SIAM J. COMPUT.
Vol. 23, No. 4, pp. 701–712, August 1994

WAIT-FREE CONSENSUS USING ASYNCHRONOUS HARDWARE*

BENNY CHOR[†], AMOS ISRAELI[‡], AND MING LI[§]

Abstract. This paper studies the wait-free consensus problem in the asynchronous shared memory model. In this model, processors communicate by shared registers that allow atomic read and write operations (but do not support atomic test-and-set). It is known that the wait-free consensus problem cannot be solved by *deterministic* protocols. A *randomized* solution is presented. This protocol is simple, constructive, tolerates up to $n - 1$ processors crashes (where n is the number of processors), and its *expected* run-time is $O(n^2)$.

Key words. asynchronous distributed systems, wait-free protocols, fault tolerance, randomized algorithms, consensus

AMS subject classifications. 68Q22, 90D10

1. Introduction. The problem of reaching consensus among different processors in a distributed environment [20] is one of the most fundamental problems whenever any type of cooperation is to be achieved. The nature of solutions to this problem depends on the properties of communication media, on the reliability of participating processors, and on their relative speeds. In this paper we investigate the consensus problem in a totally asynchronous system, where communication is carried out by shared registers that are atomic with respect to *read* and *write* operations, and up to $n - 1$ out of n processors may fail-stop (i.e., crash).

The consensus problem we study is the following multivalued problem: Every processor starts the protocol with an arbitrary input value (for example, an externally supplied variable or an internally computed constant). Upon termination, each processor decides on an output value. We have two requirements from the output. The first is that all processors that have terminated hold the same output value. The second is that the output value must be one of the input values of the processors. The consensus problem has been extensively studied in the asynchronous message passing model (e.g., [6], [24], [10]). The original version of this work [9] is the first one that studies and solves consensus in this asynchronous shared memory model.

It is convenient to think about all the *read* and *write* operations in terms of a global time model. In this model each such I/O operation takes place in a closed interval on the global-time axis. Atomicity of a register means that every set of *read*s and *write*s from/to this register is equivalent to a sequence in which each interval is shrunk to a distinct point, hence all these operations are totally ordered. We refer the reader to the works of Lamport [19], Herlihy and Wing [16], and Ben-David [5] for precise definitions of atomicity and linearizability. In particular, the techniques of [5] imply that when analyzing protocols that use atomic shared registers, the global time model can be used with no loss of generality.

*Received by the editors December 10, 1990; accepted for publication (in revised form) March 22, 1993. A preliminary version of this paper appeared in *Proceedings of the 6th Annual Association for Computing Machinery Conference on Principles of Distributed Computing*, August 1987, pp. 86–97.

[†]Department of Computer Science, Technion, Haifa 32000, Israel (benny@cs.technion.ac.il). Part of this author's research was carried out at Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts, and was supported in part by National Science Foundation grant MCS81-21431. Research at the Technion was supported in part by United States–Israel Binational Science Foundation grant 88-00282, and by Technion Vice President for Research Fund–E. & J. Bishop Research Fund.

[‡]Department of Electrical Engineering, Technion, Haifa 32000, Israel. Part of this author's research was carried out at Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts, and was supported in part by a Weizmann postdoctoral fellowship and National Science Foundation grant DCR-86-00379.

[§]Department of Computer Science, University of Waterloo, Ontario N2L 3G1, Canada. This author's research was supported in part by Office of Naval Research grant N00014-85-k-0445 at Harvard University, Cambridge, Massachusetts, and by National Science Foundation grant DCR-86-06366 at OSU.

We look for solutions to the consensus problem that satisfy the *wait-free termination* requirement. Wait-free termination means that every processor that is activated a sufficient number of times will decide and terminate. We would like to have a solution that guarantees that every schedule in which a processor is activated at least $k(n)$ times (for some $k(n)$ which is a function of n, the number of processors, but does not depend on the scheduler) leads to termination of that processor. This implies, in particular, that no processor needs to wait for other processors to take steps—it should terminate regardless of whether or not other processors were active in between its own steps (the output value could, however, depend on other processors' activity). Such a requirement is in accordance with the complete asynchrony of the system: It does not make sense to force the very fast processors to wait until a very slow processor makes a move. Furthermore, wait-free termination implies *resilience* to any number of processor crashes.

It is known that wait-free consensus cannot be achieved by *deterministic* protocols, even for systems with $n = 2$ processors. This impossibility result has been proven in the original version of this paper [9] and independently by Loui and Abu Amara [22]. It is also implicit in the work of Dolev, Dwork, and Stockmeyer [13]. All those proofs follow the ideas in the impossibility proof for the message passing model of Fischer, Lynch, and Paterson [17]. The gist of the proof is as follows: First, one shows that there are *bivalent* initial configurations of the system, namely, configurations that can lead to more than one decision value (under different schedulers). After establishing this fact, it is shown that starting from any bivalent configuration, there is an *infinite* scheduler that keeps the deterministic system in a bivalent state.

It is by now a well-known fact in the area of distributed computing that certain problems that cannot be solved by deterministic protocols do admit randomized solutions [24], [21], [6]. It is then only natural that in order to overcome the above-mentioned impossibility result, we employ a randomized protocol, allowing processors to toss coins. We present an efficient randomized protocol, that achieves consensus for systems of size n, using atomic single-writer multireader registers. The protocol is fairly simple and constructive, and its *expected* run-time is $O(n^2)$. This means that for any adversary scheduler, the system reaches a decision after $O(n^2)$ expected number of steps by *all* processors. The protocol uses unbounded size registers (though large values are actually written only with very low probability). The main usage of the unboundedness is to maintain a global order among processors. Processors who maintain larger values get preference over processors holding lower values. Coin flips are used to break possible ties among processors holding equal values.

We briefly discuss other approaches and developments. Loui and Abu-Amara [22] overcome the impossibility of deterministic consensus by using a much stronger communication primitive, namely, atomic test-and-set. Following the publication of the original version of our work [9], various improvements were made: One was designing protocols that operate in the presence of a stronger adversary model than the one used here. Another direction was the development of the so-called "bounded time stamps" [18], [14], and using them in consensus protocols with registers of bounded size. See §§2 and 4 for further details.

The remainder of this paper is organized as follows: In §2 we formally define the model, the class of admissible schedules, and the consensus problem. In §3 we present the protocol, and §4 contains some concluding remarks.

2. Model and definitions.

In this section we define our model of asynchronous concurrent computation, the consensus problem, and the class of schedulers in which we are interested.

An asynchronous concurrent system is a collection of n processors. Every processor P is a (not necessarily finite) state automaton with an internal input register i_P and an internal output register o_P. The input register contains any value v taken from a set V, while the output register has initially the value $\perp$ ($\perp \notin V$) and could be changed once to a value in V. The set of all states of processor P will be denoted by S_P. The set S_P contains a set of states I_P that are the *initial states* of the processor P. States in S_P where o_P contains a value $\neq \perp$ are called the *decision states* of processor P. The set S_P might be infinite. In particular, this enables every internal state to include a description of the whole history of the computation of the processor P.

Processors communicate via *shared registers.* We use atomic single-writer multireader registers: Every shared register can be written by one processor and read by all other processors. Processors execute their programs by taking steps. A step consists of an internal operation, possibly involving coin tosses, and an input/output operation. In the model we consider, these two parts are executed as a single atomic step whenever the processor is scheduled.[1] Formally, every processor P takes steps according to its *transition function* T_P. Each step consists of a single input/output operation, followed by a state transition. The input/output operation could either be "read register r" or "write the value v to register r." In case the communication action is a read, the new state of P depends not only on the old state but also on the value read by this action. The transition function T_P could be either deterministic or randomized. In the latter case, for every state $s \in S_P$, there is a probability measure assigned to the next step. The choice of the actual step is done, according to these probabilities, only when the processor makes its next step. Given an asynchronous system as specified above, a *protocol* is a collection of n transition functions $T_1, \ldots, T_n$, one per processor.

A *configuration* C of the system consists of the state of each processor together with the contents of the shared registers. In an *initial configuration*, every processor is in an initial state, and all shared registers and output registers contain the default value $\perp$. The set of all configurations will be denoted by $\mathcal{C}$. A *step* takes one configuration to another by activating a single processor P. A *run* of length ℓ is a sequence of ℓ steps. Each run has an associated *schedule* that is a sequence of ℓ processors, numbered according to the order of processors that take steps in that run. We denote schedules, finite or infinite, by a list of processor numbers, e.g., $(2, 3, 3, 2, 1)$. If S is a finite schedule, then we denote by $S \circ i$, where i is any processor number, the schedule obtained from the schedule S by concatenating the number i to the end of S. We say that processor P is activated k times in a run if P appears k times in its schedule. The *history* $\mathcal{H}$ of a run is the sequence obtained by interleaving the sequence of configurations with the steps in the run, starting with the initial configuration. For a finite run, we refer to the last configuration in its history as the *current* configuration.

When arguing about randomized protocols, the power of the scheduler crucially depends on its adaptivity (see [8] for a discussion of this issue). *Adaptive schedulers* can use information derived from the state of the system and its history in making scheduling decisions. Formally, an admissible *scheduler* $\mathcal{S}$ in our system is a mapping from $\mathcal{H}$ into the set of n processors. Given the configuration of the system, the scheduler picks the next processor that is to take a step. The scheduler could either be a deterministic mapping or a randomized one. The scheduler is best viewed as an adversary that tries to prevent us from reaching our goal. Under the definition, this adversary scheduler is adaptive, and it has complete knowledge on the state of every processor and on the contents of the shared registers during the entire history.[2] In case the processors are randomized, the scheduler could also base its choices on the outcome

[1] An alternative approach separates the atomic operations to internal operations, input operations, and output operations.

[2] In the more refined level of atomicity, where internal operations, input operations, and output operations are separate, the adversary is even stronger. For example, it knows what a processor is about to write before scheduling that processor. For more details, see §4.

of past coin flips. We do not allow it, though, to be able to predict *future* randomized moves of the processors. This is a necessary requirement if randomization is to be helpful at all, and it is used in all algorithms where randomization is employed, e.g., [24], [21], [6]. In particular, in randomized protocols, a processor might be in a state in which the adversary does not know which input/output operation will be taken by that processor, before the action takes place. Given a history $\mathcal{H}$ and a scheduler $\mathcal{S}$, the runs that can be produced by $\mathcal{S}$, extending $\mathcal{H}$, on some possible randomized choices are called the runs *compatible* with $\mathcal{H}$ and $\mathcal{S}$. Notice that if both processors and scheduler are deterministic, then there is a single compatible run extending $\mathcal{H}$.

We say that a configuration C is reachable from history $\mathcal{H}$ with schedule $\mathcal{S}$ if there is a run compatible with $\mathcal{H}$ and $\mathcal{S}$ that leads to configuration C. We say that a configuration has a decision value v if some processor P is in a decision state with its output register o_P containing $v \neq \perp$.

A *randomized consensus protocol* is designed for an asynchronous system of n processors ($n \geq 2$). The protocol specifies a set V of possible inputs whose cardinality is at least two (otherwise the problem is trivial). It is required to satisfy the following properties:

(1) **Consistency:** for every schedule, no configuration reachable from an initial configuration has more than one decision value.

(2) **Nontriviality:** if processor P has decided on value v in a run, then v is an input value for at least one processor.

(3) **Randomized wait-free termination:** each processor must decide after taking a finite expected number of steps. Formally, there is a probability function f from the natural numbers into the interval [0, 1] $\left(\sum_{k=1}^{\infty} f(k) = 1\right)$, satisfying $\sum_{k=1}^{\infty} kf(k) < \infty$, such that for every initial configuration C_0 and for every admissible scheduler, if a processor P was activated k times by the scheduler, then the conditional probability that P is in a decision state, conditioned on P not being in a decision state after its previous activation, is at least $f(k)$.

We required that randomized consensus protocols will never err. The randomization effects only the running time of the protocol and not its correctness. There could be a positive probability for arbitrary long nonterminating runs, but this probability should be very small (converging to 0 with the length of the run), so that the expected running time is bounded.

3. Wait-free consensus protocol. The high-level structure of the protocol is as follows: In every point of an execution, each processor holds a preferred value (which is a potential decision value) and a confidence level (which is a nonnegative integer). Initially, the confidence level is 0, and the preferred value of the processor is its input value. Both the preferred value and the confidence level are written by each processor into a shared register, which can be read by all others. Processors compare their confidence levels, and if a large enough gap forms, the leading processor decides on its preferred value. In case of ties, processors increment their confidence level. In order to prevent live locks, where competing processors concurrently increment their confidence ad infinitum, coin flips are used. Confidence levels can possibly reach any nonnegative integer, which means that registers of unbounded size are used by the protocol. There is a positive (though very small) probability for very large numbers to be written into the shared registers. This probability decreases to 0 when the numbers increase to infinity.

To simplify the description of the protocol, we will say that a processor is on node i if its confidence level is i. The initial node is node 0. Before deciding and terminating, the processor moves to a special node, denoted by ∞ (this move facilitates the design and analysis of the protocol). Each processor starts execution by writing its input value in its register while staying on the start node. The steps of each processor in any given history H of the protocol are divided into *phases*. In each *phase* a processor reads the registers of all other processors, computes a new value, and writes it in its own register. A processor decides if it is at least two nodes ahead of all other processors with contending values. Thus, by the time of decision, processors with contending values are at least one step behind and will change their preferred value to that of the leading processor. There could be a situation with ties. The protocol resolves ties by having the option of advancing (to the next node) or not advancing, according to the outcome of coin tossing. This is where the use of randomization overcomes the deterministic impossibility result. In a bivalent configuration, only some of the choices made by some processors lead to another bivalent configuration. Other choices could lead to a univalent configuration. The adversary does not know which choices the processor will make *before* scheduling it, because the choice is made by flipping a coin.

If the coin used by every processor (in choosing whether to advance or not) would be unbiased, then with high probability, about half the contenders would advance. Those lagging behind would then join them, and again we would be in a tied situation. While such protocol, using unbiased coins, would satisfy the requirements of randomized consensus, it would lead to exponential (in n) expected running time (for an appropriate adversary strategy). To be more efficient, our protocol tries to have, with high probability, only one successful advancement out of n attempts. Leading processors in a tied situation flip a *biased* coin and advance only with small probability. Intuitively, this probability should be $\theta\left(\frac{1}{n}\right)$. The specific value we use, $\frac{1}{2n}$, is based on calculations done to minimize the expected running time.

It is dangerous to let a lagging processor decide, even if all leading processors have the same preferred value. The reason is that another processor might advance substantially after its value was last read by the lagging processor. The lagging processor, who thinks all leading processors have the same value, would in fact be wrong. Therefore, in our protocol, processors lagging behind never decide, and they always advance. A lagging processor who sees all leading processors with the same preferred value changes its own value. If the lagging processor sees conflicts at the top, it retains its old preferred value. In both cases, the lagging processor advances. If it is no more than two nodes behind the maximum, it advances by one. If it more than two nodes behind the maximum, it "jumps" to a point that is the maximum minus two. This shortcut allows the protocol to converge quickly to a decision, even if it starts from a configuration where one processor is way behind other active processors (e.g., if it just woke up). Thus, our upper bound on the expected running time will be valid starting from *every* reachable configurations, and not just the initial one.

To continue the description of the protocol, some definitions are introduced. Assume that processor P_i has just finished all the read steps in its jth phase, a phase we denote by ϕ^i_j. Let maxnode^i_j be the maximum node on which P_i sees a processor during ϕ^i_j (including itself). Using the data collected on ϕ^i_j, P_i computes two sets of processors, L and AL. The set L (the leaders) contains the processors whose node, as read by P_i , is maxnode^i_j. The set AL (the almost leaders) is the set of processors whose node, as read by P_i , is $\text{maxnode}^i_j - 1$. Denote by L^i_j (AL^i_j) the set L (AL) computed by P_i in phase ϕ^i_j.

A processor P_i terminates after the write step of phase ϕ^i_j in one of two cases:

T_1 If another processor has already terminated. The decision value is the value of the terminating processor.

T_2 If P_i itself is in the set L^i_j and all processors in the set $L^i_j \cup AL^i_j$ have the same *pref.* The decision value is the common *pref.*

We say that a processor P_i is *committed to terminate* in phase ϕ^i_j if it completed all the read steps of the phase and one of the termination conditions T_1 or T_2 holds (so its next step is a write, after which P_i decides and terminates). If none of the termination conditions holds, then P_i either moves to a new node or it stays put. The motivation behind the protocol design is to create a single leader. If P_i is a leader (that is, $P_i \in L^i_j$), then the new node to which

```
     newreg := (input, 0)
     reg_1 :=write(newreg)
     repeat
          $v_1$ := newreg
          for $i$ := 2 through $n$ do $v_i$ :=read ($reg_i$) od
          maxnode := $\max_{1 \le i \le n}\{v_i.node\}$
          $L := \{P_i : v_i.node = maxnode\}$
          $AL := \{P_i : v_i.node = maxnode - 1\}$
($T_1$)   if $\exists i\ v_i.node = \infty$
            then $reg_1$ := write($v_i.pref$, $\infty$), decide $v_i.pref$ and halt.
($T_2$)   elseif $P_1 \in L$ and pref of all processors in $L \cup AL = v_1.pref$
                 then $reg_1$ :=write($v_1.pref$, $\infty$), decide $v_1.pref$ and halt.
          elseif $P_1 \in L$
                 then
                        newreg.pref:= $v_1.pref$
                        newreg.node := $v_1.node + 1$
                        toss a biased coin with $1/2n$ probability of heads
                        if tails (this occurs with probability $1 - 1/2n$)
                          then newreg := $v_1$ (retain old value)
                        endif
          elseif $maxnode - v_1.node \le 2$
                 then
                        newreg.node := $v_1.node + 1$
                        if all leading processors have the same pref
                          then
                               newreg.pref:= pref of leading processor
                          else
                               newreg.pref:= $v_1.pref$
          else ($maxnode - v_1.node \ge 3$)
               newreg.node := $maxnode - 2$
               newreg.pref:= pref of processor with minimum index in $L$
          endif
          $reg_1$ :=write(newreg)
     until decision is made
```

FIG. 1. *The n processor protocol (for P_1).*

it can move is the successor of its node. However, it moves to its successor node only with probability[3] $1/2n$. With probability $1 - 1/2n$, P_i stays on its current node. In both cases, the leader P_i retains its old preferred value.

If P_i is not a leader, then it always moves to a new node. Let k denote P_i's node during ϕ^i_j. If $\text{maxnode}^i_j - k \le 2$, then P_i moves to the successor of its node, $k + 1$. In this case, P_i's new preferred value is determined as follows: If all processors in L^i_j have the same *pref*, this value is the new *pref* of P_i. If this is not the case, then P_i keeps its own *pref*.

If P_i is not a leader and $\text{maxnode}^i_j - k \ge 3$, then P_i moves to node $\text{maxnode}^i_j - 2$. In such case, we say that P_i *jumps*. In case of a jump, the new *pref* of P_i is the *pref* of the processor in L^i_j with the minimal index.

LEMMA 3.1. *Let H_0 be an initial segment of a history H of an arbitrary execution of the protocol. If in H_0 no processor reaches node i with preferred value v, then for any node $j > i$, no processor reaches j, in H_0, with preferred value v.*

Proof. Assume, toward a contradiction, that H is the history of an execution not satisfying the lemma. Let H_0 be an initial segment of H of minimal length that violates the lemma. This

[3]This specific probability $1/2n$ was chosen in order to optimize the expected running time.

means that there is a node i such that in H_0 no processor reaches i with preferred value v and there is a processor P_k that reaches a node $j > i$ with preferred value v.

Let ϕ_ℓ^k be the phase in which P_k moves to node j. If P_k jumps to j, then, by the protocol, at least one processor $P_m \in L_\ell^k$ has been on node $j + 2$ with preferred value v before this jump. Since $j+2 > i$, this contradicts the minimality of H_0. We can therefore assume that P_k moves to node j from node $j - 1$ without jumping. First, we show that the preferred value of P_k at $j - 1$ is $v' \neq v$. If $j - 1 = i$ then, this is true by our assumption that in H_0 no processor prefers the value v on node i. If $j - 1 > i$, then this is true because of the minimality of H_0. Having established this claim, we observe that by the protocol, in order for P_k to change its preferred value from v' to v while moving from $j-1$ to j it has to see all processors in L_ℓ^k with preferred value v. We now show that for any possible value of maxnode$_\ell^k$ this is impossible. In ϕ_ℓ^k P_k is on $j - 1 \geq i$. Therefore maxnode$_\ell^k \geq j - 1 \geq i$. If maxnode$_\ell^k = i$, then by our assumption no processor has preferred value v on i. If maxnode$_\ell^k > i$, then as we have shown above, no processor in L_ℓ^k has preferred value v. □

LEMMA 3.2. *Let P_k be the first processor reaching node i with preferred value v. Then P_k does not jump to i, and its preferred value on node $i - 1$ is also v.*

Proof. Let ϕ_ℓ^k denote the phase in which P_k moves to node i, and let H_0 be the initial segment of the history of the execution that ends with the last read step of P_k in ϕ_ℓ^k. In H_0 no processor has reached node i with preferred value v. Thus, by Lemma 3.1, no processor has reached any node $j \geq i$ in H_0 with preferred value v. Therefore P_k sees no leader on any node $j \geq i$ with preferred value v during ϕ_ℓ^k. According to the protocol, P_k does not jump to node i with preferred value v in ϕ_ℓ^k. Since P_k reaches node i with preferred value v, this argument implies that P_k does not jump to node i.

Assume, by way of contradiction, that P_k changes its preferred value from v' to v while moving from $i - 1$ to i at the end of ϕ_ℓ^k. This happens only if *pref* of all processors in $L_{k,\ell}$ is v. Consider the following cases:

Case 1: maxnode$_\ell^k = i - 1$. In this case $P_k \in L_\ell^k$. By the protocol P_k keeps its preferred value while moving to i. Contradiction.

Case 2: maxnode$_\ell^k = i$. In this case the preferred value of all processors in L_ℓ^k does not equal v, since we assumed that P_k is the first processor reaching i with preferred value v. Contradiction.

Case 3: maxnode$_\ell^k > i$. In H_0 no processor has been on node i with preferred value v, and by Lemma 3.1, in this execution no processor has preferred value v on maxnode$_\ell^k$ before the completion of ϕ_ℓ^k. Contradiction.

We conclude that P_k prefers the value v during ϕ_ℓ^k, its last phase on node $i - 1$. By the protocol, processors retain their preferred values when staying on the same node. This means that P_k prefers the value v during all phases it executes while residing on node $i - 1$. □

LEMMA 3.3. *Let H be the history of an arbitrary execution of the protocol. Let P_j be a processor committed to terminate, in H, by T_2. If P_j is committed to terminate on i with decision value v, then in H, no processor reaches i with a preferred value $v' \neq v$.*

Proof. Assume, by way of contradiction, that P_ℓ is the first processor reaching i with preferred value $v' \neq v$. By Lemma 3.2, P_ℓ moves to node i without jumping, and the preferred value of P_ℓ on $i-1$ is also v'. Let ϕ_k^j be the phase in which P_j is committed to terminate with v. At the beginning of ϕ_k^j, P_ℓ is on node $m < i - 1$. (Otherwise P_j sees P_ℓ as either a leader or an almost leader during phase ϕ_k^j, and condition T_2 does not hold.) After P_ℓ advances to $i - 1$ it starts a new phase ϕ_r^ℓ. In ϕ_r^ℓ, P_ℓ reads the values of all processors. The leaders in L_r^ℓ are on a node $\geq i$. The value preferred by all these leaders before P_ℓ moves there is v, as the only value preferred at i is v. By the protocol P_ℓ takes v as its new preferred value and advances to i. Contradiction. □

LEMMA 3.4. *Let H be the history of an arbitrary execution of the protocol. Assume that in H a single processor P_i moves from node m to node $m + 1$ as a result of a successful coin toss. (That is, all other processors that tried to move from m to $m + 1$ as a result of a coin toss fail to do so.) If P_i's preferred value on node m is v, then all processors reaching node $m + 1$ in H have v as their preferred value on node $m + 1$.*

Proof. Assume, by way of contradiction, that the lemma does not hold. Let P_j $(i \neq j)$ be the first processor reaching node $m + 1$ with preferred value v' $(v' \neq v)$ in H. By Lemma 3.2, P_j does not jump to node $m + 1$. By the supposition, P_j does not flip a coin when moving from node m to node $m + 1$. This implies, by the protocol, that P_j is not one of the leaders during the phase ϕ_ℓ^j in which it moves to node $m + 1$ (i.e., $P_j \notin L_\ell^j$). Since all processors who move to node $m + 1$ in H *before* P_j's move prefer the value v on node $m + 1$, it follows from Lemma 3.1 that all leading processors (those in L_ℓ^j) prefer the value v. All these leaders are on a node $\geq m + 1$ and according to the protocol P_j moves to $m + 1$ with preferred value v in phase ϕ_ℓ^j—contradiction. □

LEMMA 3.5. *Let H be the history of an arbitrary execution of the protocol. Assume that in H, processor P_i is the first processor who moves from node m to node $m + 1$ as a result of a successful coin toss. Then after this write step of P_i, any other processor P_j makes at most one attempt to move from node m to node $m + 1$ as a result of a coin toss.*

Proof. Consider the execution after P_i's move. If P_j succeeds in its first attempt to move from node m to node $m + 1$ as a result of a coin toss, then we are done. If P_j fails, then it stays on node m. In the phase following the failed attempt, P_j is not a leader (as it sees P_i at least one node ahead). By the protocol, P_j does not flip a coin in its next phase on node m and moves to a new node in this phase. □

THEOREM 3.6. *The n processor protocol is consistent.*

Proof. Let H be an arbitrary history of the protocol. It is easy to see that a processor terminates by T_1 with value v only if some other processor terminated earlier with v by T_2. Therefore, it suffices to show that in H all processors that terminate by T_2 have the same decision value. Let i be a minimal node on which some processor is committed to terminate by T_2. Without loss of generality assume that P_j terminates on i with value v. In order to prove that the protocol is consistent we will show that no other processor is committed to terminate on any node with a value $v' \neq v$. By the protocol, the first processor committed to terminate with value v' is committed by T_2. Since i is a minimal node on which any processor is committed to terminates by T_2, no processor is committed to terminate by T_2 on any node $k < i$. By Lemma 3.3 no processor reaches i with any preferred value $v' \neq v$. By the protocol, this implies that no processor is committed to terminate by T_2 on node i with decision value v'. This also implies, by Lemma 3.1, that no processor reaches any node $k > i$ with preferred value $v' \neq v$. Therefore, by the protocol, no processor is committed to terminate by T_2 with decision value $v' \neq v$. □

We now proceed to analyze the expected running time of the multiprocessor protocol.

THEOREM 3.7. *Let C be any reachable configuration of the n processor system and $\mathcal{A}$ an arbitrary adversary scheduler. If $\mathcal{A}$ schedules the processors such that at least $15n$ entire phases are executed following C, then with probability ≥ 0.4534, at least one processor decides and terminates.*

Proof. Let m $(m \geq 0)$ be the maximal node on which any processor resides at C. By the protocol, any processor P_i that executes an entire phase following C finds that some processor resides on a node $\geq m$ and will subsequently move to a node $j \geq m - 2$ in the write step of this phase. After completing two additional phases, P_i reaches node $j \geq m$. (Recall that if a processor is not among the leaders in some phase, then it does not flip a coin and traverses at least one edge in the **write** step of that phase.)

Thus, of the $15n$ entire phases that are executed following C, at most $3n$ are executed by processors residing on nodes smaller than m. Therefore, at least $12n$ entire phases are executed, following C, by processors residing on nodes greater than or equal to m.

Consider the first processor P_i that is scheduled to make a **write** step while residing on node m following C. If some other processor has already decided before this write step of P_i, then we are done. If no other processor has yet decided, then since m is the maximal node in C, the value maxnode that P_i maintains at the time of this write equals m. If P_i decides and moves to ∞, then we are done. Otherwise, according to the protocol, P_i flips a coin when making its write step, and if it succeeds (this happens with probability $1/2n$), it moves to node $m + 1$.

We base our analysis on the following two events:

- E_1: Of the first $4n$ attempts to move from node m to node $m + 1$ as a result of a coin toss following C, *exactly* one succeeds. Subsequent to the successful move, all attempts to move from node m to node $m + 1$ as a result of a coin toss fail.
- E_2: Of the first $4n$ (or fewer) attempts to move from node $m + 1$ to node $m + 2$ as a result of a coin toss following C, *at least* one succeeds.

Using Lemma 3.5, the number of subsequent attempts to toss a coin on node m, after the first successful toss on node m, is less than or equal to $n - 1$. Since the adversary is unable to predict the outcome of a write that uses coin tossing before the action takes place, we have

$$Pr(E_1) \geq \left(\overbrace{\frac{1}{2n}}^{\text{success in 1st}} + \overbrace{\left(1 - \frac{1}{2n}\right)\frac{1}{2n}}^{\text{success in 2nd}} + \ldots + \overbrace{\left(1 - \frac{1}{2n}\right)^{4n-1}\frac{1}{2n}}^{\text{success in } 4n\text{th}} \right)$$

$$\cdot \overbrace{\left(1 - \frac{1}{2n}\right)^{n-1}}^{\text{no subsequent success}}$$

$$= \left(1 - \left(1 - \frac{1}{2n}\right)^{4n}\right)\left(1 - \frac{1}{2n}\right)^{n-1}.$$

If E_1 occurs, then at most $4n + n - 1 = 5n - 1$ of the entire phases that are executed on nodes $\geq m$ following C involve an attempt to move from node m to $m + 1$ by tossing a coin. At most $n - 1$ additional phases can involve moving from node m to $m + 1$ without tossing a coin. Thus overall, if E_1 occurs, then at most $6n - 2$ of the entire phases that are executed on nodes $\geq m$ following C are executed by processors residing on node m. This implies that at least $(12 - 6)n + 2 = 6n + 2$ entire phases are executed, following C, by processors residing on nodes greater than or equal to $m + 1$.

Consider the first processor P_i that is scheduled to make a **write** step while residing on node $m + 1$ following C. If some other processor has already decided before this write step of P_i, then we are done. If no other processor has yet decided, then since m is the maximal node in C, the value maxnode that P_i maintains at the time of this write must equal $m + 1$. If P_i decides and moves to ∞, then we are done. Otherwise, according to the protocol, P_i flips a coin when making its write step, and if it succeeds (this happens with probability $1/2n$), it moves to node $m + 2$. If P_i does not succeed, then by the same reasoning the next processor that resides on node $m + 1$ flips a coin when it is scheduled to write , and so on. Thus, until at least one processor succeeds, all processors that reside on node $m + 1$ try to move to node $m + 2$ by flipping a coin. The probability that out of the first $4n$ attempts at least one is successful satisfies

$$Pr(E_2 \mid E_1) \geq 1 - \left(1 - \frac{1}{2n}\right)^{4n} .$$

The sequence $\{(1 - \frac{1}{2n})^{n-1}\}_{n=2}^{\infty}$ is monotonically decreasing to the limit $\frac{1}{\sqrt{e}}$. The sequence $\{1 - (1 - \frac{1}{2n})^{4n}\}_{n=2}^{\infty}$ is monotonically decreasing to the limit $1 - \frac{1}{e^2}$ (the limits can easily be verified, using the fact that $\{(1 - \frac{1}{k})^{k}\}_{k=2}^{\infty}$ monotonically increases to $\frac{1}{e}$). Combining these properties with the two inequalities above, we have (for $n \geq 2$)

$$\begin{aligned} Pr(E_2 \cap E_1) &\geq \left(1 - \left(1 - \frac{1}{2n}\right)^{4n}\right)^2 \left(1 - \frac{1}{2n}\right)^{n-1} \\ &\geq \left(1 - \frac{1}{e^2}\right)^2 \cdot \frac{1}{\sqrt{e}} \\ &> 0.4534 . \end{aligned}$$

If E_2 occurs, then at least one of the first $4n$ (or fewer) attempts succeeds. Using Lemma 3.5, the number of subsequent attempts to toss a coin on node $m + 1$ after the first successful toss is less than or equal to $n - 1$. At most $n - 1$ additional phases can involve moving from node $m + 1$ to $m + 2$ without tossing a coin. Thus overall, at most $6n - 2$ of the entire phases that are executed on nodes $\geq m + 1$ following C are executed by processors residing on node $m + 1$. This implies that at least $(6n + 2) - (6n - 2) = 4$ entire phases are executed, following C, by processors residing on nodes greater than or equal to $m + 2$.

In particular, it follows that at least one processor P_i completes a phase, including a write step, while residing on node $m + 2$. By Lemma 3.4, if E_1 occurs, all processors reaching node $m + 1$ have the same preferred value v on node $m + 1$. But since on node $m + 1$ all processors prefer the same value v, Lemma 3.1 implies that on $m + 2$ all processors prefer v as well. Thus the first processor P_i that completes a phase while residing on node $m + 2$ finds out during that phase that all leaders (processors on $m + 2$) and almost leaders (processors on $m + 1$) prefer v. P_i thus moves to the decision node ∞ by T_2, and terminates.

Therefore, with probability at least 0.4534, at least one processor terminates after $15n$ phases are completed following C. □

Using Theorem 3.7, we can easily give an upper bound on the expected time until some processor decides, starting from any reachable configuration C. Dividing the execution into blocks such that in each block exactly $15n$ phases are completed, we know that the probability of termination in each block is at least 0.4534. The expected number of entire phases is thus $15n/0.4534 < 33.1n$ operations. In terms of *elementary* operations (atomic read, atomic write), each entire phase involves exactly n elementary operations. At most $n(n - 1)$ initial elementary operations can belong to phases whose execution have already begun. Thus, the expected number of elementary operations until at least one processor decides is at most $33.1n^2 + n(n - 1) < 35n^2$. This implies the following.

THEOREM 3.8. *The n processor protocol is a randomized wait-free consensus protocol. Starting from any reachable configuration, the expected number of elementary steps until at least one processor decides is less than $35n^2$.*

4. Concluding remarks.

The analysis of the expected running time of our n processors protocols relied on the inability of the adversary to predict the outcome of a write that uses coin tossing, before the action takes place. Following the publication of the original version of our paper [9], Abrahmson [1] considered a stronger adversary model, where the outcome of the coin toss that is used in the next write step is known to the adversary before the step takes place. In this adversary model, the scheduling choices can be based on the outcome of the coin. Abrahmson modified our protocol and produced one that works in the presence of the strong adversary but has exponential ($2^{O(n^2)}$) expected running time. Subsequently, this was dramatically improved by Aspnes and Herlihy [3], who designed an efficient wait-free consensus protocol for this strong adversary model, with $n^{O(1)}$ expected running time. The protocol of Aspnes and Herlihy employs the same basic structure of our protocol, namely an incremental walk on the line of nonnegative integers. It introduces novel ideas from the theory of random walks in the implementation of the coin flips. Improved algorithms that use *bounded* shared registers and work in the presence of the strong adversary were later designed by Attiya, Dolev, and Shavit [4], Aspnes [2], and Saks, Shavit, and Wohl [25]. (Most of these algorithm solve the somewhat simpler problem of *binary* consensus, where the input set is $\{0, 1\}$.) The expected running time of the later protocol is $\Theta(n^3)$ elementary steps. This has subsequently been improved by Bracha and Rachman [7] to an $O(n^2 \log n)$ consensus protocol. Despite these improvements, our protocol remains the most efficient of which we know for the model considered in this paper and is a strong candidate for practical consensus protocols. By bounding the size of the shared registers in our protocol to, say, 128 bits per processor, we get a protocol that still never errs and has probability less than 2^{-56} of nontermination.

We view the possibility of achieving wait-free consensus as a fundamental tool in shared memory systems and believe it is only the first step in a promising direction. The subsequent results of Herlihy [15] and Plotkin [23] on wait-free implementation of sequential objects and of Chor, Moscovici, and Nelson [11], [12] on solvability of distributed decision tasks and distributed interactive tasks seem to support this belief.

Acknowledgments. We thank Maurice Herlihy for pointing out an error in the original version of the protocol. Discussions with Nissim Francez and Lenny Pitt that helped initiating this work are also acknowledged.

REFERENCES

[1] K. Abrahmson, *On achieving consensus using a shared memory*, Proc. 7th ACM Conference on Principles of Distributed Computing, August 1988, pp. 291–302.

[2] J. Aspnes, *Time- and space-efficient randomized consensus*, Proc. 9th ACM Conference on Principles of Distributed Computing, August 1990, pp. 325–331.

[3] J. Aspnes and M. Herlihy, *Fast randomized consensus using shared memory*, J. Algorithms, 11 (1990), pp. 441–461.

[4] H. Attiya, D. Dolev, and N. Shavit, *Bounded polynomial randomized consensus*, Proc. 8th Annual ACM Symposium on Principles of Distributed Computing (1989), pp. 281–293.

[5] S. Ben-David, *The global time assumption and semantics for concurrent systems*, Proc. 7th Annual ACM Symposium on Principles of Distributed Computing (1988), pp. 223–231.

[6] M. Ben-Or, *Another advantage of free choice: completely asynchronous agreement protocols*, Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing (1983), pp. 27–30.

[7] G. Bracha and O. Rachman, *Randomized consensus in expected $O(n^2 \log n)$ operations*, Proc. 5th Internat. Workshop on Distributed Algorithms (1991), Lecture Notes in Comput. Sci. 579, Springer-Verlag, New York, pp. 143–150.

[8] B. Chor and C. Dwork, *Randomized algorithms for distributed agreement—A survey*, Advances in Computing Research, Vol. 5, Randomness and Computations, S. Micali, ed., JAI Press, 1989, pp. 443–497.

[9] B. Chor, A. Israeli and M. Li, *On Processors Coordination Using Asynchronous Hardware*, Proc. 6th ACM Conference on Principles of Distributed Computing, 1987, pp. 86–97.

[10] B. Chor, M. Merritt, and D. Shmoys, *Simple constant-time consensus protocols in realistic failure models*, J. ACM, 36 (1989), pp. 591–614.

[11] B. Chor and L. Moscovici, *Solvability in asynchronous environments*, Proc. 30th IEEE Conference on the Foundations of Computer Science, 1989, pp. 422–427.

[12] B. Chor and L. Nelson, *Resiliency of interactive distributed tasks*, Proc. 10th ACM Conference on Principles of Distributed Computing, 1991, pp. 37–49.

[13] D. Dolev, C. Dwork, and L. Stockmeyer, *On the minimal synchronism needed for distributed consensus*, J. Assoc. Comput. Mach., 34 (1987), pp. 77–97.

[14] D. Dolev and N. Shavit, *Bounded concurrent time stamps systems are constructible*, Proc. 21st ACM Symposium on Theory of Computing, 1989, pp. 454–466.

[15] M. Herlihy, *Impossibility and universality results for wait free synchronization*, Proc. 7th Annual ACM Conference on Principles of Distributed Computing, 1988, pp. 276–290.

[16] M. Herlihy and J. Wing, *Linearizability: A correctness condition for shared objects*, ACM Trans. Program. Languages and Systems, 12 (1990), pp. 463–492.

[17] M. Fischer, N. Lynch, and M. Paterson, *Impossibility of distributed consensus with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374–382.

[18] A. Israeli and M. Li, *Bounded time stamps*, Proc. 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 371–382.

[19] L. Lamport, *On interprocess communication*, Distributed Computing, Vol. 1, 1986, pp. 77–101.

[20] M. Pease, R. Shostak, and L. Lamport, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228–234.

[21] D. Lehmann and M. Rabin, *On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem*, Proc. 8th Principles of Programming Languages 1981, pp. 133–138.

[22] M. C. Loui and H. H. Abu-Amara, *Memory requirements for agreement among unreliable asynchronous processes*, Advances in Computing Research, JAI Press, 1987, pp. 163–183.

[23] S. Plotkin, *Sticky bits and the universality of consensus*, Proc. 8th Annual ACM Conference on Principles of Distributed Computing, August 1989, pp. 159–175.

[24] M. Rabin, *The choice coordination problem*, Acta Inform., 17 (1982), pp. 121–134.

[25] M. Saks, N. Shavit, and H. Wohl, *Optimal time randomized consensus — making resilient algorithms fast in practice*, Proc. 2nd ACM Symposium on Discrete Algorithms, 1991, pp. 351–362.