

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

AMS (2000): 68W15, 68W20, 68W40, 68Q25, 68Q60

CR (1998): C.2.2, C.4, G.3

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]).

* An extended abstract of this report appears in the *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05)*.

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

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.

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. This is done by specifying the so-called *adversaries*, which are fictitious entities designed to model scheduling uncertainties in a distributed environment. Mathematically, an adversary is a function mapping each finite execution to an available next operation. Such a function resolves all non-deterministic choices among parallel processes, thereby inducing a probability distribution on the set of executions. One can then ask if probabilistic termination is satisfied according to this distribution. By quantifying over all possible adversaries (of a certain strength), we obtain worst-case guarantees similar to those in a non-probabilistic setting.

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. The *oblivious* adversaries are the weakest, unable to observe any random outcomes and their subsequent effects on system dynamics.

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

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

heads; then immediately it writes 1 to the memory location $\text{mem}(v, r)$, where v is the current preference and r is the round number *after* the coin toss. This amounts to saying that the adversary cannot distinguish between the two locations $\text{mem}(v, r)$ and $\text{mem}(v, r + 1)$.

The original CIL algorithm relies on a similar atomicity assumption² 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 execution 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 our learning experience with PRISM and 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 correctness proofs are given in Sections 4 and 5. Section 6 is devoted to model checking and Section 7 contains closing discussions.

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.

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 $\{0, \dots, K-1\}$ and \mathbb{Z}_K^+ for $\{1, \dots, K-1\}$.

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. In fact, it is trivial to include the program counter as an explicit variable, so that local state is fully determined by valuation of local variables. This is done in our PRISM models. The *global state* is then determined by local states of all N processes, together with contents of shared MWMR atomic registers. These registers are by definition *linearizable* [HW90], meaning that each memory operation can be viewed as taking place at a particular instant in time, as opposed to an interval between *invocation* and *response*. Under this assumption, each *read* access returns the value written by the last *write* access in the linearized execution history.

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 **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 **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 $\text{mem} : \mathbb{Z}_{R+2} \times \mathbb{Z}_K \rightarrow \{0, 1\}$. (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 $r \in \mathbb{Z}_{R+1}^+$ and $v \in \mathbb{Z}_K$, $\text{mem}(r, v) = 1$ 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 $\text{mem}(0, v)$ is initialized to 1. This prevents a process from deciding (erroneously) in round 1 before all processes “wake up” and join the protocol³.
- 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 $\text{mem}(R + 1, v)$.

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,

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

the original CIL algorithm is chosen for this purpose⁴. 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 $\text{mem}(R + 1, p_i)$ 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 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, $\text{mem}(r, k) = 1$. 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. A simple solution is to increment r_i by 1 and then call **ReadMem** once to find the least k such that $\text{mem}(r_i, k) = 1$. This requires a constant amount of work per invocation of **Jump** and is implemented in our PRISM models. However, a process lagging way behind may need to step through the main loop as many as R times in order to catch up. Hence we opt for a faster implementation, which 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 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
   $r_i \in \mathbb{Z}_{R+2}$ ,
  // preference
   $p_i \in \mathbb{Z}_K$ ,
  // decision value
   $d_i \in \mathbb{Z}_{K+1}$ ,
  // values read from memory
   $\text{ahead}_i, \text{behind}_i \in \mathbb{Z}_{K+1}$ 
begin
   $p_i := p_i^0$ ;  $r_i := 0$ ;
  while  $r_i \leq R$  do
    (1)  $d_i := \text{ReadMem}(R+1, K)$ ;
    (2) if  $d_i \neq K$  then return  $d_i$ ;
        if  $r_i > 0$  then {
    (3)  $\text{behind}_i := \text{ReadMem}(r_i - 1, p_i)$ ;
    (4) if  $\text{behind}_i = K$  then {
           $\text{mem}(R+1, p_i) := 1$ ;
          return  $p_i$ 
        }
    (5) elseif  $r_i = R$  then return
         $\text{OriginalCIL}(i, p_i)$ 
      }
    (6)  $\text{ahead}_i := \text{ReadMem}(r_i + 1, K)$ ;
        if  $\text{ahead}_i = K$  then {
    (7) with probability  $\frac{1}{2^N}$  do
           $r_i := r_i + 1$ ;
           $\text{mem}(r_i, p_i) := 1$ 
        }
    (8) else  $\langle r_i, p_i \rangle := \text{Jump}(r_i + 1, \text{ahead}_i)$ 
      od
    end
  end

```

(a) Main Algorithm.

```

ReadMem( $r, p$ )
local variables
  // counter
   $k \in \mathbb{Z}_K$ ,
  // preference value found
   $v \in \mathbb{Z}_{K+1}$ ,
begin
   $k := 0$ ;  $v := K$ ;
  while  $k < K$  and  $v = K$  do
    if  $\text{mem}(r, k) = 1$  and  $k \neq p$  then
       $v := k$ ;
       $k := k + 1$ 
    od
  return  $v$ 
end

```

(b) Subroutine ReadMem.

```

Jump( $r, p$ )
local variables
  // confirmed round and preference
   $r' \in \mathbb{Z}_{R+1}$ ,  $p' \in \mathbb{Z}_K$ ,
  // current round and preference
   $l \in \mathbb{Z}_{R+1}^+$ ,  $u \in \mathbb{Z}_{K+1}$ ,
  // counter
   $c \in \mathbb{Z}_{R+1}$ ,
begin
  if  $r \geq R$  then return  $\langle r, p \rangle$ ;
   $r' := r$ ;  $p' := p$ ;  $c := \lceil \log(R - r) \rceil$ ;
  while  $c > 0$  do
     $l := r' + 2^{c-1}$ ;
    if  $l \leq R$  then {
       $u := \text{ReadMem}(l, K)$ ;
      if  $u \neq K$  then {
         $r' := l$ ;  $p' := u$ 
      }
    }
     $c := c - 1$ 
  od
  return  $\langle r', p' \rangle$ 
end

```

(c) Subroutine Jump.

Fig. 1. Modified CIL Algorithm

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 $s.v_i$). This should not cause any confusion because each process runs at most one instance of any subroutine at any given point of the execution.

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 $s_j.\text{mem}(r, v) = 1$. Then $s_{j'}.\text{mem}(r, v) = 1$ for all $j \leq j' \leq m$.*

Proof. A process writes to the shared memory only if it executes Lines (4) or (7) in Figure 1(a). In either case, the value 1 is written. Therefore, once `mem`(r, v) becomes 1, it retains that value in the rest of the path. \square

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 `ReadMem`(r, p) returning $v \neq K$, then $s_m.\text{mem}(r, v) = 1$.*

Proof. The return value v of `ReadMem` is set to a value other than K only if the **if-then** clause is executed. Let s_j ($0 \leq j \leq m$) be the state from which this instance of `ReadMem` reads from `mem`(r, v). Clearly, $s_j.\text{mem}(r, v) = 1$. By Lemma 1, this also holds in s_m . \square

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 `Jump`(r, p) returning $\langle r'', p'' \rangle$. If `mem`(r, p) = 1 when `Jump`(r, p) is called, then $s_m.\text{mem}(r'', p'') = 1$.*

Proof. We prove that `mem`(r', p') = 1 is an invariant of the **while**-loop in `Jump`. By assumption, the claim holds for initial values $r' = r$ and $p' = p$. Noticed that $\langle r', p' \rangle$ is updated only if the **if-then** clause is executed, in which case $v \neq K$. Since v is the return value of `ReadMem`(l, K), we have by Lemma 2 that `mem`(l, v) = 1, hence `mem`(r', p') = 1 still holds after the update. Let s_j be the state immediately after the last update of $\langle r', p' \rangle$. Then we know $s_j.\text{mem}(r'', p'') = 1$. By Lemma 1, this also holds in s_m . \square

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

Proof. We proceed by induction on the length of paths. For the initial state s_0 , recall that round-0 entries are initialized to 1 and all other entries 0, therefore the two claims hold trivially.

Now we consider a path $s_0 \dots s_m s_{m+1}$. Suppose the last step is taken by process P_i . Let r denote $s_m.r_i$ and v denote $s_m.p_i$. Notice that only Lines (4), (7) and (8) in Figure 1(a) update variables r_i , p_i and mem .

- Line (4). By Lemma 1, Item (i) is trivial because r_i is not updated. Item (ii) holds because the only entry of interest is $\text{mem}(R+1, s_m.p_i)$.
- Line (7). If the coin toss fails, there is no state change other than the program counter of P_i . Suppose the coin toss succeeds. For Item (i), simply note that

$$s_{m+1}.\text{mem}(s_{m+1}.r_i, s_{m+1}.p_i) = s_{m+1}.\text{mem}(r+1, v) = 1$$

because the two updates in Line (7) are assumed to be simultaneous. On the other hand, Item (ii) also holds because $s_{m+1}.p_i = v$.

- Line (8). Item (i) follows from Lemma 3. Item (ii) follows from the induction hypothesis.

□

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 $v \in \mathbb{Z}_K$, $s.\text{mem}(r, v) = 1$ 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 $s.\text{ahead}_i$ and $s.\text{behind}_i$.*

Proof. We prove these claims simultaneously using induction on the length of paths. First consider the initial state s_0 . For Item (i), every $s_0.p_i$ is valid because it is set to the input value p_i^0 . Item (ii) holds trivially because all but round-0 entries are initialized to 0. Item (iii) is also trivial because all relevant variables are initialized to K .

Now we consider a path $s_0 \dots s_m s_{m+1}$. Suppose the last is taken by process P_i . We examine Figure 1(a) line by line for all possible actions of P_i .

- Line (1). In this case, one update is possible: d_i is set to the return value v of $\text{ReadMem}(R+1, K)$. Suppose v is not K . Then we can apply Lemma 2 to conclude that $s_{m+1}.\text{mem}(R+1, v) = 1$. Since mem is not updated in the last step, this also holds in s_m . Applying the induction hypothesis, we conclude that $s_{m+1}.d_i = v$ is valid.
- Line (2). In this case, P_i terminates by returning the value $s_m.d_i$ and there are no variable updates. We simply apply the induction hypothesis.
- Lines (3) and (6). Similar to Line (2).

- Line (4). In this case, $\text{mem}(R + 1, s_m.p_i)$ is set to 1. By the induction hypothesis, $s_m.p_i$ is valid. Therefore Items (ii) hold in s_{m+1} . Item (iii) follows from the inductive hypothesis.
- Line (7). Similar to Line (4).
- Line (8). $\langle r_i, p_i \rangle$ is set to the return values of `Jump`. Notice that this update is executed only if $s_m.\text{ahead}_i \neq K$. Therefore, we can conclude that in Line (6) `ReadMem` returned a value v other than K . Moreover, notice that from then on r_i and ahead_i have not be updated. Therefore, by Lemmas 1 and 2, we know that $\text{mem}(s_m.r_i + 1, s_m.\text{ahead}_i) = 1$ at the time `Jump` is called. Applying Lemma 3, we have $s_{m+1}.\text{mem}(s_{m+1}.r_i, s_{m+1}.p_i) = 1$. Since mem is not updated in the last step, we have $s_m.\text{mem}(s_{m+1}.r_i, s_{m+1}.p_i) = 1$. Applying the induction hypothesis, we conclude that $s_{m+1}.p_i$ is valid. \square

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 $s.\text{mem}(r, v) = 1$ and $s.\text{mem}(r - 1, v') = 0$.*

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 $\text{mem}(r, v)$ is marked 1, then every entry $\text{mem}(r', v)$ 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 $s.\text{mem}(r, v) = 1$ then $s.\text{mem}(r', v) = 1$ for all $r' \leq r$.*

Proof. We proceed by induction on the length of paths. Clearly this holds at the initial state s_0 . Consider a path of the form $s_0 \dots s_{m+1}$ and assume the property holds for s_m . The only case of interest is when $\text{mem}(r, v)$ changes from 0 to 1 as the result of some process P_i executes Line (7) from s_m . In that case, we have $s_m.r_i = r - 1$ and $s_m.p_i = v$. By Lemma 4, we may infer that $s_m.\text{mem}(r - 1, v) = 1$. By the induction hypothesis, $s_m.\text{mem}(r', v) = 1$ for all $r' \leq r - 1$. Using Lemma 1, we conclude that $s_{m+1}.\text{mem}(r', v) = 1$ for all $r' \leq r$. \square

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 $j' \in \{j, \dots, m\}$, $s_{j'}.\text{mem}(r, v') = 0$.*

Proof. By the definition of Φ , we have $s_j.\text{mem}(r-1, v') = 0$ and $s_j.\text{mem}(r, v) = 1$. Using Lemma 5, we infer that $s_j.\text{mem}(r, v') = 0$. For contradiction, suppose that the claim doesn't hold. We focus on the least $j' \geq j$ with $s_{j'}.\text{mem}(r, v') = 1$. Then it must be the case that $s_{j'-1}.\text{mem}(r, v') = 0$ and some process P_i executes Line (7) from $s_{j'-1}$. Moreover, $s_{j'-1}.r_i = r - 1$ and $s_{j'-1}.p_i = v'$.

On the other hand, using Lemma 4 and the fact that $s_j.\text{mem}(r - 1, v') = 0$, we know either $s_j.r_i < r - 1$ or $s_j.p_i \neq v'$. Therefore, P_i must have entered the current phase *after* s_j . Since $\text{mem}(r, v)$ is 1 in every state following s_j , the invocation of `ReadMem` in Line (6) of the current phase of P_i must have returned a value other than K . This contradicts the claim that P_i executes Line (7) in the current phase. \square

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. Let $v' \neq v$ be given and let s_j denote the first state in which P_i enters the current phase. Thus $s_j.r_i = r$ and $s_j.p_i = v$. By Lemma 4 and Lemma 1, we have $s_{j''}.\text{mem}(r, v) = 1$ for all $j'' \in \{j, \dots, m\}$.

Since Line (4) is executed, r must be non-zero and the invocation of `ReadMem` in Line (3) must have returned K . Let $s_{j'}$ be the state from which `ReadMem` reads from $\text{mem}(r - 1, v')$. Since the return value of `ReadMem` is K , we may infer that $s_{j'}.\text{mem}(r - 1, v') = 0$. Moreover, we have $j' > j$ and hence $s_{j'}.\text{mem}(r, v) = 1$. Therefore $s_{j'} \models \Phi(v, v', r)$. \square

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. For the sake of contradiction, suppose $v \neq v'$. Let r and r' denote the final round numbers of P_i and $P_{i'}$, respectively. Without loss of generality, assume that $r \leq r'$. By Lemma 7, we know that $s_j \models \Phi(v, v', r)$, therefore $s_j.\text{mem}(r, v) = 1$ and $s_j.\text{mem}(r - 1, v') = 0$. On the other hand, $s_{j'}.r_{i'} = r'$ and $s_{j'}.p_{i'} = v'$, so by Lemma 4 we have $s_{j'}.\text{mem}(r', v') = 1$.

First we consider the case in which $j < j'$. By Lemma 6, we know that $s_{j'}.\text{mem}(r, v') = 0$. Applying Lemma 5, we have $s_{j'}.\text{mem}(r', v') = 0$, which yields a contradiction.

Next we consider the case in which $j' < j$. By Lemma 1, we may infer that $s_j.\text{mem}(r', v') = 1$. By Lemma 5, this implies $s_j.\text{mem}(r - 1, v') = 1$, contradicting the fact that $s_j \models \Phi(v, v', r)$. \square

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 $s_m.\text{mem}(R + 1, v) = 1$. 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 .*

Proof. Let j be the index for the first state in this path such that $s_j.\text{mem}(R + 1, v) = 1$. Since $\text{mem}(R + 1, v)$ is initialized to 0, we know that $j > 0$. Let P_i be the first process writing to $\text{mem}(R + 1, v)$ from s_{j-1} . Then P_i must have terminated with decision value v by executing Line (4) from s_{j-1} . \square

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. We claim that there exist $j'' \in \mathbb{Z}_{m+1}$ and $i'' \in \mathbb{Z}_N$ such that $P_{i''}$ terminates with decision value v by executing Line (4) from $s_{j''}$. If P_i terminates by Line (4), then we simply set $i'' := i$ and $j'' := j$. Otherwise, P_i terminates by Line (2) and the invocation of `ReadMem` in Line (1) of the last phase of P_i must have returned $v \neq K$. By Lemma 2 and Lemma 1, we know that $s_m.\text{mem}(R + 1, v) = 1$. We can then choose j'' and i'' using Lemma 9.

The same claim also holds for v' . Now we apply Lemma 8 to infer that $v = v'$. \square

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 $s.\text{mem}(r, v) = 1$ 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. The proof of Lemma 11 resembles the analysis given in [CIL94].

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. For readability, write r_{\max} for $s_m.r_{\max}$ and r for $s_m.r_i + 1$. Consider the first complete phase executed by P_i between s_m and s'_m . Without loss of generality, assume that s_m is the first state in that phase and that $r \leq r_{\max}$.

By assumption, P_i does not crash, terminate, or exit. Therefore it reaches Line (6) in this phase. By Lemma 1 and Lemma 5, $r \leq r_{\max}$ implies there is $v \in \mathbb{Z}_K$ such that $s_j.\text{mem}(r, v) = 1$ for all $j \in \{m, \dots, m'\}$. Hence the invocation of `ReadMem` in Line (6) returns a value other than K and P_i executes Line (8). It remains to show `Jump` returns r_{\max} for the round number.

Note that `Jump` returns r if $r \geq R$, in which case $r = R = r_{\max}$. Otherwise, let c denote $\lceil \log(R - r) \rceil$. The **while**-loop of `Jump` calculates the following sequence $\{r'_0, \dots, r'_c\}$ of natural numbers: r'_0 is r and r'_{i+1} is

- r'_i , if $r'_i + 2^{c-i} > R$ or `ReadMem`($r'_i + 2^{c-i}, K$) returns K ;
- $r'_i + 2^{c-i}$, otherwise.

From this we obtain a nested sequence of intervals:

$$[r'_0, r'_0 + 2^c), \dots, [r'_i, r'_i + 2^{c-i}), \dots, [r'_c, r'_c + 2^0).$$

It is easy to see that r_{\max} belongs to every one of these intervals and, since the last is a singleton, we know $r'_c = r_{\max}$. This is precisely the round number returned by `Jump`. \square

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. By assumption, at least one process survives throughout the execution of the algorithm. Therefore, if no successful termination ever takes place, the algorithm stops only if all surviving processes reach round R and switch to the exit algorithm. Without loss of generality, we assume that no successful termination occurs in round $r + 1$ or lower.

Consider the first complete phase following s . There are two cases.

- It is executed by a process P_i in round $< r$. By Lemma 10, P_i reaches round r by the end of this phase.
- It is executed by a process P_i in round r . Then P_i reaches Line (7) in this phase and, with probability $\frac{1}{2N}$, P_i advances to round $r + 1$.

Suppose that either the first case applies, or the second case applies but P_i fails to advance to round $r + 1$. Then the same case distinction can be made on the next complete phase. This repeats until all surviving processes have reached round r and, after that point, every complete phase involves a coin toss to advance to round $r + 1$ until a success occurs. Moreover, since a lagging process catches up

to round r in one complete phase, at most N complete phases following s are executed by processes in round strictly lower than r .

Consider the event E_1 , in which “a success occurs before $5N$ attempts to move from r to $r + 1$ are made” and “all subsequent attempts to move from r to $r + 1$ fail.” Notice the first condition is equivalent to “it is not the case that all of the first $5N$ attempts to move from r to $r + 1$ fail,” which occurs with probability $1 - (1 - \frac{1}{2N})^{5N}$. By the reasoning above, this first success occurs before $6N$ complete phases are executed following s .

Let P_i be the successful process, thus the first to reach round $r + 1$. By our atomicity assumption, $\text{mem}(r + 1, s.p_i)$ is set to 1 as soon as P_i reaches round $r + 1$. After that point, every other $P_{i'}$ tosses a coin at most once to advance from r to $r + 1$. This is because in the subsequence phase $P_{i'}$ sees it's no longer leading and therefore does not execute Line (7). As a result, the probability of “all subsequent attempts to move from r to $r + 1$ fail” is at least $(1 - \frac{1}{2N})^{N-1}$ and hence the probability of E_1 is at least $(1 - (1 - \frac{1}{2N})^{5N})(1 - \frac{1}{2N})^{N-1}$. Moreover, after P_i reaches round $r + 1$, at most $2N - 2$ complete phases are executed by processes in round r or lower: at most $N - 1$ failed coin tosses to move from r to $r + 1$ and at most $N - 1$ phases to catch up to $r + 1$.

By assumption, no successful termination takes place until a process has reached round $r + 2$. Thus, every complete phase executed by a process in round $r + 1$ is a coin toss to move to round $r + 2$, until a success occurs. Let E_2 denote the event that “a success occurs before $5N$ attempts to move from $r + 1$ to $r + 2$ are made.” The probability of E_2 given E_1 is then $1 - (1 - \frac{1}{2N})^{5N}$. Similarly, this success occurs before $6N + (2N - 2) + 5N = 13N - 2$ complete phases are executed following s and, after this success, at most $2N - 2$ complete phases are executed by processes in round $r + 1$ or lower.

Therefore, given E_1 and E_2 , at least one process executes a complete phase in round $r + 2$ before $15N$ complete phases are executed following s . Due to E_1 , no process reaches round $r + 1$ with preference value other than $s.p_i$. Therefore the first process to complete a phase in round $r + 2$ sees no disagreement in round $r + 1$ or higher. It then terminates successfully by Line (4). It remains to consider the probability of both E_1 and E_2 occurring. Recall that $\{(1 - \frac{1}{n})^n\}_{n=2}^\infty$ is an increasing sequence with limit $\frac{1}{e}$. Therefore $\{(1 - \frac{1}{2n})^{n-1}\}_{n=2}^\infty$ is a decreasing sequence with limit $\frac{1}{\sqrt{e}}$ and $\{1 - (1 - \frac{1}{2n})^{5n}\}_{n=2}^\infty$ is a decreasing sequence with limit $1 - \frac{1}{e^{2.5}}$. Therefore, the probability of both E_1 and E_2 occurring is at least

$$(1 - (1 - \frac{1}{2N})^{5N})^2 \cdot (1 - \frac{1}{2N})^{N-1} \geq (1 - \frac{1}{e^{2.5}})^2 \cdot \frac{1}{\sqrt{e}} > 0.511.$$

□

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.

Proof. We use arguments similar to those in the proof of Lemma 11. First suppose $r = R$. Then at most $N - 1$ complete phases are executed before a process completes a phase in round R . Suppose P_i is the first to do so. If P_i does not terminate by Line (4) in that phase, it must be the case that $\text{mem}(R - 1, v_1) = \text{mem}(R - 1, v_2) = 1$ for some $v_1 \neq v_2$. Then P_i , as well as every other process that reaches round R , invokes the exit algorithm. By assumption, this is not the case and hence P_i terminates by Line (4) in that phase. Therefore, with probability 1, at least one process terminates before N complete phases are executed.

If $r = R - 1$, then at most $N - 1$ complete phases are executed before a process completes a phase in round $R - 1$. Similar to the previous case ($r = R$), if the first process completing a phase in round $R - 1$ does not terminate by Line (4) in that phase, every process reaching round $R - 1$ will try to advance to round R by Line (7), until one of them succeeds. The probability of at least one success before $4N$ attempts are made is $1 - (1 - \frac{1}{2N})^{4N}$, which is bounded below by $(1 - \frac{1}{e^2}) > 0.864$. After that success, the problem reduces to the case where $r = R$ and successful termination is guaranteed before N complete phases. Therefore, with probability at least 0.864, at least one process terminates before $6N$ complete phases are executed. \square

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. \square

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. Again we consider execution blocks of $15N$ complete phases each. The expected number of blocks is:

$$\sum_{n=1}^{\infty} (n \cdot 0.511 \cdot (1 - 0.511)^{n-1}) = \sum_{n=0}^{\infty} ((n + 1) \cdot 0.511 \cdot 0.489^n).$$

Factoring out 0.511 and rearranging the summands, we have

$$\begin{aligned}
& \sum_{n=0}^{\infty} ((n+1) \cdot 0.511 \cdot 0.489^n) \\
&= 0.511 \cdot \left(\sum_{n=0}^{\infty} 0.489^n + \sum_{n=1}^{\infty} 0.489^n + \sum_{n=2}^{\infty} 0.489^n + \dots \right) \\
&= 0.511 \cdot \left(\sum_{n=0}^{\infty} 0.489^n + 0.489 \sum_{n=0}^{\infty} 0.489^n + 0.489^2 \sum_{n=0}^{\infty} 0.489^n + \dots \right) \\
&= 0.511 \cdot \left(\sum_{n=0}^{\infty} 0.489^n \right)^2 \\
&= 0.511 \cdot \left(\frac{1}{0.511} \right)^2 = \frac{1}{0.511} < 2.
\end{aligned}$$

Thus the expected number of complete phases is at most $30N$. Moreover, there are at most $N - 1$ incomplete phases. Since each phase involves $O(\log(\log N))$ elementary operations, the expected number of elementary operations is at most $O(N \log(\log N))$. \square

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. In this case the algorithm steps through all R rounds without successful termination. Using a similar calculation as in the proof of Lemma 13, the expected number of coin tosses to move from r to $r + 1$ is

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

Following each success, at most $N - 1$ phases are executed by processes lagging behind. Therefore, the expected number of complete phases before switching to original CIL is at most $3NR \leq 6N(\log N + 1)$. The expected number of elementary operations before switching is then $O(N(\log N)(\log(\log N)))$.

In [CIL94], it is shown that the expected number of elementary operations for the original CIL algorithm is $O(N^2)$. Therefore, the overall expected number of elementary operations is $O(N^2 \log(\log N))$. \square

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. By Lemma 11, this probability is at most $(1 - 0.511)^{\frac{R}{2}}$. Since $R = 2\lceil \log N \rceil$, we have

$$(1 - 0.511)^{\frac{R}{2}} \leq (1 - 0.511)^{\log N} < (0.5)^{\log N} = \frac{1}{N}.$$

\square

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 [Che05].

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.

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

The MWMMR 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 MWMMR 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.

To our best knowledge, our algorithm is faster (in terms of expected total work) than all other algorithms for dynamic adversaries, with the exception of [AKL99]. The algorithm of [AKL99] works in a SWSR setting and is $O(N \log N e^{\sqrt{\log N}})$ in expected total work. To have a fair comparison, we should take into account an emulation of MWMMR from SWSR, thus adding a linear slowdown to our complexity result.

However, the algorithm of [AKL99] is based on a primitive called *cooperative sharing*, in which processes propagate knowledge by reading and writing entire knowledge sets into registers. These knowledge sets grow throughout the execution and may eventually contain the identifiers and input values of all N processes. Therefore polynomial-size registers are necessary. In contrast, we require constant-size registers. Even if our registers are provided by an emulation from SWSR, it is sufficient to have logarithmic-size registers [IS92].

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).

Another possibility for future work is to consider *contention cost*, which measures the amount of conflict in memory access [AB04]. The contention cost for ModifiedCIL is high because, in a roughly synchronous execution, all N processes try to access a constant number of registers at the same time. It would be interesting to modify the algorithm further to reduce contention.

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

- [AB04] Y. Aumann and M.A. Bender. Efficient low-contention asynchronous consensus with the value-oblivious adversary scheduler. *Distributed Computing*, 2004. Accepted in 2004.
- [ACT00] M.K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash 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 Annual 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.
- [Che05] L. Cheung. Collection of PRISM models of the modified CIL algorithm, 2005. Available at <http://www.niii.ru.nl/~lcheung/mcil/>.
- [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.
- [HW90] M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
- [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>.