Randomized Wait-Free Consensus using An Atomicity Assumption

Ling Cheung*

Department of Computer Science, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. We present a randomized algorithm for asynchronous wait-free consensus using multi-writer multi-reader shared registers. This algorithm is based on earlier work by Chor, Israeli and Li (CIL) and is correct under the assumption that processes can perform a random choice and a write operation in one atomic step. The expected total work for our algorithm is shown to be $O(N \log(\log N))$, compared with $O(N^2)$ for the CIL algorithm, and $O(N \log N)$ for the best known weak adversary algorithm. We also model check instances of our algorithm using the probabilistic model checking tool PRISM.
Keywords: Asynchronous Consensus, Randomized Algorithms, Wait-Free Termination, Weak Adversary, Probabilistic Model Checking

1 Introduction

Distributed consensus refers to a class of problems in which a set of parallel processes exchange messages in order to agree on a common preference. Initially, each process is given an input value from a fixed, finite domain and, at the end of the algorithm, each non-faulty process outputs a decision value. Correctness requirements are typically formulated as follows.

- *Validity*: the output of any non-faulty process must have been the input of some process.
- *Agreement*: all non-faulty processes decide on the same value.
- *Termination*: every non-faulty process decides after a finite number of steps.

As shown in [FLP85], there exists no deterministic algorithm that solves distributed consensus in a setting of asynchronous communication with undetected process failure. Nonetheless, many efficient solutions exist under stronger assumptions (e.g. partial synchrony [DLS88] and failure detection [ACT00]) or weaker correctness requirements (e.g. probabilistic termination [CIL87]).

Our algorithm falls into the category of *randomized consensus algorithms*, where processes may use coin tosses to determine their course of actions. In this setting, termination is weakened to a probabilistic statement: the set of all non-terminating executions has probability 0. We refer to [Asp03] for a comprehensive overview on randomized consensus.

* Supported by DFG/NWO bilateral cooperation project Validation of Stochastic Systems (VOSS2).

The first randomized consensus algorithm was proposed by Chor, Israeli and Li [CIL87,CIL94]. It satisfies the following termination condition.

– *Probabilistic wait-free termination*: with probability 1, each non-faulty process decides after a finite number of steps.

We adopt the same requirement. In fact, the logical structure of our algorithm closely resemble that in [CIL94], while we borrow ideas from [Cha96] to reduce the amount of shared and local data. We shall refer to [CIL94] as the original CIL algorithm and our own as the modified CIL algorithm.

Adversary Models and Work Bounds. To prove probabilistic termination, we must reason about probability distributions on the set of executions. These distributions are induced by the so-called *adversaries*, which are functions from finite histories to available next steps.

The strength of an adversary varies according to the amount of information it can extract from a finite history. The *strong* adversaries have access to complete history of all processes and shared registers. Some weaker forms, such as *write-oblivious* and *value-oblivious*, delay the adversary's knowledge of outcomes of internal coin tosses. Clearly, a stronger adversary model permits more possibilities and therefore renders consensus more difficult. Consensus against strong adversaries is shown to be $\Omega(N^2/\log^2 N)$ in expected total work, where N is the number of processes participating in the algorithm [Asp98]. The best known algorithms achieve expected $O(N^2 \log N)$ total work [BR91] and $O(N \log^2 N)$ per process [AW96]. Against write-oblivious adversaries, one can achieve expected $O(\log N)$ per process work and $O(N \log N)$ total work [Aum97]. Against value-oblivious adversaries, the fastest algorithm is $O(N \log N\, e^{\sqrt{\log N}})$ in a single-writer single-reader (SWSR) setting [AKL99][1].

Our adversary model takes the form of an atomicity assumption: processes can perform a random choice and a write operation in one atomic step. In particular, the process increments its round number if and only if the coin lands heads; then immediately it writes 1 to the memory location $\mathsf{mem}(r, v)$, where r is the round number *after* the coin toss and v is the current preference. This amounts to saying that the adversary cannot distinguish between the two locations $\mathsf{mem}(r, v)$ and $\mathsf{mem}(r+1, v)$. The original CIL algorithm relies on a similar atomicity assumption[2] and achieves expected $O(N^2)$ total work [CIL94]. In the present paper, we replace the single-writer multiple-reader (SWMR) registers of [CIL94] with multi-writer multi-reader (MWMR) registers, thereby reducing the expected total work to $O(N \log(\log N))$.

Since our adversaries are value-sensitive, every non-faulty process must perform at least one *read* operation, otherwise we can easily construct an execu-

[1] This is faster than other value-oblivious algorithms because SWSR is a weak primitive. More discussion can be found in Section 7.

[2] The assumption in [CIL94] says that the adversary cannot distinguish between the values r and $r + 1$ as they are written to the same memory location.

tion that violates the agreement property. Therefore, expected total work in this model is $\Omega(N)$, which is almost matched by our upper bound of $O(N \log(\log N))$.

We have adopted the worst case expected total work as our complexity measure, mainly because it is more natural to reason about the collective effect of all processes on the shared memory. In fact, per process work in our case is comparable to total work: if all but one process suffer crash failures, the lone survivor carries the total work burden and performs expected $\Omega(N)$ tosses in order to pull far enough ahead for termination. In this sense, our algorithm is less efficient than [Cha96,Aum97], where polylogarithmic upper bounds are given for per process work.

Probabilistic Model Checking. We model check instances of our algorithm using PRISM, which can check PCTL (*Probabilistic Computation Tree Logic*) formulas against an MDP (*Markov Decision Process*) [PRI,BK98]. This tool has been applied to many randomized algorithms, including the consensus algorithm of Aspnes and Herlihy [AH90,KNS01] and Byzantine agreement [KN02].

Consensus algorithms are often hard to model check, because the state space grows exponentially with the number of participating processes. In [KNS01], PRISM is applied only to a shared-coin subroutine, while full correctness relies on verification using Cadence SMV, as well as higher level manual proofs. Unfortunately, the structure of our algorithm does not provide such convenient isolation of probabilistic reasoning. Nevertheless, we are able to build models of binary consensus with up to 4 processes and verify relevant properties. In Section 6, we briefly describe these models and give a summary of PRISM results. In Section 7, we discuss some prospects in improving feasibility of model checking.

Overview. Section 2 describes in greater detail our computational setting and assumptions. Section 3 presents the algorithm and Sections 4 and 5 outline correctness proofs. Detailed proofs are carried out in [Che05b]. Section 6 is devoted to model checking and Section 7 contains closing discussions.

2 System Model

We consider a system of N processes interacting asynchronously via shared memory objects. Each process P_i is given as input an initial preference p_i^0, which belongs to a fixed, finite domain. Without loss of generality, this preference domain is assumed to be $\mathbb{Z}_K$ for some natural number constant $K \geq 2$. As a convention, we write $\mathbb{Z}_K$ for $\{0, \ldots, K-1\}$ and $\mathbb{Z}_K^+$ for $\{1, \ldots, K-1\}$.

We take a state-based view of our system. The *local state* of a process is determined by a valuation of all of its local variables, plus a program counter indicating the next line of code to be executed. The *global state* is then determined by local states of all N processes, together with contents of shared MWMR atomic registers.

A process executes a possibly infinite sequence of discrete steps, each consisting of a change in local state and/or a memory operation. It may also exhibit

a limited form of non-deterministic behavior: crashing at any point of its execution. A crashed process may never recover and re-enter the algorithm.

An execution of the entire system is obtained by interleaving executions of individual processes, where scheduling among processes is determined by an adversary that satisfies the atomicity assumption stated in Section 1. That is, if a process is scheduled to toss a coin, it must be allowed to write to the memory before another process is given a turn. The worst-case complexity is measured in terms of the expected number of *read* and *write* operations taken by all processes, quantifying over all admissible adversaries.

3 Modified CIL Algorithm

As in many other consensus algorithms (e.g. [BO83,CIL94,AH90,Cha96]), we make use of a *round* structure. During each round, a process goes through a possibly infinite sequence of *phases*, each of which is a complete pass through the main **while**-loop.

In original CIL, the shared memory is configured into an array of N many SWMR registers, one for every process. Each $\mathsf{register}_i$ contains two pieces of information: round number r_i and preference value p_i. At the beginning of each phase, process P_i copies the contents of all $\mathsf{register}_j$ $(i \neq j)$ and stores them locally. These entries are then examined to decide the next action of P_i: output a decision value and terminate, toss a coin to advance to the next round, or jump to a higher round.

The initial copying of each phase is the main source of inefficiency in original CIL: copied data contain more information than necessary for decision making. For example, P_i need not know exactly which P_j is in a higher round, as long as it knows *some* P_j is. This observation is precisely the motivation of our move from SWMR memory to MWMR memory. Thus, instead of a *race among processes*, we envision a *race among preference values*. In this way, processes participate anonymously and the number of *read* operations in the main loop is reduced from $O(N)$ to $O(1)$. Moreover, consensus is achieved with high probability using only $O(\log N)$ registers containing one bit each.

Following [Cha96], our MWMR shared memory is configured into K arrays of bits, each of length $R + 2$, where $R := 2\lceil \log N \rceil$. In other words, we have $\mathsf{mem} : \mathbb{Z}_{R+2} \times \mathbb{Z}_K \to \{0, 1\}$. (Recall that K is the size of the preference domain and is a constant, while N is the number of participating processes.) These bits can be interpreted as follows.

- For all $r \in Z^+_{R+1}$ and $v \in \mathbb{Z}_K$, $\mathsf{mem}(r, v) = 1$ if and only if value v has reached round r (i.e., some process holds/held preference v while in round r). These entries are initialized to 0.
- We assume every value v participates in the race from round 0, therefore $\mathsf{mem}(0, v)$ is initialized to 1. This prevents a process from deciding (erroneously) in round 1 before all processes "wake up" and join the protocol[3].

[3] As noted in [CH05], original CIL contains this initialization error.

– Round-$(R+1)$ entries are initialized to 0 and are used for marking decision values. That is, if a process decides on value v, it writes 1 to $\mathsf{mem}(R+1, v)$.

Each process P_i maintains a current preference p_i and a round number r_i. Intuitively, P_i "believes" that p_i is a leading value and r_i is the highest round reached by p_i. If P_i detects any value v in a round higher than r_i, it updates its "belief" by running a subroutine Jump. In this way, lagging values are quickly abandoned by active processes and are eventually *eliminated* from the race. (This notion is made precise in Definition 1 in Section 4.) Therefore the number of contending preference values never increases and the algorithm terminates when that number decreases to 1. If P_i sees p_i at least two rounds ahead in the race, the algorithm guarantees that every other contending value has been eliminated, therefore P_i can safely terminate with p_i.

Notice, biased coin tosses are used to break ties in the lead pack, so that with probability 1 the number of contending preferences eventually reaches 1. This technique is used in [CIL94] and is quite different from the more common approach of shared coin subroutines, in which processes cast randomly generated votes to obtain a weak shared-coin (e.g. [AH90,BR91]).

Although every non-faulty process is guaranteed (with probability 1) to terminate after a finite number of steps, the round in which it terminates can become arbitrarily high. This requires an unbounded number of registers and is infeasible. Therefore we stop our algorithm when it reaches a certain round without successful termination, in which case we switch to a slower algorithm that requires bounded memory. We call this the *exit* algorithm. For convenience, the original CIL algorithm is chosen for this purpose[4]. We will show that any exit algorithm is invoked with probability at most $\frac{1}{N}$, therefore the higher cost of original CIL does not affect overall expected complexity.

Figure 1(a) contains the pseudocode for process P_i. The numbered lines can be described informally as follows.

(1) Check if some process has decided.
(2) If so, decide for the same value.
(3) Check if a value other than p_i has reached round $r_i - 1$.
(4) If not, write 1 to $\mathsf{mem}(R+1, p_i)$ and terminate with output p_i.
(5) Otherwise, if round R is reached, run the original CIL algorithm.
(6) Otherwise, check if some value has reached round $r_i + 1$.
(7) If not, advance p_i to the next round with probability $\frac{1}{2N}$.
(8) Otherwise, run subroutine Jump to find a leading value.

Notice that the atomicity assumption discussed in Section 1 applies at Line (7). This prevents the adversary from selectively delaying *write* operations of processes who are ready to advance its preference to the next round.

Figures 1(b) and 1(c) contain the subroutines ReadMem and Jump, respectively. The former is used to read from the shared memory, while the later is

[4] Technically, original CIL requires registers with unbounded size. However, according to [CIL94], the probability of non-termination is already extremely small (2^{-56}) when each register is 128 bits.

```
ModifiedCIL(i, p_i^0)
local variables
  // round number
  r_i ∈ Z_{R+2},
  // preference
  p_i ∈ Z_K,
  // decision value
  d_i ∈ Z_{K+1},
  // values read from memory
  ahead_i, behind_i ∈ Z_{K+1}
begin
  p_i := p_i^0;  r_i := 0;
  while r_i ≤ R do
(1)  d_i := ReadMem(R + 1, K);
(2)  if d_i ≠ K then return d_i;
     if r_i > 0 then {
(3)    behind_i := ReadMem(r_i − 1, p_i);
(4)    if behind_i = K then {
         mem(R + 1, p_i) := 1;
         return p_i
         }
(5)    elseif r_i = R then return
         OriginalCIL(i, p_i)
       }
(6)  ahead_i := ReadMem(r_i + 1, K);
     if ahead_i = K then {
(7)    with probability 1/(2N) do
         r_i := r_i + 1;
       mem(r_i, p_i) := 1
       }
(8)  else ⟨r_i, p_i⟩ := Jump(r_i + 1, ahead_i)
  od
end
```

(a) Main Algorithm.

```
ReadMem(r, p)
local variables
  // counter
  k ∈ Z_K,
  // preference value found
  v ∈ Z_{K+1},
begin
  k := 0;  v := K;
  while k < K and v = K do
    if mem(r, k) = 1 and k ≠ p then
      v := k;
    k := k + 1
  od
  return v
end
```

(b) Subroutine ReadMem.

```
Jump(r, p)
local variables
  // confirmed round and preference
  r′ ∈ Z_{R+1}, p′ ∈ Z_K,
  // current round and preference
  l ∈ Z^+_{R+1}, u ∈ Z_{K+1},
  // counter
  c ∈ Z_{R+1},
begin
  if r ≥ R then return ⟨r, p⟩;
  r′ := r;  p′ := p;  c := ⌈log(R − r)⌉;
  while c > 0 do
    l := r′ + 2^{c−1};
    if l ≤ R then {
      u := ReadMem(l, K);
      if u ≠ K then {
        r′ := l;  p′ := u
        }
      }
    c := c − 1
  od
  return ⟨r′, p′⟩
end
```

(c) Subroutine Jump.

Fig. 1. Modified CIL Algorithm

used to find a faster value. When called with parameters r and p, ReadMem scans one-by-one the r-th entry of every bit vector, except for the p-th. In other words, we would like to know if any process has reached round r with preference other than p. It returns the first k such that both $k \neq p$ and, at the time of *read* access, $\mathsf{mem}(r, k) = 1$. If no such k is encountered, ReadMem returns K.

In every pass through the **while**-loop of Figure 1(a), ReadMem is called with at most three round numbers: $R+1$, $r_i - 1$, and $r_i + 1$. This does not reveal the highest round ever reached by any value. Therefore, a separate subroutine Jump is run when the process sees itself behind. This is a key difference between our algorithm and original CIL: in exchange for fewer *read* operations in the main loop, more work is needed for slower processes to catch up.

The subroutine Jump can be implemented in various ways. The version presented here is essentially a binary search on mem. This involves $O(\log(\log N))$ operations per invocation of Jump, but a process can correctly locate a fastest value in one complete phase (provided no further progress is made in the mean time).

4 Validity and Agreement

In this section, we treat all coin tosses as non-deterministic choices. Let s_0 denote the initial state of our system, where all N processes as well as the shared memory have been properly initialized. A *path* of the system is a finite sequence of states $s_0 s_1 \dots s_m$ where, for all $j \in \mathbb{Z}_m$, s_{j+1} can be obtained from s_j by allowing exactly one non-faulty process to execute its next instruction. A state s is *reachable* if there is a path ending in s. Finally, a value $k \in \mathbb{Z}_K$ is said to be *valid* if there is $i \in \mathbb{Z}_N$ such that k equals the input p_i^0 to process P_i.

We use record notation to indicate valuation of variables. For example, $s.r_i$ denotes the round number of P_i in state s. If P_i is running a subroutine (e.g. ReadMem), we add subscript i to variables of that subroutine (e.g. $s.k_i$ and $s.v_i$).

First we state some properties about mem and subroutines ReadMem and Jump. Lemma 1 says that an entry in mem never changes from 1 to 0. Lemma 2 says that the return value of ReadMem is correct (although it may be out-of-date). Similarly, Lemma 3 states the correctness of Jump.

Lemma 1. *Let $r \in \mathbb{Z}_{R+2}$, $v \in \mathbb{Z}_K$ and a path $s_0 \dots s_m$ be given. Suppose there is $j \in \mathbb{Z}_{m+1}$ with $s_j.\mathsf{mem}(r, v) = 1$. Then $s_{j'}.\mathsf{mem}(r, v) = 1$ for all $j \leq j' \leq m$.*

Lemma 2. *Let $r \in \mathbb{Z}_{R+2}$, $p, v \in \mathbb{Z}_{K+1}$ and a path $s_0 \dots s_m$ be given. If the last step is $\mathsf{ReadMem}(r, p)$ returning $v \neq K$, then $s_m.\mathsf{mem}(r, v) = 1$.*

Lemma 3. *Let $r, r'' \in \mathbb{Z}_{R+1}$, $p, p'' \in \mathbb{Z}_K$ and a path $s_0 \dots s_m$ be given. Suppose the last step is $\mathsf{Jump}(r, p)$ returning $\langle r'', p'' \rangle$. If $\mathsf{mem}(r, p) = 1$ when $\mathsf{Jump}(r, p)$ is called, then $s_m.\mathsf{mem}(r'', p'') = 1$.*

Proof (Sketch). This follows from the fact that $\mathsf{mem}(r', p') = 1$ is an invariant of the **while**-loop in Jump. □

Lemma 4 below states that mem correctly reflects the preference history of participating processes. Validity is then proven to be an invariant (Theorem 1).

Lemma 4. *Let a path $s_0 \dots s_m$ be given.*

(i) For all $i \in \mathbb{Z}_N$, $s_m.r_i \leq R$ implies $s_m.\mathsf{mem}(s_m.r_i, s_m.p_i) = 1$.
(ii) For all $r \in \mathbb{Z}^+_{R+2}$ and $v \in \mathbb{Z}_K$, $s_m.\mathsf{mem}(r, v) = 1$ implies there exist $i \in \mathbb{Z}_N$ and $j \in \mathbb{Z}_{m+1}$ such that $s_j.p_i = v$.

Theorem 1. *The following claims hold in every reachable state s.*

(i) For every $i \in \mathbb{Z}_N$, $s.p_i$ is valid.
(ii) For every $r \in \mathbb{Z}^+_{R+2}$ and $v \in \mathbb{Z}_K$, $s.\mathsf{mem}(r, v) = 1$ implies v is valid.
(iii) For every $i \in \mathbb{Z}_N$, if $s.d_i \neq K$ then $s.d_i$ is valid. Similarly for $s.\mathsf{ahead}_i$ and $s.\mathsf{behind}_i$.

Corollary 1. *The modified CIL algorithm in Figure 1 is valid, assuming the exit algorithm (in this case, the original CIL algorithm) is valid.*

Next we prove agreement. A key ingredient is a predicate Φ on global states.

Definition 1. *Let $v, v' \in \mathbb{Z}_K$ and $r \in \mathbb{Z}^+_{R+1}$ be given. We say that v* eliminates *v' in round r in global state s (denoted $s \models \Phi(v, v', r)$) just in case $s.\mathsf{mem}(r, v) = 1$ and $s.\mathsf{mem}(r-1, v') = 0$.*

We state a string of lemmas leading to the claim that no two processes terminating by Line (4) do so with conflicting decision values (Lemma 8). First, if an entry $\mathsf{mem}(r, v)$ is marked 1, then every entry $\mathsf{mem}(r', v)$ with $r' \leq r$ is also marked 1 (Lemma 5). Second, if v' is eliminated by v in round r, then no process subsequently reaches round r with preference v' (Lemma 6). Finally, if a process P_i terminates by Line (4) with value v in round r, then every other v' must have been eliminated by v in round r at some earlier state (Lemma 7).

Lemma 5. *Let s be a reachable state. For all $r \in \mathbb{Z}_{R+1}$ and $v \in \mathbb{Z}_K$, if $s.\mathsf{mem}(r, v) = 1$ then $s.\mathsf{mem}(r', v) = 1$ for all $r' \leq r$.*

Lemma 6. *Let $v, v' \in \mathbb{Z}_K$ and $r \in \mathbb{Z}^+_{R+1}$ be given. Consider a path $s_0 \dots s_m$ such that $s_j \models \Phi(v, v', r)$ for some $j \in \mathbb{Z}_{m+1}$. Then, for all $j' \in \{j, \dots, m\}$, $s_{j'}.\mathsf{mem}(r, v') = 0$.*

Proof (Sketch). If the claim doesn't hold, then some process P_i must have written 1 to $\mathsf{mem}(r, v')$ by executing Line (7) between s_j and s_m. This leads to a contradiction because the definition of Φ implies that P_i does not reach Line (7).

Lemma 7. *Consider a path $s_0 \dots s_{m+1}$. Suppose that in the last step some process P_i terminates by executing Line (4). Let r denote $s_m.r_i$ and v denote $s_m.p_i$. For every $v' \neq v$, there is $j' \in \mathbb{Z}_{m+1}$ such that $s_{j'} \models \Phi(v, v', r)$.*

Proof (Sketch). Set $s_{j'}$ to be the state from which the last invocation of ReadMem in Line (3) reads from $\mathsf{mem}(r-1, v')$. □

Lemma 8. *Let a path $s_0 \dots s_m$ and $j, j' \in \mathbb{Z}_{m+1}$ be given. Assume that process P_i terminates by Line (4) with output v from state s_j and some other process $P_{i'}$ does the same with output v' from state $s_{j'}$. Then $v = v'$.*

Proof (Sketch). From the assumptions we prove that v and v' have eliminated each other, which by Lemma 6 is a contradiction. □

It remains to consider termination by Line (2). Lemma 9 below implies that every process terminating by Line (2) must be preceded by a process terminating by Line (4) with the same decision.

Lemma 9. *Let $v \in \mathbb{Z}_K$ and a path $s_0 \dots s_m$ be given. Assume that $s_m.\mathsf{mem}(R+1, v) = 1$. There is $j \in \mathbb{Z}_{m+1}$ such that some process P_i terminates with decision value v by executing Line (4) from s_j.*

Theorem 2. *Let a path $s_0 \dots s_m$ be given. Assume that process P_i terminates by executing either Line (2) or Line (4) from state s_j ($j \in \mathbb{Z}_{m+1}$) and its decision value is v. Similarly for $P_{i'}$, $s_{j'}$ and v'. Then $v = v'$.*

Proof (Sketch). Applying Lemma 9, we find a process that has terminated by Line (4) with v. Similarly for v'. The claim is then reduced to Lemma 8. □

5 Probabilistic Termination and Expected Complexity

Let us first consider the amount of work required during each phase of the algorithm. (Recall that a phase is an entire pass through the **while**-loop in Figure 1(a)). Notice each phase involves at most (i) three calls to ReadMem, (ii) one *write* operation and (iii) one call to Jump. Each call to ReadMem requires $O(1)$ *read* operations, because the size K of the preference domain is a constant in our analysis. Therefore, aside from Jump, each phase involves constant work.

Consider the **while**-loop in Jump. Each pass through this loop involves at most one call to ReadMem. Furthermore, this loop is executed at most $\log R + 1$ times. Since $R = 2\lceil \log N \rceil$ by definition, each call to Jump requires $O(\log(\log N))$ *read* operations. This is then also the cost of a complete phase. Later on, we will prove that the expected number of complete phases until at least one process terminates successfully is $O(N)$ and hence the expected number of *read*/*write* operations is $O(N \log(\log N))$ (Lemma 13).

For any state s, let $s.r_{\max}$ denote the highest round reached by any process in state s. In other words, $s.r_{\max} := \max_{i \in \mathbb{Z}_N} s.r_i$. Since the two updates in Line (7) of Figure 1(a) are performed in a single step, $s.r_{\max}$ is also the largest r such that $s.\mathsf{mem}(r, v) = 1$ for some value $v \in \{0, \dots, K-1\}$. Lemma 10 below says, if no value advances to round $r_{\max} + 1$, a lagging process can catch up to round $r_{\max}$ in one complete phase. Lemma 11 then shows, whenever $r_{\max}$ is at most $R-2$, the probability of at least one process terminating successfully within the next two rounds is bounded below by a constant. Moreover, this termination takes place before $15N$ complete phases are executed.

Lemma 10. *Let $s_0 \dots s_m \dots s_{m'}$ be a path with $m < m'$. Assume that $s_j.r_{\max} = s_m.r_{\max}$ for every $j \in \{m, \dots, m'\}$. Moreover, assume that P_i completes a phase between s_m and $s_{m'}$ without crashing, successfully terminating or switching to the exit algorithm. Then $s_{m'}.r_i = s_m.r_{\max}$.*

Proof (Sketch). First we argue that P_i reaches Line (8) in its first complete phase after s_m. Then, based on the **while**-loop in Jump, we construct a nested sequence of intervals shrinking to the singleton $\{s_m.r_{\max}\}$. Therefore $s_m.r_{\max}$ is the round number returned by Jump. □

Lemma 11. *Suppose* ModifiedCIL *starts from a reachable state s. Let r denote $s.r_{\max}$ and suppose $r \leq R-2$. Then, with probability greater than 0.511, at least one process terminates successfully in a round no higher than $r+2$. Moreover, at most $15N$ complete phases are executed between s and the successful termination.*

Proof (Sketch). Consider two events: E_1 is "a success occurs before $5N$ attempts to move from r to $r+1$ are made and all subsequent such attempts fail" and E_2 is "a success occurs before $5N$ attempts to move from $r+1$ to $r+2$ are made." We argue that the conjunction of E_1 and E_2 implies at least one process terminates successfully in round $r+2$ before $15N$ complete phases are executed. Moreover, the probability of both E_1 and E_2 occurring is at least 0.511, using the fact that $\{(1-\frac{1}{n})^n\}_{n=2}^{\infty}$ increases to the limit $\frac{1}{e}$. □

Notice Lemma 11 applies only to executions starting in round $R-2$ or lower. The next lemma covers rounds $R-1$ and R, assuming a decision is reached without switching to the exit algorithm.

Lemma 12. *Suppose* ModifiedCIL *starts from a reachable state s. Let r denote $s.r_{\max}$ and suppose $R-2 < r \leq R$. Assuming the exit algorithm is not invoked, the (conditional) probability that at least one process terminates successfully before $15N$ complete phases are executed after s is greater than 0.511.*

Theorem 3. *If the exit algorithm is wait-free and satisfies probabilistic termination, the same holds for* ModifiedCIL.

Proof. By correctness of the exit algorithm, we may focus on the case in which the exit algorithm is not invoked. Consider execution blocks of $15N$ complete phases each. By Lemma 11 and Lemma 12, the probability of successful termination within each block is at least 0.511. Thus, with probability 1, the algorithm terminates successfully after a finite number of blocks. Since we have made no assumption on the number of surviving processes, the algorithm is wait-free. □

We now turn to complexity considerations. Again, we make a case distinction based on whether the exit algorithm is invoked.

Lemma 13. *Assume that the exit algorithm is not invoked. The expected number of elementary read/write operations until at least one process terminates successfully is $O(N \log(\log N))$.*

Proof (Sketch). Again we consider blocks of $15N$ complete phases and argue that the expected number of blocks is at most 2. Hence the expected number of complete phases is $O(N)$. Since each phase involves $O(\log(\log N))$ elementary operations, the expected number of elementary operations $O(N \log(\log N))$. □

Lemma 14. *Suppose the exit algorithm is the original CIL algorithm and is invoked. The expected number of elementary read/write operations until at least one process terminates successfully is* $O(N^2 \log(\log N))$.

Proof (Sketch). The expected number of elementary operations before switching is shown to be $O(N(\log N)(\log(\log N)))$. Using results of [CIL94], the expected complexity after switching is $O(N^2)$. Therefore, the overall expected complexity is $O(N^2 \log(\log N))$. □

Lemma 15. *Suppose the* ModifiedCIL *starts from the initial state* s_0. *The probability of failing to reach a decision in or before round* R *is at most* $1/N$.

Proof (Sketch). By Lemma 11, this probability is at most $(1-0.511)^{\frac{R}{2}} \leq \frac{1}{N}$. □

Putting together Lemmas 13, 14, and 15, we conclude that the expected complexity of ModifiedCIL is $O(N \log(\log N))$.

Theorem 4. *Suppose* ModifiedCIL *starts from the initial state* s_0 *and the exit algorithm is original CIL. The expected number of elementary read/write operations until at least one process terminates successfully is* $O(N \log(\log N))$.

6 Model Checking

It is quite straightforward to specify our algorithm in PRISM's state-based input language. Each process is modeled as a *module* and the shared memory is modeled using global variables. Two more global variables are used to keep track of process failures and the number of completed phases.

We consider binary consensus (i.e., $K = 2$) with $N = 2, 3, 4$ processes. Processes are assumed to disagree initially, therefore validity is trivial. Agreement is satisfied in all models constructed. For probabilistic termination, we ask PRISM to compute the (exact) minimum probability of at least one process terminating successfully, given an allowance of $R = 2\lceil \log N \rceil$ rounds and $15N \cdot \frac{R}{2} = 15N\lceil \log N \rceil$ complete phases. This result is compared against our analytic lower bound of $1 - \frac{1}{N}$.

In the case of $N = 4$, the model becomes too complex (with $2\lceil \log N \rceil = 4$ rounds and $15N\lceil \log N \rceil = 120$ complete phases). However, we discover that the analytic bound of $1 - \frac{1}{N} = 0.750$ is already met when we restrict to 40 complete phases. This suggests that we have made some overly conservative estimates while deriving the analytic bound.

The table below summarizes our results. We use PRISM version 2.1, running on a 1.4 GHz Pentium M machine with 500 Mb memory under Linux 2.6. The MTBDD engine is used with a CUDD memory limit of 400 Mb. Other parameters

remain at default settings. All relevant files, including model checking logs, can be found in [Che05a].

N	R	#Phases	Model		Agreement	Termination		
			#States	Time(s)	Time(s)	Time(s)	MinProb	AnalyticBd
2	2	30	42,320	4	0.025	6	0.745	0.511
3	4	90	12,280,910	213	0.094	2,662	0.971	0.667
4	2	60	45,321,126	429	0.078	602	0.755	0.511
4	4	40	377,616,715	5224	3.926	55,795	0.765	0.750

7 Conclusions

We have given a simple algorithm that solves asynchronous wait-free consensus in expected $O(N \log(\log N))$ total work. We follow a value-based (as opposed to process-based) approach and make use of MWMR atomic registers. This strategy, also adopted in [Cha96,Aum97], leads to a significant reduction in data handling and hence more efficient consensus algorithms. As a pleasant side-effect, the reduction in both global and local data makes model checking significantly more feasible, for it helps to avoid the typical state explosion problem.

MWMR memory is often regarded as a stronger primitive than SWMR memory. Indeed, there are optimal implementations of MWMR from physical SWMR registers using linear time and logarithmic space [IS92]. However, if one makes comparisons from the basis of SWSR, then MWMR and SWMR become roughly the same: when implemented from SWSR, both require linear time and logarithmic space. Moreover, it is argued in [BPSV00] that SWMR memory requires the hidden assumption of *naming*: existence of distinct identifiers known to all. In that sense, MWMR is a weaker primitive compared to SWMR. This idea is echoed by the fact that, unlike the original CIL algorithm, our version allows processes to participate anonymously.

The MWMR strategy has another advantage, namely, flexibility in memory usage. We have shown that, with high probability, consensus can be reached using $O(\log N)$ many single-bit MWMR registers. (That is, the main algorithm succeeds and thus the exit algorithm is not invoked.) This can be seen as a temporary reprieve from the lower bound of $\Omega(\sqrt{N})$ for the space requirement of randomized consensus [FHS98]. In practice, one may be willing to accept a small probability of failing to reach consensus, in which case we can remove the exit algorithm altogether. The main algorithm can be repeated to increase the success probability, and memory is allocated only as needed.

For future work, we want to improve the per process work bound of our algorithm. In [AW96], a similar improvement is achieved by allowing fast processes to cast votes of increasing weights. However, their proofs rely on properties of Martingale processes and cannot be adapted immediately to our setting. At this time, we do not know if per process work is inherently high in our setting (e.g. $\Omega(\frac{N}{f(N)})$, where f is a polylogarithmic function).

Finally, we comment on model checking using PRISM. Although the current limit seems to be 4 processes, we conjecture a vast improvement using a symme-

try reduction option, which is under development by the PRISM team. Before symmetry reduction is available, manual abstraction can be used to increase feasibility. That is, we manually construct an abstraction that captures core ideas of an algorithm, while significantly decreasing the model size. We experimented with such an abstraction of original CIL, by focusing on the shared memory and filtering out local states of processes. Having done so, we were in fact able to handle up to 10 processes. However, it is non-trivial to prove soundness of the abstraction. Standard techniques such as probabilistic simulation are available for this purpose, but substantial investment of time is required.

Overall, PRISM allows us to conduct experiments during the development stage of an algorithm, with minimal learning effort. Although in most cases it still cannot handle large instances of a full algorithm, it is perfectly feasible to model check a subroutine or an abstract version. This already provides valuable information, especially to those who simply wish to gain more insight into an algorithm.

Acknowledgment. We thank James Aspnes for his inspiring article [Asp03] and many helpful comments, as well as David Parker for support in using PRISM. Also we thank Jaap-Henk Hoepman and the anonymous referees at OPODIS'05 for their useful suggestions.

References

[ACT00] M.K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crach recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990.

[AKL99] Y. Aumann and A. Kapah-Levy. Cooperative sharing and asynchronous consensus using single-read single-writer registers. In *Proceedings of the 10th ACM-SIAM Anuual Symposium on Discrete Algorithms (SODA)*, pages 61–70, 1999.

[Asp98] J. Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, 1998.

[Asp03] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[Aum97] Y. Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, 1997.

[AW96] J. Aspnes and O. Waarts. Randomized consensus in expected $O(n \log^2 n)$ operations per process. *SIAM Journal on Computing*, 25(5):1024–1044, 1996.

[BK98] C. Baier and M. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[BO83] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

[BPSV00] H. Buhrman, A. Panconesi, R. Silvestri, and P.M.B. Vitányi. On the importance of having an identity or is consensus really universal? In *Proceedings of the 14th International Conference on Distributed Computing*, volume 1914 of *LNCS*, pages 134–148. Springer-Verlag, 2000.

[BR91] G. Bracha and O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, volume 579 of *LNCS*, pages 143–150, 1991.

[CH05] L. Cheung and M. Hendriks. Causal dependencies in parallel composition of stochastic processes. Technical Report ICIS-R05020, Institute for Computing and Information Sciences, University of Nijmegen, 2005.

[Cha96] T.D. Chandra. Polylog randomized wait-free consensus. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 166–175, 1996.

[Che05a] L. Cheung. Collection of PRISM models of the modified CIL algorithm, 2005. Available at `http://www.niii.ru.nl/~lcheung/mcil/`.

[Che05b] L. Cheung. Randomized wait-free consensus using an atomicity assumption. Technical Report ICIS-R05035, Institute for Computing and Information Sciences, University of Nijmegen, 2005. Available at `http://www.niii.ru.nl/~lcheung/cilTR.pdf`.

[CIL87] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings PODC'87*, pages 86–97, 1987.

[CIL94] B. Chor, A. Israeli, and M. Li. Wait-free consensus using asynchronous hardware. *SIAM Journal on Computing*, 23(4):701–712, 1994.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[FHS98] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, September 1998.

[FLP85] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[IS92] A. Israeli and A. Shaham. Optimal multi-writer multi-reader atomic register. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 71–82, 1992.

[KN02] M. Kwiatkowska and G. Norman. Verifying randomized Byzantine agreement. In *Proc. Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *LNCS*, pages 194–209, 2002.

[KNS01] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In *Proceedings CAV'01*, volume 2102 of *LNCS*, pages 194–206, 2001.

[PRI] PRISM web site. `http://www.cs.bham.ac.uk/~dxp/prism`.