Wait-free Test-and-Set

(Extended Abstract)

Yehuda Afek[1,4], Eli Gafni[2*], John Tromp[3], and Paul M.B. Vitanyi[3,5]

[1] Computer Science Department, Tel-Aviv University, Ramat-Aviv 69978 Israel.
[2] Department of Computer Science, University of California, Los Angeles CA 90024.
[3] Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.
[4] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.
[5] Faculteit Wiskunde en Informatica, Universiteit van Amsterdam.

Abstract. This paper presents an economical, randomized, wait-free construction of an n-process *test-and-set* bit from read write registers. The test-and-set shared object has two atomic operations, test&set, which atomically reads the bit and sets its value to 1, and the reset operation that resets the bit to 0.

We identify two new complexity measures by which to evaluate wait-free algorithms: (a) The amount of randomness used, and (b) 'Parallel-Time'—the maximum sequential depth of an execution (i.e. longest chain of operations that must precede each other).

The previously best known algorithm for n-process test-and-set [Her91] takes an expected $\Omega(n^2)$ parallel time, and $\Omega(n^4)$ sequential time per operation, and $\Omega(n^2 \log n)$ space per processor. In contrast, our direct implementation improves this on all counts by using $O(\log n)$ coin flips, $O(\log n)$ parallel time, $O(n)$ sequential time, per operation, and $O(n)$ space per processor. Thus the question on the difference in the expected complexity of randomized constructions of concurrent objects from read/write registers is raised.

1 Introduction

A recent watershed paper by Herlihy [Her88] established the existence of a hierarchy of wait-free concurrent objects. The hierarchy classifies objects according to the number of processes among which these objects can solve the consensus problem. An object has a consensus number k if k is the maximum number of processors for which the object can be used to solve the consensus problem. Thus objects with higher consensus number cannot be deterministically implemented by employing objects with lower consensus numbers. Randomized implementations are required in order to implement higher objects in the hierarchy from lower ones. This gives rise to the intuitive feeling that as one goes up the hierarchy more randomness is needed.

* Supported by NSF Presidential Young Investigator Award under grant DCR84-51396 & matching funds from XEROX Co. under grant W881111.

The test-and-set consensus number is 2 while that of read/write atomic registers is 1. In this paper we propose an economical randomized implementation of n-processes test-and-set from read/write registers.

Each of the n processors that share the test-and-set object accesses it through the operations test&set and reset. The *sequential* specification of the object assumes that the two operations operate on a binary register which is initialized to 0. The test&set operation is like a swap of the register with a local variable whose value is 1, i.e. it atomically reads the register, writes 1 into it, and returns the value read. A reset operation writes 0 to the register and, returns no value. The specification also imposes a well-formedness requirement on the processors behavior, a processor can perform a reset only if its most recent operation was a test&set returning 0 (i.e. a successful test&set). Concurrent test-and-set is linearizable if each operation appears to occur in an indivisible time instant inside the operation's interval, and with respect to these serial instances the object satisfies the sequential specification above [Lam86].

Aside from their consensus number concurrent objects may be characterized by the following parameters:

- **Single use vs. Multi use:** A single use concurrent object can be accessed at most one time by each processor. It is in general easier to construct single use objects than multi use ones.
- **The number of processors that can access the object.**

Note that 2-process binary consensus is equivalent to a single use 2-process test-and-set object. Furthermore, the consensus number of n-process test-and-set is 2, but the consensus number of n-process *ternary* test-and-set is already ∞ [LAA87, Her91] (ternary test-and-set is a variant of read-modify-write).

In the same paper, [Her88], Herlihy established the existence of universal objects, whose consensus number is ∞. That is, objects which are at the top of the hierarchy. The randomized implementation of these objects, especially repeated consensus, from read/write registers has been the subject of research for a few years now [Abr88, Plo88, ADS89, Asp90, SSW90, Her91].

Previous work to implement n-process wait-free test&set proceeded by implementing a universal wait-free concurrent object in terms of repeated randomized n-process 'sticky bits', and bounding space by garbage collection, [Plo88]. The previously best known algorithm [Her91] for n-process test-and-set goes by way of constructing a general read-modify-write object using repeated randomized n-process consensus and a space garbage collection subroutine as building blocks. It uses $\Omega(n^2)$ mutually independent coin flips, to implement the best algorithm for n-process consensus [Asp90, SSW90, Her91], and its time complexity is $\Omega(n^4)$, per operation, and $\Omega(n^2 \log n)$ space per processor.

In this paper we give a simple direct implementation of wait-free n-process test-and-set in terms of atomic read/write variables. It improves the best previous method on all counts by using $O(\log n)$ flips of pairwise independent coins, $O(n)$ sequential time, per operation, and $O(n)$ space per processor.

A feature of shared memory algorithms is the fact that from time to time a processor may perform a set of (many, e.g. $O(n)$) operations, whose order of execution is unimportant. This gives rise to the question about the time it takes to complete such a set of operations. We advocate the view that processors communicate with memory via an interface, be it a communication network or a bus. In either case, getting to the memory in the former, or getting control of the bus in the latter, is the dominant factor. Thus in both cases it can be argued that making n simultaneous requests or a single one should not make much difference. In the case of communication networks the node issues n simultaneous requests to the network (very much in the same way that we allow a processor in a network to simultaneously send messages to all of its neighbors). We thus define the *parallel time* complexity measure in which we charge each set of simultaneous concurrent operations 1 unit of time. Under the new measure, the time complexity of Herlihy's algorithm is $\Omega(n^2)$, while the parallel time complexity of our algorithm is $O(\log n)$.

Our construction proceeds in two steps: First we construct a single-use n-process test-and-set object from a 2-process test-and-set object. Second we transform the single-use test-and-set object into a multi-use (repeated) test-and-set object. Any single use 2-process test-and-set object can be used as the building block of the first step, thus one can use the 2-process test-and-set construction of Tromp and Vitányi [TV90], or any 2-process consensus protocol [Asp90, AH90, SSW90, BR91]. All of these constructions have a constant expected complexity ($n = 2$).

Our protocols to extend test-and-set from 2-process single-use to n-process repeated are deterministic (the consensus number of n-process test-and-set is also 2 !). The randomness of the algorithm is only in the building block of a single-use 2-process test-and-set. Although it may take any 2-process test-and-set solution, it does not implement the n-process solution from the 2-process solution in the strict sense of the word (a la [Her88]). Rather, it is more like a compiler. All the formal proofs of correctness and of the complexity results are omitted from this extended abstract.

We view test-and-set as the wait-free equivalent of mutual exclusion. Capturing the test&set bit is analogous to getting the critical section (CS). However, in mutual exclusion, a processor that failed to get the CS is kept busy waiting until it enters the CS, while in test-and-set, a failed processor returns to the "remainder" section without going through the "Critical-Section". In test-and-set we should only be able to linearize its decision, that the "CS" is occupied, to be after we linearized another processor decision to enter the "CS" (which is in general not required from mutual exclusion protocols, where a processor may be denied entry at some point while at that point there is still no other processor that is about to enter the CS). The ease and simplicity of implementing test-and-set as opposed to consensus, stems from the fact that well punctuated rounds are now identified by the execution of the "CS" exit protocol (the reset operation). Thus, if the execution of the "CS" is slow, as long as the single processor executing it did not finish, this processor can still be accounted for rejecting other processors from entering the "CS". Therefore, we can rely on this single processor to do all the book-keeping so that we can precisely know what is obsolete and what is current data. This is not the case with consensus, fetch-and-add and other concurrent objects, where the moment any processor makes a decision, it can start a new round. Thus test-and-set is much more 'sequential' in this respect.

2 Informal Description of the n-Process Algorithm

We assume that the n processors perform test&set and reset operations according to the following generic template:

```
Forever do
    remainder_1 − section
    if test&set = 0 then
        zero − section
        reset
    fi
od
```

Conceptually any run of the n-process solution will be divided into rounds. A round is the interval between the termination of two consecutive reset operations. (The first round starts in the beginning of the run and ends in the first reset operation.) The test-and-set semantics together with the assumption that processors follow the above template guarantee that no two such operations are concurrent.

In each round a single-use n-process test-and-set protocol is executed. That is, an algorithm which is correct for an execution in which no processor will execute reset. The single-use algorithm is then composed into a multi-use algorithm (the desired solution).

We assume that in the single-use algorithm all variables are atomic single-writer-multi-reader registers.

2.1 Single-use n-process test-and-set

In this section we construct an n-process single-use test-and-set from similar 2-process building blocks. The basic idea is rather simple: have the n processors contest each other in a tennis like (binary tree) tournament [PF77]. Only one processor can win the tournament, and it is the winner in the single-use test-and-set, while all other processors lose. In each level of the tournament we match the winners of the previous level in pairs (according to a fixed binary tree) that compete each other using a 2-process test-and-set. A processor loses (returning 1) as soon as it loses in a match. The problem with this solution is that one processor could lose, while the eventual winner has not yet started to play. This kind of a scenario violates the serializability condition of n-process test-and-set. To overcome this problem each processor that loses will prevent any processor that has not yet started, from winning (see Figures 2.1 and 2.1).

The following facts hold for the single use Test&set:

```
Code for processor i single-use n-process test-and-set

function SINGLE-USE-TEST&SET
        for j=1 to n do if r[j].D = 1 then return(1) and exit fi od
        if tournament= 1 then                    %See next figure for the tournament.
                r[i].D := 1                                          %Close the door.
                return(1)
        else
                return(0)
        fi
end-function SINGLE-USE-TEST&SET
```

Fig. 1. Single use n-process test-and-set.

If processor i had lost, then any processor arriving after i had lost should also lose. This is implemented by an array, called "door", of atomic single-writer/multi-reader bits. That is, every processor has a 1-bit field in its own variable, together comprising an n-bit vector D. The door is open for a processor if all the $r[k].D$ bits are found $= 0$, $1 \leq k \leq n$. It is otherwise closed. Processor i closes the door by by setting $r[i].D$ to 1. Initially all the registers are set to 0.

1. In a bounded expected number of steps a processor returns either 0 or 1. Sketch: 2-process T&S involves a constant expected number of steps. Therefore the work at each floor is bounded and since the number of floors is bounded, the total number of steps is bounded.
2. Exactly one of all the test&set operations in a single use returns 0, and it starts before any other terminates. Sketch: by induction on the floor numbers of the tournament. Also the winner necessarily found the door open, which losers close before they terminate.

2.2 The multi-use test-and-set algorithm

For ease of exposition we present the multi-use algorithm using unbounded *round_numbers*. In the sequel we show that the system of round_numbers we use is required to satisfy the requirements of a *sequential* time stamp system, thus, replacing the unbounded numbers with bounded ones is conceptually straight forward using [LTV89, IL87].

Each processor has its own variable which it alone can write and all others can read. This variable is divided into a number of fields (so far we have seen *field-1*: the single-use 2-process variables, *field-2*: the D bit of the door, and *field-3*: the floor variable for the single-use n-process tournament). We add another field now, called round_number, which will determine whether the other fields are considered up to date, or out of date, in which case certain default values are assumed instead.

The code of the multi-use shell appears in Figure 3. The reset operation by processor p consists of atomically setting all the single-use test-and-set fields of p to their initial values and increasing p's round number by 1 (this is the only way by which round numbers are increased). Each test-and-set copies the

Code of the tournament for processor i

```
function TOURNAMENT
    for j=1 to log n do
        if 2-process-test&set(Sb[i, j])=1           %Each read in the 2-process-test&set
        then return(1) and exit fi                  %is performed as explained below.
    od
    return(0)

function READ(X) in 2-process-test&set(Sb[i, j])
    for each p in Sb[i, j] do
    rp := r[p]
        if rp.floor > j then return(1) and exit the protocol fi
        if rp.floor = j then return(rp.X) and exit the READ fi
    od
    return(x)                          %x is X's initial value in the 2-process-test&set
```

Fig. 2. The tournament function.

The implementation of the tournament follows the ideas of Peterson and Fischer [PF77] (who constructed n-processes mutual exclusion from 2-process mutual exclusion). We assume that in the 2-process single-use test-and-set each processor has one register that it writes and the other processor reads. The tournament consists of $\log n$ *floors*, corresponding to the levels of a binary tree tournament, whose leaves are the processors. Each internal node in the tree corresponds to a 2-process test-and-set fight between winning processors from each subtree of that node. A processor that test&set 0 in a 2-process test-and-set is considered to win the fight. Processors go up the floors, to the corresponding node in the binary tree. To this end, we add a *floor* field to the 2 process test-and-set register of each processor, indicating its progress up the floors of the tournament. Processor i that gets to the j'th floor looks for the group of processors that belong to the subtree sibling to the subtree it came from. This group is called $Sb[i,j]$ and assuming that processor ids are $0, 2, \ldots, n-1$,

$$Sb[i,j] = \{a[i,j], a[i,j]+1, \ldots, a[i,j]+2^{j-1}-1\}.$$

where $a[i,j] = 2^{j+1}\left\lfloor \frac{i}{2^j} \right\rfloor - 2^{j-1}\left(\left\lfloor \frac{i}{2^{j-1}} \right\rfloor - 1\right)$. Now we use the invariant that only one processor from that group may change its 2-process test&set variables at this level. Whenever, in the 2-process algorithm it reads a variable X of the other processor, it will now read the variable X of all processors in the sibling subtree, $Sb[i,j]$. There are three cases according to which the processor interprets the value of X. *Case 1:* If one of $Sb[i,j]$ has a higher floor number, then the processor has lost the fight (returns 1 and exits). *Case 2:* If none of $Sb[i,j]$ has floor number equal to that of i, processor i assumes the default initial value for X. *Case 3:* There is only one processor, k, in $Sb[i,j]$ with equal floor number (the winner of the sibling subtree), then processor i reads the value of X of processor k. A processor that arrives at the highest floor has won the tournament.

```
Code for processor i:

function TEST-AND-SET
    read all r[j] registers
    max := max_j {r[j].round_number}
    if (participated_i = max)
      then return(1) and exit                          %Already lost in this round
      else atomically{ r[i] := [2-process-variables: initial values,
                                D: 0,                    %set all i's single-use fields
                                floor: 0,                   %to their initial values
                                round_number: max]}
          participated_i := max
          return(SINGLE-USE-TEST&SET)          %Each read of a shared variable in the
                                   single-use is done via the extended-read of Figure 4
    fi
end-function TEST-AND-SET

function RESET
      atomically{ r[i] := [2-process-variables: initial values,
                                D: 0,                    %set all i's single-use fields
                                floor: 0,                   %to their initial values
                                round_number: participated_i + 1]}
end-function RESET
```

Fig. 3. Implementing multi-use test-and-set from single-use.

```
Code for extended-read of variable X of j (with initial value x) by processor
i:

extended-read(X, j):
      m := read(r[j])
      if r[j].round_number = r[i].round_number then
              return(m.2-process-variables.X) fi
      if r[j].round_number < r[i].round_number then
              return(x) fi                          {* x is the initial value of X *}
      if r[j].round_number > r[i].round_number then
              return 1 and exit the algorithm fi
```

Fig. 4. Extended read.

round number it participates in into a local variable 'participated', so it can later check for a new round. To start the algorithm a processor checks for the maximum round number of all other processors. If it has already participated in this round it returns 1. Otherwise it writes the new round number, and sets all its single-use variables to their initial values in one atomic write. When executing the single-use algorithm, any value which is read together with a round number which is less than its own, is interpreted to have the initial default value of the single-use algorithm. A processor executing the single-use algorithm, that reads a variable with a round number larger than its own, returns 1 for its test&set and exits.

For the code of how to interpret a read in the single-use algorithm see Figure 4.

To dispense with the unbounded round numbers we employ standard techniques like the scheme of [LTV89], or the Sequential Time Stamps System (STSS) [IL87]. The extra cost of such schemes is not more than $O(n)$ sequential time, and $O(1)$ parallel time.

The use of the sequential time stamp system (STSS) involves placing tokens on a graph of a complete tournament in which for any $n-1$ nodes there is a node that majorizes them. Each processor is assigned one such token whose location in the tournament corresponds to a time-stamp given by the STSS to that processor. The property of the STSS is that it gives a total order on token positions if the actions of advancing tokens are done sequentially. Usually, as the case is here, when a token is advanced it is advanced to a position in which it dominates, at the time of the advancement, all other token positions in the tournament.

The use of STSS in our algorithm is complicated because processors can either do advances in the STSS (sequentially, by the resetter) or adapt the time-stamp of the processor that dominates all others (when taking the maximum and putting it into their own round_number). This new operation of equating the time-stamp to that of the maximum time-stamp has two sources of complications: (1) This operation may take place concurrently with other such operations, and concurrently with a sequence of token advancement operations, and (2) the equating operation is not atomic, first the processor reads all other time-stamps, and then in a later operation it writes the value it decided to equate to. That is, concurrently with the sequential advancement of tokens in the tournament, some processors might move their tokens to location which they have observed another token in. In the following we extend the standard STSS to accommodate the new equate operation.

The specification of the *advance*, and time-stamp *comparison* operations in the extended STSS are the same as in the original STSS. The serial specification of the additional *equate* operation is that it acquires a time-stamp equal to the maximum time-stamp in the system. The *equate* and *advance* operations are implemented as follows: To equate the most dominant time-stamp a processor goes through two cycles of reading and writing. In the first cycle it reads the sequence-numbers of all the processors and selects the most dominant one if there is such a time-stamp, then it writes it as its tentative time-stamp. In the second cycle it again reads the sequence-numbers of all the processors and double check that the register it copied from, is still holding the same sequence-number and it is still the most dominant. If it is, then the equating processor updates its time-stamp to be valid (not tentative). If in the first cycle a processor finds out that its own valid time-stamp is the maximum one, then it does not change its time-stamp and does not go through the second cycle, it may assume its own time-stamp to be valid. In case the processor did not see a majorizing time-stamp in the first cycle, it failed to equate and returns without changing its time-stamp, in this case it may return 1 and exit the test-and-set operation. To label a new time-stamp (performed by the resetter), a processor reads twice all the time-stamps (valid and non-valid ones) and selects a new time-stamp that dominates all of the ones it read, according to the algorithm of [IL87]. When a processor reads a variable in the extended read of the test-and-set operation, it compares its valid time-stamp only with other valid time-stamps. One time-stamp is larger than the other if and only if it majorizes it in the time-stamp tournament graph. A processor with a tentative time-stamp is regarded to have a lower time-stamp.

3 Complexity Analysis

We now analyze the time complexity of the algorithm. Each 2 process test-and-set takes constant expected time. The outer algorithm requires once (at the beginning) that a processor will read all the time-stamps. This takes $O(n)$ sequential time and $O(1)$ parallel time. Similarly, in the single use algorithm reading the state of the door requires $O(n)$ sequential time and $O(1)$ parallel time. Finally, each level in the tournament requires $O(2^{level})$ sequential time and $O(1)$ parallel time and there are $\log n$ levels. Thus the sequential time adds up to $O(n)$ and the parallel time adds up to $O(\log n)$. Similar calculation shows that each test&set requires $O(\log n)$ coin flips.

4 Conclusion

This paper has presented: (1) A direct implementation of randomized wait-free test-and-set instead of an implementation in terms of repeated n process randomized consensus; (2) A deterministic implementation of wait-free n-process test-and set from wait-free 2-process test-and-set; (3) Introduced and argued the notion of randomness and parallel time as new cost measures in wait-free algorithms; and finally (4) Improved the complexity of existing constructions for test-and-set [Her88, Plo88] in terms of the new resources, random bits, and parallel time, as well as in terms of the standard complexity measures of sequential time and space.

Aside from the above, the paper raises the question of the randomized computational complexity (sequential time, parallel time, space, and number of coins) of implementing concurrent objects with consensus number k from concurrent objects with consensus number smaller than k.

Acknowledgments: We thank Michael Merritt and Michael Saks for helpful comments on earlier drafts.

References

[Abr88] K. Abrahamson. On achieving consensus using a shared memory. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 291–302, 1988.

[ADS89] Hagit Attiya, Danny Dolev, and Nir Shavit. Bounded polynomial randomized consensus. Extended Abstract, January 1989.

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, pages 281–294, September 1990.

[Asp90] J. Aspnes. Time and space efficient randomized consensus. In *Proc. of the Ninth ACM Symp. on Principles of Distributed Computing*, pages 325–331, August 1990.

[BR91] G. Bracha and O. Rachman. Randomized consensus in expected $o(n^2 logn)$ operations. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, October 1991.

[Her88] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. of the Seventh ACM Symp. on Principles of Distributed Computing*, pages 291–302, 1988.

[Her91] M. P. Herlihy. Randomized wait-free concurrent objects. In *Proc. of the Tenth ACM Symp. on Principles of Distributed Computing*, pages 11–22, 1991.

[IL87] A. Israeli and M. Li. Bounded time stamps. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 362–371, October 1987.

[LAA87] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research, JAI Press*, 4:163–183, 1987.

[Lam86] L. Lamport. On interprocess communication, parts i and ii. *Distributed Computing*, 1:77–101, 1986.

[LTV89] M. Li, J. Tromp, and P. M. B. Vitányi. How to construct concurrent wait-free variables. Technical Report CS-8916, CWI, Amsterdam, April 1989. See also: pp. 488-505 in: *Proc. International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 372, Springer Verlag, 1989.

[PF77] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proc. 9th ACM Symp. on Theory of Computing*, pages 91–97, 1977.

[Plo88] S. A. Plotkin. *Chapter 4: Sticky Bits and Universality of Consensus.* PhD thesis, M.I.T., August 1988.

[SSW90] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proc. of SODA 90*, December 1990.

[TV90] J. Tromp and P. M. B. Vitányi. Randomized wait-free test-and-set. Manuscript, November 1990.