Wait-free Consensus in "In-phase" Multiprocessor Systems

Marina Papatriantafilou
Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
& CTI & Patras University, Greece
ptrianta@mpi-sb.mpg.de

Philippas Tsigas
Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
tsigas@mpi-sb.mpg.de

Abstract

In the *consensus* problem in a system with n processes, each process starts with a private input value and runs until it chooses irrevocably a decision value, which was the input value of some process of the system; moreover, all processes have to decide on the same value. This work deals with the problem of *wait-free*—fully resilient to processor crash and napping failures—consensus of n processes in an "in-phase" multiprocessor system. It proves the existence of a solution to the problem in this system by presenting a protocol which ensures that a process will reach decision within at most $n(n-3)/2+3$ steps of its own in the worst case, or within n steps if no process fails.

AMS Subject Classification (1991): 68M07, 68Q22, 68Q25, 90B12
CR Subject Classification (1991): B.3.2, B.4.3, B.4.5, C.1.2, C.2.4, C.4, D.1.3, D.4.1, D.4.5
Keywords & Phrases: Agreement, Consensus, Distributed Computation, Fault-Tolerance, Multiprocessor Systems, Napping Failures, Processor Crahses, Shared Memory, Wait-Free Synchronization.
Note: Partially supported by the EC ESPRIT II BRA ALCOM II (contr. # 7141).
This work will also appear in the Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing

1. Introduction

In the consensus problem in a system with n processes, each process starts with a private input value and runs until it chooses irrevocably a decision value, which has to be *valid*, i.e. to equal the input value of some process and *consistent*, i.e. it has to be the same value for all processes. Whereas this is no problem in an ideal, failure-free environment, it imposes certain constraints on the capabilities of an actual system, which is viable only if it permits protocols tolerant to failures. In a system with failures the consensus problem becomes a central issue of multiprocessor synchronization and coordination. Solutions which guarantee that each process decides after a certain number of its own steps, regardless of the other processes' relative speeds, are called *wait-free*. Wait-freedom is a desirable property in concurrent systems, since it helps in taking advantage of the inherent parallelism in the system by ensuring that no process may be blocked by others which might be slow, preempted, swapped out, delayed without warning by interrupts; moreover, it implies maximum tolerance to processor crash and napping failures.

As expected, such a fundamental problem received the attention of many researchers; as a result, many faces of the problem have been studied. In [12] and [20] it has been proven that in completely asynchronous systems –message passing and shared memory, respectively– not even one processor crash can be tolerated by a deterministic consensus protocol. In [9] the result is generalized for message passing systems; several critical parameters are identified and it is examined how they affect the number of faults that can be tolerated by a consensus protocol. In [14] shared memory data objects are partially classified according to the number of processes that can reach consensus in a wait-free manner using them. Recently, three groups with [7], [17] and [22], concurrently and independently have proven a conjecture first stated in [8], that even in the case when the agreement condition is weakened so that the decision values produced may differ, there is no protocol to tolerate k failures, where k is the maximum number of distinct values that may be chosen as decisions. Of particular interest was the introduction of algebraic and combinatorial topology in the study of these problems ([17, 22, 16, 18]). On the other hand, since in the asynchronous model the fault-tolerant consensus problem cannot be solved deterministically, solutions that have been given employ randomization or assume some form of synchrony. For a survey and for detailed references to those works cf. [11], [21].

If we want to sum these up, from the theoretical point of view, we have many surprising negative results, while the interest in the problem remains high. This is because, on one hand, it is interesting to develop a thorough understanding of the borders and relations between classes of objects with respect to their synchronization power; on the other hand, it is interesting to study more up-to-date architectures, which provide more fundamental synchronization primitives than just atomic reads and writes. Besides, it is easily noticeable that there is an important middle ground between the completely asynchronous and completely synchronous extreme; this middle ground is reasonable for modeling real concurrent systems. As a result, there is an increasing interest in the past few years in research towards defining and designing new architectures (e.g. the transactional memory [15]) or exploiting the properties of already existing ones (which are not all present in the theoretical models), in order to implement wait-free shared data objects ([13, 2, 6], to mention but a few).

1.1 Results and comparison with previous work

Following the direction mentioned above, in this work we consider the wait-free consensus problem in an "in-phase" multiprocessor system ([19]). In the same system model the wait-free clock synchronization problem has been earlier studied in [10]. Since by today's technology multiprocessor computers have large numbers of processors and since the probability of a crash increases with the number of processors in the system, it is vitally important to design multiprocessor systems that tolerate faults.

In an "in-phase" multiprocessor system processors share a common clock pulse; in the duration of a pulse a processor reads the shared data of *one* processor, does some local computation and updates *its own* shared data. (It should be pointed out that a processor cannot modify the contents of registers owned by other processors.) It is possible that processes in this system operate at very different speeds, i.e. miss pulses because of preemption, interrupts, page faults, or even processor crashes. So, although in a step a process atomically reads and writes, and, therefore, 2-process wait-free consensus is solvable ([14]), due to pulse misses and possible processor crashes, it is not obvious whether the n-process consensus problem can be solved wait-free in the system. This work presents a solution for the wait-free consensus problem with n processes in an "in-phase" multiprocessor system, thus answering an open question stated in [10] and showing that this system model/architecture is strong enough to support deterministic n process fault tolerant agreement. The protocol ensures that a process will reach decision within $n(n-3)/2+3$ steps of its own in the worst case, or within $n-1$ steps when no process misses a pulse.

To the best of our knowledge no solution to this problem has been given before. Previous results that could *serve* as solutions can be found in [1, 4, 5]. Those protocols are for the the asynchronous model, which supports only read/write atomically, and guarantee decision in exponential, $O(n^4)$ and $O(nlog^2n)$ expected number of steps, respectively, but at the cost of randomization. On the other hand, the n process wait-free consensus protocols presented in [14] require some form of multi-writer read-modify-write or more sophisticated primitives (augmented queue, memory-to-memory-swap); the system model studied here does not provide that directly. Besides, the three protocols presented in [9] as part of a thorough analysis of the cases when consensus is solvable in message-passing systems, cannot be translated into solutions for our system model. Protocols E2 and E3 (of [9]) assume synchronous processes (no napping faults) and totally ordered messages (not just FIFO channels), respectively. Protocol E1 relies much on the nature of synchronous message communication, i.e. that a process which had a long napping failure receives all the messages sent to it during that time interval as soon as it resumes execution. In our model a process has to make $n-1$ steps to learn about shared variable modifications; during that time it might suffer a new napping fault and this might be repeated unbounded many times.

2. The Computation Model

The system consists of n processes which are identified by distinct identity numbers, denoted by $P_1, \ldots, P_n$. The processes communicate via a set of single-writer, multi-reader atomic registers. Each one owns a subset of these registers. The owner of a register can write the

register while all the other processes can read it. A *step* by a process P_k consists of the following actions: (i) read by P_k of the shared registers owned by some process $P_{k'}$ ($k \neq k'$), (ii) transition of P_k's local state (program counter, local variables), and (iii) update of its own shared registers.

We consider "in-phase" multiprocessor systems, in which all processors share a common clock pulse. Each pulse is a (possibly empty) set of process names; the set of processes that *make a step* in the pulse. Each process can make at most one step in one pulse; if it does not make a step in some pulse it will be said to *miss* that pulse. A *configuration* is a tuple of process states and values of the shared variables. A system *execution* is a sequence $c_0\pi_1 c_1 \pi_2 \ldots$ of alternating pulses (denoted by π_x) and configurations (denoted by c_x); consecutive pulses are indexed with consecutive integers. Each configuration c_i in a system execution is derived from its directly preceding configuration c_{i-1} by the state transitions and the shared variable updates of the processes that make a step in pulse π_i; the reads of shared registers that occur in pulse π_i return the respective values of c_{i-1}, while the updates of the shared registers in the same pulse take place in unison to derive c_i. For any pulse π_j in any system execution E and for any process P_k, let $work(P_k, \pi_j)$ denote the number of steps that P_k made from the beginning of E (pulse π_1) until (and including) pulse π_j.

In the consensus problem each process P_k is given an input value v_k and is required to return an output value v'_k; we call the step when this happens the *return step* of P_k. If P_k makes its return step in some pulse π_j, it makes no more steps in subsequent pulses in the execution; if at some pulse in an execution P_k has not made its return step yet, it is called *undecided* in that pulse. A *wait-free consensus* protocol should satisfy the following *requirements* for every system execution:

Wait-freedom There should be a bounded number T such that: there is no process P_k and pulse π_j such that P_k is undecided in π_j and $work(P_k, \pi_j) > T$.

Validity For each process P_k, its output value v'_k should equal the input value $v_{k'}$ of some process $P_{k'}$ of the system.

Consistency For any processes P_k and $P_{k'}$ with output values v'_k and $v'_{k'}$ it should be $v'_k = v'_{k'}$.

3. Description of the Protocol

The protocol is described in C-like pseudocode in figures 1 and 2. We have adopted several conventions, like using capital case names for shared variables and capital boldface for calls to read/write shared registers. The following paragraphs describe the protocol more intuitively.

Each process P_k ($k \neq 1$) first plays a game with P_1. If it wins it is called *dominant* (in the set $\{P_1, ..., P_k\}$) and writes that information on its DOM_k shared register. Subsequently, the final decision is reached through stages in which partial decisions are made inductively. Let $D_{1..proc}$ denote the decision that would be taken if the system consisted only of processes $P_1, \ldots, P_{proc}$. The protocol tries to follow the inductive rule: $D_{1..n}$ is the input value of P_n, if P_n is dominant; otherwise, $D_{1..n} = D_{1..(n-1)}$, with $D_{1..1} = VAL_1$, the input value of P_1. Each process tries to find $D_{1..proc}$ for $proc = 1, \ldots, n$. After a process P_k makes a partial decision for $D_{1..proc}$, it writes the value v decided and the process identity $proc$ on its DEC_k

```
var (VAL_1, DOM_1, D_SET_1, DEC_1), ...,
    (VAL_n, DOM_n, D_SET_n, DEC_n): (valtype, boolean, 1..n, valtype) ;
 /* Shared var's; Initially all = (null, 0, 1, null) */


DECIDE(k, val) /* process P_k with input value val */
    var proc, d_set, d_set_p : 1..n ; /* Initially = 1, 1, 1 */
        dom, dom_p, rech: boolean ; /* Initially = 0, 0, 0 */
        dec, dec_p, val_p: valtype ;
    /* Local variables are globally binded in the module */

    proc READ&check(i) /* Read P_i's registers as first action in a step; check for decisions */
    begin
        READ (val_p, dom_p, d_set_p, dec_p) from (VAL_i, DOM_i, D_SET_i, DEC_i) ;
        if d_set_p ≥ i then d_set := d_set_p; dec := dec_p ; /* copy (partial) decision */
    end

    proc UPDATE() /* Update own var's as last action of step */
    begin
        WRITE (val, dom, d_set, dec) to (VAL_k, DOM_k, D_SET_k, DEC_k) ;
    end

    proc recheck(l : 1..n, dec_tmp: valtype)
    var i : 1..n ;
    begin
        for (i := 2; i < l; i++) do
            if i ≠ k do
              READ&check(i) ;
              if d_set ≥ l then proc := i := d_set ; /* advance current process ptr */
              if (i = l − 1 or i = l − 2 and k = l − 1) and d_set < l then d_set := l; dec := dec_tmp ;
              UPDATE() ;
            endif
        end_for
    end
```

Figure 1: Protocol DECIDE: (a) Shared Variables and auxiliary procedures

and D_SET_k shared registers, respectively. Let $D_k(proc) = v$ denote that mapping, which is the estimation of P_k for $D_{1..proc}$; P_k will finally return its $D_k(n)$. Since processes might miss pulses and are, therefore, asynchronous, deviations from the rule for deciding $D_{1..proc}$ are allowed, so that a fast process $P_{k'}$ can meet the wait-free requirement when P_1 and P_{proc} are so slow that the result of the game between them is not known at the time that $P_{k'}$ needs to find out $D_{1..proc}$. The deviation is that the fast process $P_{k'}$ (arbitrarily) considers that P_{proc} is not dominant and sets $D_k(proc) = D_k(proc - 1)$. Since this is done by a fast process, i.e. early enough, that information is available in the respective shared registers, for the slow

```
proc SafePhase(k) /* Estimate D_{1..k}= decision if system consited only of P_1..P_k */
begin
    READ&check(proc) ; /* step S_1.k: dominance in {P_1,...,P_k}? (check presence of P_1) */
    if d_set > 1 then proc := d_set ;
    else if val_p = null then dom = 1 ;
    if dom and k = 2 then d_set := 2; dec := val ;
    UPDATE() ;
    for (++proc; proc < k; proc++) do
        READ&check(proc) ;
        if d_set ≥ proc then proc := d_set ; /* advance current process ptr */
        else if ¬(dom) then d_set := proc ;
                if dom_p then dec := val_p ; /* else dec = D_k(proc) = D_k(proc − 1) */
        end_if ;
        if proc = k − 1 and d_set < k then d_set := proc := k ;
                if dom then dec := val ;
        end_if ;
        UPDATE() ;
    end_for
end

begin /* Main body of procedure DECIDE */
    if k = 1 then dec := val ; /* Step S_0.k: announce presence */
    UPDATE() ;
    if k ≠ 1 then SafePhase(k) ;
    for (++proc; proc ≤ n; proc++) do
        READ&check(proc) ;
        if d_set ≥ proc then proc := d_set ; /* advance current process ptr */
        else if dom_p then
                if k = 1 then d_set = proc; dec := val_p ;
                else dec_tmp := val_p; rech := 1 ; /* should recheck for "deviate" decisions */
        else d_set = proc ; /* consider proc not dominant ("deviate" decision) */
        UPDATE() ;
        if rech then recheck(proc, dec_tmp); rech := 0 ;
    end_for
    return(dec) ;
end
```

Figure 2: Protocol DECIDE: (b) Procedure $SafePhase(k)$ and main body of DECIDE

processes to find out about the deviation from the rule, and, therefore, decide consistently. Naturally, each process, in each one of its steps, checks whether a final decision or a more advanced —than the one it knows so far— partial decision is reached and adopts it, thus advancing its process scanning pointer, i.e local variable *proc*.

The game between P_1 and an arbitrary P_k is played as follows: Each process (including P_1), as its first action of the protocol —*announcement* or0- step— simply announces its

participation in the game by writing its input value on its VAL_k register. In its next step, each P_k ($k \neq 1$) reads the register of P_1 and becomes dominant only if it reads that P_1 has not made its announcement step yet (i.e. if $VAL_1 = null$); in all other cases P_k looses the game. P_1 plays with each P_{proc} in $\{P_2, \ldots, P_n\}$, one at a step and in that order (except when advancing its local variable $proc$), by checking whether P_{proc} is dominant; if not, P_{proc} will never become dominant, since it will read that P_1 has already made its announcement step. Thus, in that case the partial decision is $D_{1..proc} = D_{1..(proc-1)}$ and P_1 sets $D_1(proc) = D_1(proc-1)$. Otherwise —if P_{proc} is dominant—, P_1 safely (for reasons that become clear later in this section) concludes that $D_{1..proc} = VAL_{proc}$ and sets $D_1(proc) = VAL_{proc}$.

Each process P_k ($k \neq 1$) follows a protocol of 2 stages; first, using procedure $SafePhase(k)$ it estimates the partial decision $D_{1..k}$ and then continues for $D_{1..(k+1)}, \ldots, D_{1..n}$. After making an observation, we first describe the second phase, to give the intuition in a more clear way.

Observation 1 *If a process P_k is not dominant and reads $DOM_{k'} = 0$, then it knows that $P_{k'}$ will not become dominant in that execution, because P_1 has made its announcement step.*

Suppose that P_k has an estimation (from procedure $SafePhase(k)$) of the partial decision $D_{1..k}$. Then, for each $proc = k+1, \ldots, n$, it estimates $D_{1..proc}$ —in that order except from advances of the local variable $proc$— as follows: if it reads $DOM_{proc} = 0$, i.e. that P_{proc} is not dominant, then, only if P_k itself is not dominant, it can safely make a conclusion, namely that P_{proc} will never become dominant (by observation 1); otherwise, it cannot conclude in a wait-free manner (i.e. without waiting for P_1 and/or P_{proc} to make a step) anything about this future. Therefore, it arbitrarily considers P_{proc} not dominant and writes that $D_k(proc) = D_k(proc-1)$ (by leaving its DEC_k shared register unchanged and updating only its D_SET_k into $proc$). This implies that any other process P'_k ($k' < proc$), which will possibly later read that P_{proc} is dominant, before feeling free to decide $D_{k'}(proc) = VAL_{proc}$, it will have to *recheck* for earlier, deviate decisions about $D_{1..proc}$, among processes with identity P_x, s.t. $x < proc$ (i.e. among processes that might have had to decide about that with insufficient information). Note that P_1 need not recheck for deviations in decisions for any $D_{1..proc}$ if it reads that P_{proc} is dominant; if such a decision was made, this would have happened before the first step of P_{proc}, therefore before the 0-step of P_1 itself, by a process P_x, s.t. $x < proc$. Thus, P_1 would have read that decision before reading P_{proc}'s registers.

Now, let's see what the first phase is. In procedure $SafePhase(k)$, if P_k becomes dominant, in which case it should be $D_{1..k} = VAL_k$, it only has to check whether a fast process in $\{P_2, \ldots, P_{k-1}\}$ has earlier made a deviate decision about $D_{1..k}$. If P_k is not dominant it has to estimate $D_{1..(k-1)}$. In both cases it suffices for P_k to scan once the shared registers of each one of the processes in $\{P_2, \ldots, P_{k-1}\}$. If from a process P_{proc} it reads $DOM_{proc} = 0$, then it knows that P_{proc} will never become dominant (by observation 1) and decides $D_k(proc) = D_k(proc-1)$. Otherwise, if P_{proc} is dominant, it decides $D_k(proc) = VAL_{proc}$. It does not have to *recheck* for earlier, deviate decisions for $D_{1..proc}$, because if there was any, P_k would have read it before reading P_{proc}'s registers (by an argument similar to the one in the previous paragraph).

4. Correctness and Performance of the Protocol

For what follows in this section, we need some auxiliary notation/terminology. Consider an arbitrary system execution E:

• A process P_k maps a value v to the set $\{1, \ldots, proc\}$ in E if there exists a configuration such that $DEC_k = v$ and $D\_SET_k = proc$; we denote this mapping by $D_k(proc)$ and say that P_k *decides* this value v for the set $\{1, \ldots, proc\}$. This might happen either because P_k *copies* that decision from the shared register of some other process (in $READ\&check$) or because P_k *computes* that decision (in $SafePhase$ on in the main body of $DECIDE$). Note that P_k in its return step in E returns $D_k(n)$. In analogy with consistency for the final output value, we say that the decisions for a set $\{1, \ldots, proc\}$ are *consistently* made in E by the processes of a set P, if for any two $P_k, P_{k'} \in P$ that decide for $\{1, \ldots, proc\}$, it holds that $D_k(proc) = D_{k'}(proc)$.

• If s and s' are steps by processes, $s \rightarrow s'$ denotes that s precedes s', while $s \| \rightarrow s'$ denotes that s either precedes or is concurrent with s' in E; the latter is equivalent with $\neg(s' \rightarrow s)$. The step of P_k in which it reads P_{proc}'s shared registers for the first time, not during procedure *recheck*, is denoted by $S_{proc}.k$. Furthermore, $S_0.k$ denotes the announcement, or 0-step of P_k.

• A process P_k is *dominant* in E, if in the configuration after $S_1.k$ —and, henceforth in all subsequent configurations in E— it is $DOM_k = 1$.

Lemma 1 *(Wait-freedom) In a system with n processes, in any execution, each process P_k makes its return step after having made at most $n(n-3)/2 + 3$ steps.*

PROOF. (Sketch) Each process P_k decides $D_k(k)$ in k steps of its own; for each $D_k(x)$ $(k < x \leq n)$ it needs in the worst case —i.e. if *recheck* is necessary— $1 + (x - 3)$ steps of its own. Summing this all up, we have that in the worst case P_k terminates using $DECIDE$ in

$$k + \sum_{x=k+1}^{n} (x - 2) = n(n-3)/2 - k(k-5)/2$$

steps of its own. The largest value is when $k = 2$ (because P_1 never *rechecks* and terminates in at most n steps) and equals $n(n-3)/2 + 3$ steps. □

Lemma 2 *(Validity) In each execution each process which makes a return step, outputs a value that equals the input value of some process in the execution.*

PROOF. (Sketch) A process P_k that terminates returns the value $D_k(n)$ that decides and holds in its DEC_k shared variable. That value came either from a copy from some process $P_{k'}$'s $DEC_{k'}$ shared variable, or from an assignment to DEC_k of the input value VAL_{proc} of a process P_{proc}. In the latter case the lemma is straightforward; in the former case, if we trace back the origin of the value held in $DEC_{k'}$, by the same argument, we will find that it is an input value of some process in that execution. □

Lemma 3 *(Consistency) In a system with n processes the decisions for all sets* $\{1,\ldots,p\}$ $(1 \leq p \leq n)$ *are consistently made by the processes of the set* $\{P_1,\ldots,P_n\}$*, in every system execution.*

PROOF. (Sketch) This can be proven by induction on the number p.

Consider an arbitrary execution E in a system with n processes. For $p = 1$ the lemma holds because for any process P_{proc} in $\{P_1,\ldots,P_n\}$, in order to decide $D_{proc}.1$ it should be $S_1.proc \rightarrow S_0.1$, which implies that the input value of P_1 is available in its VAL_1 shared variable; therefore, for each P_{proc} that decides for $\{1,\ldots,1\}$, it will be $D_{proc}(1) = VAL_1$. Assume the lemma holds for all $p \leq k$; we will show that it also holds for $p = k+1$. From the induction hypothesis we know that the decisions for $\{1,\ldots,k\}$ are consistently made by the processes of the set $\{P_1,\ldots,P_n\}$. In order to prove that the decisions for $\{1,\ldots,k+1\}$ are also consistently made by the processes of the set $\{P_1,\ldots,P_n\}$, there are two cases to be considered:

Case 1: P_{k+1} is *not dominant* in E. Each $P_{proc} \in \{P_1,\ldots,P_k\} \cup \{P_{k+2},\ldots,P_n\}$ that makes $S_{k+1}.proc$ in E, reads $DOM_{k+1} = 0$ and, therefore, can only possibly decide $D_{proc}(k+1) = D_{proc}(k)$. Besides, P_{k+1} itself will also decide $D_{k+1}(k+1) = D_{k+1}(k)$. Since the decisions for $\{1,\ldots,k\}$ are consistently made by the processes in $\{P_1,\ldots,P_n\}$, the same follows in this case for the decisions for $\{1,\ldots,k+1\}$.

Case 2: P_{k+1} is *dominant* in E. There are two possibilities:

- Each P_{proc} in $\{P_1,\ldots,P_k\} \cup \{P_{k+2},\ldots,P_n\}$ that executes $S_{k+1}.proc$ and decides for $\{1,\ldots,k+1\}$, reads $DOM_{k+1} = 1$, and, therefore, sets $D_{proc}(k+1) = VAL_{k+1}$; similarly, P_{k+1} will set $D_{k+1}(k+1) = VAL_{k+1}$. Therefore, in this case the decisions for $\{1,\ldots,k,k+1\}$ are consistently made by the processes in $\{P_1,\ldots,P_n\}$.
- Some P_{proc} in $\{P_1,\ldots,P_k\}$, which executes $S_{k+1}.proc$ and decides for $\{1,\ldots,k+1\}$, reads $DOM_{k+1} = 0$. Clearly, $proc \neq 1$ and $proc \neq k+1$; otherwise, the dominance of P_{k+1} would be contradicted. P_{proc} computes $D_{proc}(k+1) = D_{proc}(k)$. It holds that:

$$
\begin{array}{ll}
S_{k+1}.proc \| \rightarrow S_1.(k+1) & (P_{k+1} \text{ is dominant but } P_{proc} \text{ reads } DOM_{k+1} = 0) \\
S_1.(k+1) \| \rightarrow S_0.1 & (P_{k+1} \text{ is dominant in } E) \\
S_1.(k+1) \rightarrow S_{proc}.(k+1) \| \rightarrow S_k.(k+1) & (1 < proc \leq k) \\
S_0.1 \rightarrow S_{proc}.1 \rightarrow S_{k+1}.1 & (0 < proc < k+1)
\end{array}
$$

Since in $S_{k+1}.proc$ process P_{proc} computes $D_{proc}(k+1) = D_{proc}(k)$, the above imply that P_1 and P_{k+1} will copy that decision from P_{proc}'s shared register in their $S_{proc}.1$ and $S_{proc}.(k+1)$ steps, respectively, i.e. before the steps in which they would have to decide for $\{1,\ldots,k,k+1\}$.

For any $P_{proc'}$ in $\{P_1,\ldots,P_k\}$ that executes $S_{k+1}.proc'$ and decides for $\{1,\ldots,k+1\}$, we have either that (a) $P_{proc'}$ reads $DOM_{k+1} = 0$ and, therefore, sets $D_{proc'}(k+1) = D_{proc'}(k)$, or that (b) $P_{proc'}$ reads $DOM_{k+1} = 1$ and executes *recheck*. In the latter case, since P_{proc}, $P_{proc'}$ read from DOM_{k+1} values $0, 1$, respectively, it holds that

$$S_{k+1}.proc \| \rightarrow S_1.(k+1) \rightarrow S_{k+1}.proc'$$

Moreover, from the protocol we have that

$$S_{k+1}.proc' \rightarrow \{recheck(k+1, VAL_{k+1}) \; steps \; by \; P_{proc'}\}$$

Since P_{proc} in $S_{k+1}.proc$ computes $D_{proc}(k+1) = D_{proc}(k)$, the above imply that $P_{proc'}$ will copy that decision from P_{proc}'s shared register during *recheck*.

For any $P_{proc''}$ in $\{P_{k+2}, \ldots, P_n\}$ that executes $S_{k+1}.proc''$ and decides for $\{1, \ldots, k+1\}$, we have the following: (a) if $P_{proc''}$ is dominant in E, since $k+1 < proc''$, from the protocol it follows that $P_{proc''}$ may only copy $D_{proc''}(k+1)$ from a process in $\{P_1, \ldots, P_k\}$. (b) if $P_{proc''}$ is not dominant in E and reads $DOM_{k+1} = 0$, it will decide $D_{proc''}(k+1) = D_{proc''}(k)$; if it reads $DOM_{k+1} = 1$ then we have the following about precedence relations of certain steps of these processes in E:

$$\begin{array}{ll}
S_{k+1}.proc \| \rightarrow S_1.(k+1) & (P_{k+1} \text{ is dominant but } P_{proc} \text{ reads } DOM_{x+1} = 0) \\
S_1.(k+1) \| \rightarrow S_0.1 & (P_{k+1} \text{ is dominant in } E) \\
S_0.1 \rightarrow S_1.(proc'') & (P_{proc''} \text{ is not dominant in } E) \\
S_1.(proc'') \rightarrow S_{proc}.(proc'') \rightarrow S_{k+1}.(proc'') & (\text{since } 1 < proc < k+1)
\end{array}$$

Since P_{proc} in $S_{k+1}.proc$ computes $D_{proc}(k+1) = D_{proc}(k)$, the above imply that in its $S_{proc}.(proc'')$ step, $P_{proc''}$, will copy that decision into its $DEC_{proc''}$.

Since the decisions for $\{1, \ldots, k\}$ are consistently made by the processes in $\{P_1, \ldots, P_n\}$, the same follows in this case for the decisions for $\{1, \ldots, k+1\}$.

□

Considering the requirements from a solution to the wait-free consensus problem, the previous lemmas imply the following theorem:

Theorem 1 *The* **DECIDE** *protocol correctly implements a wait-free solution to the consensus problem in an in-phase multiprocessor system with n processes, with* $T = n(n-3)/2 + 2$.

Conclusions

For the n-process consensus problem we have given a solution that tolerates processor crash and napping failures in an in-phase multiprocessor system, thus showing that this system model has a nice property, which is useful for fault-tolerant multiprocessor coordination and synchronization. Besides fault-tolerant consensus and clock synchronization, it is interesting to solve other problems efficiently and fault-tolerant in this system model; moreover, it would be useful to implement these algorithms.

References

1. K. Abrahamson On Achieving Consensus Using a Shared Memory. *Proceedings of PODC 1988*, pp. 291-302.
2. Juan Alemany and Edward W. Felten Performance Issues in Non-Blocking Synchronization on Shared Memory Multiprocessors. *Proceedings of PODC 1992*, pp. 125–134.
3. James Aspnes, Maurice Herlihy Wait-free data structures in the Asynchronous PRAM. *Proceedings of SPAA*'90, 340-349.
4. James Aspnes, Maurice Herlihy Fast Randomized Consensus Using Shared Memory. *Journal of Algorithms* **11**, pp. 441-461 (1990).
5. James Aspnes, Orli Warts Randomized Consensus in Expected $O(n \log^2 n)$ Operations Per Processor. *Proceedings of FOCS 1992*, pp. 137-146.
6. Greg Barnes A Method for Implementing Lock-Free Shared Data Structures. *Proceedings of SPAA 1993*, pp. 261–270.
7. Elisabeth Borowsky, Eli Gafni Generalized FLP Impossibility Results for t-resilient Asynchronous Computations. *Proceedings of STOC 1993*, pp. 91-100.
8. Soma Chaudhuri Agreement Is Harder Than Consensus: Set Consensus Problem in Totally Asynchronous System Systems. *Proceedings of PODC 1990*, pp. 311-324.
9. Danny Dolev, Cynthia Dwork, Larry Stockmeyer On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, vol. 34, No. 1, January 1987, pp. 77-97.
10. Shlomi Dolev, Jennifer L. Welch Wait-free clock synchronization. *Proceedings of PODC*'93, pp. 97-108.
11. M.J. Fischer The Consensus Problem in Unreliable Distributed Systems (A Brief Survey). *YALEU/DCS/RR-273*, June 1983.
12. Michael Fischer, Nancy Lynch and Michael Paterson Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, Vol. 32, No. 2, April 1985, pp. 374-382.
13. Maurice Herlihy A Methodology for Implementing Highly Concurrent Data Objects. *Proceedings of ACM PPoPP 1990*, pp. 197–206.
14. Maurice Herlihy Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 1, January 1991, pp. 124-149.
15. Maurice Herlihy, J.E.B. Moss Transactional Memory: Architectural Support for Lock-Free Data Structures. *Proceedings of ISCA 1993.*
16. Maurice Herlihy and Sergio Rajsbaum Set Consensus Usibg Arbitrary Objects. *Proceedings of PODC 1994*, pp.324-333.
17. Maurice Herlihy and Nir Shavit The Asynchronous Computability Theorem for t-resilent Tasks. *Proceedings of STOC 1993*, pp. 111-120.
18. Maurice Herlihy and Nir Shavit The Asynchronous Computability Theorem for

Wait-free Computation. *Proceedings of STOC 1994.*

19. K. Hwang Advanced Computer Architectures, Parallelism, Scalability, Programmability. McGraw-Hill, Inc. 1993.
20. Michael C. Loui and Hosame H. Abu-Amara Memory Requirements for Agreement among Unreliable Asynchronous Processes. *Advances in Computing Research*, Vol. 4 (1987), pp. 163-183
21. Nancy A. Lynch and Isaac Saias Distributed Algorithms: Lecture Notes *MIT/LCS/RSS 16 Research Seminar Series*
22. Michael Saks and Fotios Zaharoglou Wait-Free k-set Agreement is impossible: The topology of Public Knowledge. *Proceedings of STOC 1993*, pp. 101-110.