent with their real order (see Definition 3.1), i.e. if an access was completed before another one was started, then the same relation between these accesses should exist in the imposed order. In other words, we must construct a sequential schedule which is consistent with the real schedule.

A natural approach, proposed by Herlihy [7], is to assume that the system supports an atomic operation that prepends an element to the beginning of a list. The idea is that a

processor that wants to access the object stores the command in the list, and then uses the commands that were stored beforehand in this list to compute the "current" state of the object and the appropriate response. The commands stored in the list correspond to a sequential schedule which is consistent with the real one, and therefore the implementation is atomic.

In order to be able to show that it is possible to implement such list from Sticky Bit objects, we review the construction of [7], adding several details. Our modifications mostly concern memory allocation, since we do not want to assume any "built-in" memory-management primitives.

Upon receiving a command ***cmd,*** processor p_i proceeds as follows:

1. Gets a free cell Cell and stores ***cmd*** in this cell.
2. Uses APPEND to prepend *Cell* to the list.
3. Reads the cells in the list one by one to construct the suffix S of the sequential schedule of the simulated object. The cells are read until it encounters a cell that holds a state instead of a command.
4. Computes the state of the object that results from applying S to the encountered state and stores it in *Cell*.
5. Frees cells of the list that belong to it and that have at least ***n*** cells that hold states (and not actions) ahead of them in the list.
6. Computes the response ***rsp*** of the object and returns it.

Observe, that because of Step ***4*** there are at most n cells in the list that hold actions and not states. Hence, in order to compute the "current" state of the object it is enough to scan at most ***n*** cells of the list, and therefore the implementation is wait-free. Step 5 is needed to bound memory. As a result of the fact that a processor never scans more than n cells of the list and the fact that it stops scanning after encountering a cell that holds a state, Step 5 does not remove from the list any cell that might be read by some processor.

Though it seems that the main problem lies in implementing the APPEND command, there are several additional issues which were omitted from the description of universal constructions of Herlihy [7]. First, note that care should be taken while implementing Step 5. In particular, a straightforward implementation, i.e. recognizing that there are at least ***n cells*** that hold states ahead in the list by scanning these cells, is incorrect. The problem lies in the fact that some of these cells might be already freed, initialized, and used again, causing us to follow "dangling" pointers. One way to solve this problem is to add ***n*** bits $\{b_1, b_2, \ldots, b_n\}$ to each cell, initially all 0. After a processor modifies its current cell to hold a state, it scans the following n cells in the list, writing 1 in the appropriate bit of each cell, according to the distance to this cell. Each time a processor is invoked and needs a new cell, it first checks all of its cells that are still in the list, and frees those that have all the bits b_i equal to 1. This way the list is traversed only in the forward direction. Moreover, if a processor accesses a cell during this traversal, then the appropriate b_i bit of this cell was 0 before the access, and therefore this cell could not have been initialized, which means that we never follow "dangling" pointers.

The second point concerns the space complexity. Herlihy claims in [7] that because there are at most ***n cells*** in the list that hold actions and not states, and because each such cell can

delay at most n other cells from being freed from the list, the space complexity is $\Theta(n^2)$ in the worst case. This claim is correct if we assume that there exists a single bounded-size pool from which new cells are allocated. The problem is that it is impossible to implement such pool from safe registers, because it allows wait-free 2-processor consensus. Therefore, it seems that the actual space complexity of Herlihy's construction (which assumes that the system supports a wait-free atomic operation that prepends an element to the beginning of a list[2]) is $\Theta(n^3)$ *in* the worst case, which is achieved by allocating disjoint pools to each one of the processors.

6 Universality of Sticky Bit

In the previous section we have described how to transform a safe implementation of a sequential object into an atomic wait-free one if we are given an "augmented list" object, which supports the following operations:

- GFC – Gets a free cell.
- INIT – Frees and initializes the given cell.
- APPEND – Prepends the given cell to the beginning of the list.

In this section we show universality of the atomic Sticky Bit object by presenting an implementation of these operations. Similarly to the implementation of the Sticky Byte, the heart of the presented algorithms is the idea of extending "help" to less fortunate processors.

In order to implement the operations atomically, the procedures described in this sections "busy- wait" until the operation is executed and only then return. Thus, the problem is to limit the amount of this waiting in order to achieve wait-freeness, and this is where the "help" paradigm comes into play. Each implementation of an operation is constructed from two procedures: the kernel procedure which actually implements the operation, and the control procedure, which repeatedly invokes the kernel. The kernel procedure is designed so that it either succeeds in executing the operation or returns "failure". Moreover, if[1] it returns "failure" then at least one other processor has succeeded in executing the same operation concurrently. The control procedure repeatedly invokes the kernel procedure until it returns with success, and then invokes the kernel procedure ***on behalf*** of each one of the processors that is currently trying to execute this operation. This ensures that there can be at most n consecutive "failures" of kernel procedure, which makes our algorithms wait-free. Sticky Bits are used in order to enable one processor to execute on behalf of the other one without interfering with him; essentially, the idea is that the processor that is being helped does not "know" this.

The information stored in each cell of the list is shown in Figure 3. In particular, the Prev field is constructed as an atomic Sticky Byte, and is used to decide which processor succeeds in appending his cell to the list; *ProcID* is also a Sticky Byte which is used to decide which processor currently "owns" the cell (this processor is responsible for initializing it).

[1] Herlihy's construction of a wait-free atomic object directly from multibit consensus uses unbounded memory.

Claimed:	Equals to 1 only if the cell is not free. (Sticky)
ProcID:	Holds the processor ID that "owns the cell". (Sticky)
NotHead:	Equals true when the cell is in the list but is not the head of the list. (Sticky)
Data:	The value of the cell. (Safe)
Next:	Points to the next cell in the list. (Sticky)
Prev:	Points to the previous cell in the list. (Sticky)
hit:	Equals 1 if the cell is being initialized. (Safe)
$_1, r_2, \ldots, r_n$:	The bit r_i equals to 1 if processor p_i does not want the cell to be initialized. (Safe)
CountInit:	The highest i such that for all $j \leq i$, the owner of the cell saw each one of r_j being equal to 0. (Safe)

Figure 3: Information stored in a single cell

6.1 Implementation of INIT

Initializing a cell involves, in particular, flushing the Prev field. We use the GRAB and RELEASE procedures, shown in Figure 4, to prevent the flushing while there exists a possibility that some processor might read this field, because by definition of the Sticky Bit, FLUSH can not be overlapped by any other operation. Before accessing a cell, we require that the processor will GRAB it first, by checking the *Init* bit, setting the appropriate r_i, and checking the *Init* bit again.

The initialization of a cell is done by the processor that "owns" it, i.e. the processor whose ID is stored in the *ProcID* field of the cell. This processor fist sets *Init* bit to state that the cell is under initialization, and then it checks the r_i bits one by one. The idea is that by deferring the initialization until the processor who owns the cell sees each one of the r_i bits being zero at least once, we guarantee that the access to the "sticky" data in the cell and the initialization of the cell are never executed concurrently.

Lemma **6.1** If a call to GRAB(*Cell*) by processor p_i returns "success", *Cell* will not be initialized until a subsequent call to RELEASE(*Cell*) by the same processor.

Proof: The fact that the call by processor p_i to GRAB(*Cell*) returned "success" means that p_i saw *Cell.Init=false* after setting *Cell.*r_i to 1. If the processor p_j tries to initialize the *Cell* after this has happened, it can not succeed since the first thing it will check is whether *Cell.*r_k =0 for all $1 \leq k \leq n$. The only possible problem might arise if p_j is in the middle of initialization of Cell. Clearly, it can happen only if the second access of *Cell.Init* by p_i and the access by p_j occurred concurrently. Therefore, in this case, p_j will start the access of *Cell.*r_i only after p_i has completed writing 1 into it, and INIT will fail. ∎

Lemma 6.2 A call to INIT fails only if there is some processor that is currently executing GRAB on this cell or it has already finished executing GRAB but has not executed RELEASE yet.

Proof: Omitted. ∎

```
procedure GRAB( Cell);
     if Cell.Init = true
       then return "fail";
       else begin
         Cell.r_i ← 1;
         if Cell.Init = true
           then begin
             Cell.r_i ← 0;
             return "fail";
           end;
           else return "empty";

       end;
end.

procedure RELEASE(CELL);
     CELL.r_i ← 0;
end.
```

Figure 4: The GRAB and RELEASE procedures executed by processor p_i

```
procedure INIT( Cell);
     if Cell.Init = 0 then Cell.Init ← 1;
     RELEASE(Cell);
     j ← Cell.CountInit;
     while j < n and r_j = 0 do begin
         j ← j + 1;
     end;
     Cell.CountInit ← j
     if j = n
       then begin
         initialize the cell;
         Cell.CountInit ← 0;
         Cell.Init ← 0;
         return "success";
       end;
       else return "fail";

end.
```

Figure 5: The INIT procedure.

```
procedure GFC;
     AnnounceGFC(i) ← 1;
     Cell ← GFC-INNER(~);
     Cell. Chimed ← true;
     AnnounceGFC(i) ← 0;
     for all j such that AnnounceGFC(i) = 1 do begin
         Tmp ← GFC-INNER(~);
         RELEASE( Tmp);
     end;
     return Cell;
end.

procedure GFC-INNER(ID);
     for all cells Cell in memory do begin
         if GRAB( Cell)= "empty" and Cell.ProcID = ID and Cell.Claimed = 0
           then begin
             return Cell;
           end;
     end;
     do forever
        for all cells Cell in memory do begin
            if GRAB(Cell) = "empty"
              then begin
                try to jam ID into Cell.ProcID;
                if "success" and Cell.Claimed = false
                   then return Cell;
                   else RELEASE( Cell);
              end;
        end;
     end;
end.
```

Figure 6: The ***GFC*** procedure executed by processor p_i

6.2 **Implementation of** GFC

The code implementing the **GFC** operation is presented in Figure 6. To get a free cell, the processor p_i first sets $AnnounceGFC(i)$ to state that it is in the middle of trying to get a free cell and scans all memory, checking whether there is an empty cell that was "prepared" for him, i.e. a cell with *Claimed* bit off and *ProcID* being equal to i. If such cell was not found, it scans the cells one-by-one, trying to jam i into the *ProcID* field of the cell; the scan continues until success. When the processor has succeeded, i.e. the *ProcID* field of the current cell C holds i, it checks whether C is his "own" cell by checking the $C.Claimed$ bit. If C was already claimed, the search continues. Otherwise, it sets $C.Claimed$ to "true" and resets $AnnounceGFC(i)$. C is the cell that is eventually returned as the response to the GFC command.

Before returning, the processor ***helps*** all other processors that are in the middle of looking for a free cell. For each such processor p_j, the processor p_i scans all the cells, looking for a cell with j jammed into *ProcID* and the Claimed bit equal to 0. If such a cell was found, it means that there is a cell that only p_j can claim, and in this case p_i returns. If not, it starts participating in the search again ***on behalf of*** p_j. The only difference is that after succeeding with jamming j into *ProcID* of a cell with *Claimed* bit 0, it does not turn on the Claimed bit.

Since each p_i before "helping" p_j scans all the cells and checks whether there is a cell which already belongs to p_j, but was not yet claimed, we have the following lemma.

Lemma 6.3 At any point, for any $1 \leq i \leq n$, there can be at most n ceils with *ProcID* bit equal i and the Claimed bit equal to 0.

Observe, that if during one memory scan no cells were allocated, a free cell is found. The following lemma bounds the number of scans in which a free cell was not found.

Lemma 6.4 At most $2n^2$ ceils can be allocated from the moment a processor announces that it is trying to get a free cell until a cell with *ProcID* being equal to the ID of this processor is actually allocated.

Proof: Assume that the lemma is false. Consider $2n^2$ cells that were allocated after p_i announced that it is trying to get a free cell. There exists at least one processor p_j that has allocated at least ***2n cells among*** these $2n^2$. Observe, that a processor, during a single invocation of **GFC** procedure, can allocate at most n cells – one per each participating processor. Thus, since p_i has announced that it is looking for a free cell, p_j has executed **GFC** procedure from the beginning to the end at least once, which means that one of the $2n$ cells it has allocated has *ProcID* being equal to i, i.e. it belongs to p_i. This contradicts the assumption that there were no cells allocated for p_i. ∎

6.3 **Implementation of** APPEND

The idea is to use the Prev field, which is an atomic Sticky-Byte object, to decide which processor succeeds with appending his cell to the list. The code of APPEND is given in Figure 8. First, look for the current head of the list using procedure FIND-HEAD, which code is given in

```
procedure FIND-HEAD(Cell);
    Head ← ⊥;
    while Head =I and Cell.Next =I do begin
        for all cells TmpCell in memory do begin
            GRAB(TmpCell);
            if "success" and TmpCell.Next ≠ ⊥ and TmpCell.NotHead = false
              then Head ← TmpCell;
              else RELEASE( TmpCell);
        end;
    end;
end.
```

Figure 7: The ***Find-Head*** procedure.

```
procedure APPEND(Cell);
    AnnounceAppend(i) ← 1;
    Head ←FIND-HEAD(Cell);
    Head ← APPEND-INNER(Cell, Head, i);
    AnnounceAppend(i) ← 0;
    for all j such that AnnounceAppend(i) = 1 do begin
        Cell' ← the cell processor p_j is trying to append;
        if GRAB(Cell) = "success"
          then begin
            Head ← APPEND-INNER(Cell', Head, j);
            RELEASE(Cell);
          end;
    end;
    RELEASE( Read);
end.

procedure APPEND-INNER(Cell, Head, ID);
    if Cell.Next ≠ ⊥ then return Head;
    OldHead ← Head.Next;
    while Cell.Next =⊥ do begin
        GRAB(OldHead);
        if "success" then jam OldHead.NotHead with "true";
        try jamming Head.Prev with pointer to Cell;
        RELEASE{ OldHead);
        OldHead ← Head;
        Head ← Head.Prev;
        GRAB(Head);
        if "success" and (Head = Cell or Cell.Next =⊥) then Head.Next ← OldHead;
    end;
    return Bead;
end.
```

Figure 8: The ***Append*** procedure.

Figure 7. To find the current head, scan all the cells looking for a cell with Next field defined, but with *NotHead* being false. This uses the fact that while appending a new cell C to the head *H* of the list, we first define $H.Prev$ to point to C, then we define $C.Next$ to point to *H* and only then we set $H.NoHead$ to true. Observe, that if during a single scan no cells were appended to the list, we will find the head. We will show below that from the moment processor p_i sets $AnnounceAppend(i)$, there are at most ***n cells*** appended to the list before its cell is appended. Therefore, after at most *n* scans of all the cells, we will either find the head of the list or will recognize that the Cell was already appended by some other processor.

After finding the Head of the list, processor p_i checks whether the cell it tries to append is already in the list. If not, it tries to jam the pointer to Cell into the Prev field of the head. If success, it writes the pointer to the head into the Next field of Cell, and indicates that Head is already not the head of the list by setting the *NoHead* field.

A failure to jam the Prev field with the pointer to the Cell means that some other processor succeeded in appending another cell C' to the list. Before writing anything into this cell, p_i tries to GRAB C' in order to prevent initialization while writing. We have to take into account the case where C' was already deleted from the list or is in process of being deleted. (Note that this is the only case where GRAB might fail.) But, as we have described in the previous section, a cell can be removed from the list only if there are at least ***n cells*** ahead of it in the list. By Lemma 6.5, in this case Cell was already appended to the list. On the other hand, if the GRAB succeeds, and the Cell is not yet in the list, then there are less than n cells in the list ahead of C', and therefore it could not have been initialized in between the moment it was added to the list and the GRAB. Hence, it is safe to write into it.

Before returning, p_i checks which processors have announced that they want to add their cell to the list, but have not succeeded in doing so yet. For each such processor p_j, processor p_i checks if the cell that p_j is currently trying to append is already in the list, and if not it appends it as if it was its own cell, by executing the above algorithm.

Lemma *6.5* At most *n* cells can be appended to the list from the moment a processor announces that it is trying to append a cell until this cell is actually appended.

Proof: Assume that the lemma is false. Consider n cells that are appended after p_i announces that it is trying to append a cell. There exists at least one processor that "owns" two cells among them. After appending the fist one of these cells and before appending the second one, this processor has to help all other processors that are trying to append a cell. In particular, it has to help p_j, which leads to a contradiction. ∎

6.4 Space and time complexity

The following theorem states the space complexity of our construction.

Theorem 6.6 Any sequential object can be atomically implemented from $O(n^2 \log n)$ sticky bits and $O(n^2)$ cells, where each cell is large enough to hold a state of the object.

Proof: We allocate the cells from a central pool, and therefore it is enough to count the number of cells which are in the list plus the cells which are being initialized. As we have already pointed

out in the previous section, the number of cells in the list which are not being initialized is $O(n^2)$. By Lemma 6.3, each processor might allocate at most a single cell during execution of **GFC** command per each other processor. In addition, each processor GRABS at most 3 cells at any moment, preventing their initialization. ∎

The time complexity of the construction depends on the time complexity of the safe implementation of the object. Denote by T the time needed by the safe implementation to execute one **access** to the object. In the worst case the universal construction leads to O(n) calls to the safe implementation of the access ($n - 1$ calls in order to reconstruct "current state" and another call in order to actually execute the access). Since **GFC** and APPEND procedures take $O(n^3 \log n)$ time in the worst case and $O(n^2 \log n)$ time if there are no concurrent accesses, this leads to the total of $O(nT + n^3 \log n)$ running time in the worst case and $O(T + n^2 \log n)$ if the object is accessed sequentially.

7 Conclusions and Open Problems

We have described a new approach to implement wait-free shared data objects in a shared-memory multiprocessor, based on the idea of one processor helping another. We have introduced the Sticky Bit object, which can be viewed **as a** memory-oriented generalization of consensus, and showed how to use it to implement this approach. In particular, we showed how to construct a wait-free atomic implementation of any sequential object in terms of $O(n^2 \log n)$ Sticky Bits, where n is the number of processors.

Dolev, Dwork, and Stockrneyer [5] and Chor, Israeli, and Li showed that there is no wait-free implementation of l-bit Test-&-Set by safe registers. Furthermore, Herlihy [7] and Loui and Abu-Amara [10] showed that there is no wait-free protocol that achieves 3-processor consensus using atomic L-bit Test-&-Set for arbitrary k. It is easy to extend these proofs and show that no atomic Read-Modify-Write (RMW) operation on **a** single bit is powerful enough to implement a wait-free 3-processor consensus. On the other hand, observe that 3-processor consensus can be trivially achieved using **a** wait-free atomic RMW on 2 bits. This indicates that there exists a *RMW-hierarchy*, where safe-bits are on the bottom, l-bit RMW is on the fist level, 2-bit RMW is on the second level, etc. The universality of Sticky-Bit proves that the *RMW hierarchy collapses* at the 3rd level, because an **atomic** Sticky-Bit is trivially simulated by an atomic 2-bit RMW.

The results presented in this paper suggest a number of natural direction for further research.

- Though the construction presented in Section 6 uses polynomial amount of memory, it is clearly not efficient. Is it possible to improve the memory and time complexities of this construction ?
- Can we prove a lower bound on the amount of memory needed to implement an arbitrary wait-free atomic object directly from safe bits ?
- Are there any other **natural** objects, besides Sticky Bits, that are universal, easily implementable in hardware, and convenient to use when programming shared-memory multiprocessors ?

Acknowledgments

I would like to thank Nancy Lynch, who introduced me to the subject, for numerous suggestions. I would also like to thank Michael Merritt, Yehuda Afek, Alan Fekete, and Maurice Herlihy for lively discussions and many helpful comments on the various drafts of this paper. I am grateful to Charles Leiserson for his enthusiastic support and encouragement.

References

[1] K. Abrahamson. On Acheiving Consensus Using Shared Memory. In *Proc. 7th Annual ACM Symposium on Principles of Distributed Computing,* **August** 1988.

[2] J. Aspnes and M. Herlihy. A Polynomial Algorithm for Randomized **Asynchronous Con**census using Shared Memory. Unpublished manuscript, 1989.

[3] H. Attiya, D. Dolev, and N. Shavit. Bounded Polynomial Randomized Consensus. In *Proc. 8th Annual ACM* Symposium *on Principles of Distributed Computing,* **August 1989.**

[4] B. Chor, A. Israeli, and M. Li. On Processor Coordination Using Asynchronous Hardware. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing,* **August** 1987.

[5] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. ***J. ACM,*** 34(1):77–97, January 1987.

[6] M. J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Commit with One Faulty Process. ***J. ACM,*** 32(2), April 1985.

[7] M. Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Proc. 7th Annual ACM Symposium on* Principles *of* ***Distributed*** Computing, August 1988.

[8] M. Herlihy and J. Wing. Axioms for Concurrent Objects. ***In Proc. 14th ACM Symp. on Principles of Programming Languages,*** pages 13-26, January 1987.

[9] L. Lamport. On Inter-process Communication, Parts I and II. Technical Report 8, Digital, System Research Center, December 1985.

[10] M. C. Loui and H.H. Abu-Amara. ***Advances In*** **Computing** ***Research.*** Jai Press, 1987.

[11] N. A. Lynch and E.W. Stark. **A** Proof of the Kahn Principle for Input/Output Automata. Technical Report MIT/LCS/TM-349, M.I.T., Laboratory for Computer Science, January 1988.

[12] N. A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. Technical Report MIT/LCS/TR-387, M.I.T., Laboratory for Computer Science, April 1987.

[13] S. A. Plotkin. ***Graph- Theoretic Techniques*** *for* ***Parallel, Distributed, and Sequential Computation.*** PhD thesis, M.I.T., August 1988. (Also available **as** Technical Report TR-430, Lab. for Computer Science, M.I.T., 1988).