-- --

Optimality of Wait-Free Atomic Multiwriter Variables

Ming Li

Computer Science Department, Waterloo University
Waterloo, Ontario N2L 3G1, Canada

Paul M.B. Vitanyi

Centrum voor Wiskunde en Informatica
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
and
Faculteit Wiskunde en Informatica, Universiteit van Amsterdam

ABSTRACT

Known implementations of concurrent wait-free atomic shared multiwriter variables use $\Theta(n)$ control bits per subvariable. It has been shown that implementations of sequential time-stamp systems require $\Omega(n)$ control bits per subvariable. We exhibit a sequential wait-free atomic shared multiwriter variable construction using $\log n$ control bits per subvariable. There arises the question of the optimality of concurrent implementations of the same, and of weak time-stamp systems. We also show that our solutions are self-stabilizing.

[Information Processing Letters, 43(1992), 107-112]

1980 Mathematics Subject Classification: 68C05, 68C25, 68A05, 68B20.

CR Categories: B.3.2, B.4.3, D.4.1, D.4.4.

Keywords and Phrases: Shared variable (register), concurrent reading and writing, atomicity, multiwriter variable, simulation.

Note: this paper will be published elsewhere.

1. Introduction

In [La] it is shown how an atomic variable---one whose accesses appear to be indivisible---shared between one writer and one reader, acting asynchronously and without waiting, can be constructed from lower level hardware rather than just assuming its existence. Multi-user atomic variables of that type have been constructed: [VA] using unbounded tags, and [LV], [Sc] using bounded tags.

Usually, with asynchronous readers and writers, atomicity of operations is simply assumed or enforced by synchronization primitives like semaphores. However, active serialization of asynchronous concurrent actions always implies waiting by one action for another. In contrast, our aim is to realize the maximum amount of parallellism inherent in concurrent systems by avoiding waiting altogether in our algorithms. In such a setting, serializability is *not* actively enforced, rather it is the result of a pre-established harmony in the way the executions of the algorithm by the various processors interact. Any one of the references, say [La], [VA], [AKKV], describes the problem area in some detail.

The point of departure is the solution of the following problem. (We keep the discussion informal). A flip-flop is a Boolean variable that can be read (tested) by one processor and written (set, reset, or changed) by another. Suppose, one is given atomic flip-flops as building blocks, and is asked to implement an atomic variable with range 0 to $n-1$, that can be written by one processor and read by another one. Of course, $\log_2 n$ flip-flops suffice to hold such a value. It is stipulated that the two processors are asynchronous and do not wait for one another. Suppose the writer gets stuck after it has written half the bits of the new value. If the reader executes a read while the writer is stuck, it obtains a value that consists of half the new value and half the old one. Obviously, this violates atomicity. Such atomic variables, correctly implemented [Pe], [La], serve as the building blocks for our constructions.

All constructions [LV, Sc] for implementing wait-free atomic variables which can be read and written by all n users, use $\Theta(n)$ bits of control information (time-stamps) per building block, be it 1-writer 1-reader subvariables as defined above [LV], or 1-writer n-reader (equivalent, multireader) subvariables as the construction in [Sc]. Following [IL], recent work [DS] aims at providing a general method for replacing unbounded time-stamps by bounded time-stamps in concurrent systems of multi-reader variables. Application to a multi-writer variable uses $\Theta(n)$ bits of control information per multireader subvariable.

Is the linear tag-size optimal? In [IL], an $\Omega(n)$ lower bound is proved for the tag-size for sequential binary comparison algorithms. Let us explain what this means in the current context. An algorithm is *sequential*, if it contains no overlapping operation executions. The algorithms considered above are *concurrent*, they allow overlapping. A lower bound proven for a sequential restriction of an algorithm holds *a fortiori* for the concurrent version. In our context *binary comparison* means that a user can determine

M. Li was supported by NSERC under grants OGP-0036747 and OGP-0046506. A preliminary discussion of this material appeared in the ICALP89 paper [LV].

the (apparent) atomic order between every two writes. However, it does not need to do so - we need only to be able to determine the *latest* write from a set of writes, and we do not care about the relative order among the remaining writes. In fact, the lower bound proven in [IL] is not relevant for the multiwriter problem, since we exhibit an $\log n$ upper bound for a sequential solution below. There arises the following open question.

Question. *Can the* $\Theta(n)$ *control bit implementation of concurrent wait-free atomic multiwriter variable be improved in terms of using less control bits per subvariable?*

2. Informal Preliminaries

We briefly discuss the used concepts and assume acquaintance with the published literature, say [VA], [IL], [LV], [DS], for the formal definitions. A concurrent system consists of a collection of sequential processes that communicate through shared data structures. The most basic such data structure is a shared variable. A user of such a variable V can start an action a (read or write) at any time when it is not engaged in another action, by invoking an ‘‘execute a’’ command on V, which finishes at some later time, possibly returning the value read. The semantics can be expressed in terms of a local value v of a process P and the global value contained in V. In absence of any other concurrent action the result of process P writing its local value v to V is that V:=v is executed, and the result of a process reading the global V is that v:=V is executed.

An *implementation* of V consists of a set of *protocols*, one for each user process, and a set of single-reader subvariables $X, Y, ..., Z$. An operation execution a by user process P on V consists of an execution of the associated protocol in which it applies some transformations on the subvariables $X, Y, ..., Z$, followed by returning a result to P. An implementation is *wait-free* if the number of subvariable accesses in an operation execution is bounded by a constant, which depends only on the number of users.

To emphasize the distinction between actions at a higher level, and those at a lower level, the word *operation execution*, or shortly *action*, is used for the former, and *subaction* is used for the latter.

Linearizability or atomicity is defined in terms of equivalence with a sequential system in which actions are mediated by a sequential scheduler that permits only one operation at a time to execute on any variable. A shared variable is *atomic*, if each read and write of it actually happens, or appears to take effect, instantaneously at some point between its invocation and response, irrespective of its actual duration.

3. Weak Time-Stamp System

We generalize the time-stamp system defined in [IL], removing all restrictions. This discussion assumes some knowledge of [IL].* A *sequential weak time-stamp system of order n* is $<G, f>$, where G is a set of nodes (or just numbers) and f is a (possibly partial) symmetric function from G^n to G such that the following n pebble game can be

* In [LV], ICALP89 version, we called this notion ‘generalized time-stamp system’, but ‘weak time-stamp system’ as coined later by [CS] seems more appropriate.

infinitely played on G,

- Initially all n pebbles are on an initial set of nodes.
- At each step, the adversary chooses a pebble and the pebbler has to move this pebble to a node v such that, with the remaining pebbles on nodes $v_1,..,v_{n-1}$, we have $f(\{v, v_1, \cdots, v_{n-1}\}) = v$.

We call f the labeling function. Obviously the new time-stamp system has most nice properties of the old time-stamp system of [IL]. However in [IL] it was proved that $2^n - 1$ nodes are needed for the sequential [IL]-time-stamp system of order n.

Theorem 1. *There is a sequential weak time-stamp system of order n, using n^2 nodes.*

Proof. The set of nodes G consists of $\{1,..,n\} \times \{1,..,n\}$. We exhibit the appropriate function f. The function f will always put pebble i on an element of $\{1, .., n\} \times \{i\}$. Initially, pebble i is on node $(1, i)$, $1 \le i \le n$. Suppose the pebbles are at nodes $(i_1, 1), \ldots, (i_n, n)$. If the adversary chooses pebble j, then it has to be moved to node (m, j) such that

$$j \equiv (m + \sum_{k=1, k \neq j}^{n} i_k) \bmod n. \tag{1}$$

That is, f is defined as

$$f(\{(m, j), (i_1, 1),..,(i_{j-1}, j-1), (i_{j+1}, j+1),..,(i_n, n)\}) = (m, j)\ ,$$

with m given by Equation (1). The effect is that the sum modulo n of the first coordinates, indicates a second coordinate which identifies the pebble which has just been moved. □

The labeling function in the proof has the advantage of being transparant and shows that the nodes can be described in $2 \log n$ bits for a sequential weak time-stamp system, rather than n bits as required in a sequential [IL]-time-stamp system. This suffices to illustrate the exponential difference between the two. But clearly f in the proof is not optimal. In fact, it has been recently shown that n^2 in the theorem can be improved to $2n - 1$ [CS].

4. A Sequential Multiwriter Algorithm Using n Tags

The following modification of the unbounded time-stamp algorithm in [VA], assuming the operation executions do not overlap, needs only $\log n$ bits to encode a (sequential) weak time-stamp system of order n. This is an application of the Theorem 1 above. (We use n^2 nodes since the nodes are (time-stamp , index) pairs, where the numbers of time-stamps and indices are both n.) The variable V can be read and written by users $u_1, u_2, \ldots, u_n$. It is implemented in subvariables $R_{i,j}$, $1 \le i, j \le n$, where $R_{i,j}$ can be written by user u_i and read by user u_j. The *tag*s (= time-stamps in this case) are just $1,...,n$ and are initialized with value 1.

About the same algorithm works with multireader variables as building blocks. Namely, use $R_1, \ldots, R_n$ as subvariables, where R_i is written by user u_i aand read by all users. Replace lines R1 and W1 by ‘‘Read $R_1, \ldots, R_n$’’, and replace line W3 by ‘‘Write

u_i **reads value: /* value** $:= V$***/**

R1) Read $R_{1,i},..,R_{n,i}$.

R2) Compute $m = (\sum_j tag@R_{j,i}) \bmod n$.

R3) **value** := value@$R_{m,i}$.

u_i **writes newvalue** **(**$1 \le i \le n$**): /*** V := **newvalue*/**

W1) Read $R_{1,i},..,R_{n,i}$.

W2) Compute m such that $i = (m + \sum_{j \ne i} tag@R_{j,i}) \bmod n$.

W3) Write $tag := m$ and value := **newvalue** to $R_{i,1},..,R_{i,n}$.

Figure. Sequential multiwriter algorithm using n tags.

$tag := m$ and value := **newvalue** to R_i''.

Theorem 2. *The algorithm implements a sequential wait-free atomic n-writer n-reader variable from* n^2 *subvariables, each of which is an atomic 1-writer 1-reader variable.*

Proof. In a sequential system, where the actions do not overlap, correctness is not an issue because of concurrency, but because of communication. In our setting, user u_i communicates with user u_j through $R_{i,j}$ and $R_{j,i}$. The system is sequential is equivalent to requiring there is a total precedence order on the operation executions in a system run. Since only the writes write, atomicity is ensured if a read returns the value written by the last preceding write. Since a writer selects its tag according to line W2, and a reader returns the value according to line R2, this is the case. The protocols consist of at most $2n$ subvariable accesses, hence the implementation is wait-free. □

This algorithm is partly related to the elegant 2-writer algorithm in [Bl]. The version with multireader subvariables was the first try at a concurrent multiwriter protocol by the second author in Februari '86. But the algorithm is not a *concurrent* atomic multiwriter variable by the following scenario for three writers and one reader: Initially, all writers have tag = 1, initializing the situation such that Writer 3 wrote last.

1. Reader starts to read and reads Writer 1's tag $(=1)$; and Writer 2's tag $(=1)$;
2. Writer 2 writes and sets $tag := 0$;
3. Writer 3 writes and sets $tag := 2$;
4. Reader reads Writer 3's tag $(=2)$, and returns the value held by Writer 1. (But Writer 1 has not written at all!)

5. Self-Stabilizing Property

Suppose all users in a network run the same program. A *state* consists of a vector of all values of the local variables, the shared variables, and the value of the program counter of the program execution, for each user. Each entry of the vector has its associated domain of values. The cartesian product of all these domains is called the *state space*. Clearly, from a fixed initial state a subset of the state space is *reacheable* by the system.

For most algorithms, there will be a subset of the state space which is *correct*, and its complement which is *forbidden*. The set of reacheable states should be a subset of the set of correct states. For instance, in mutual exclusion algorithms, states with two users in the critical zone are forbidden.

An algorithm is *self-stabilizing*, if started in any state of the state space, the system will move to a correct state within a finite number of steps.

Self-stability is a robustness property which guaranties that, whatever disturbance happens, if there is a long enough disturbance-free interval, the system will converge back to correct operation. This notion is due to Dijkstra [Dij]. Recent work is [BP].

If we take the sequential multiwriter construction, where the tag t of user u_i is always (implicitly) the pair (t,i), then whatever way we start the users in their protocols with whatever values of the local and global variables, the following happens. Let the system be started in an arbitrary state at time zero. Let S be the first system state reached when all users have executed at least one complete write. Let user u_j do the *first* write after the system reached state S. Subsequent to termination of this write execution by u_j, the system state will be correct. Namely, at the start of this write execution all program counters are zero (the system is sequential), each row $R_{i,1},..,R_{i,n}$ contains the same value in each of its subvariable elements (all users u_i, $i=1,..,n$, have executed a complete write), and the value of no local variable will ever be used again. After user u_j has finished its write execution, the sum of the values in $R_{1,i},..,R_{n,i}$ modulo n equals j, for $i=1,..,n$.

This argument is generalized in the obvious way to the weak timestamp system constructed. Viz., wherever the pebbles are placed on nodes in the graph, if a pebble is moved then it moves to a node (t,j) determined as follows. Prior to the move no pebble is on a node $(.,j)$. The sum of the second coordinates of the pebbles which are not moved and t, equals j *mod* n.

The reader may wonder that the multiwriter register had to do $n+1$ writes to be stabilized again. This is because there we used 1-writer 1-reader variables. Time stamp systems use multireader variables. Collapsing the rows of the multiwriter subvariable matrix $R_{i,j}$ to multireader variables R_i, the system stabilizes after only a single complete write. To get the system in its sequential mode again, this write must be executed after all program counters have been at zero at least once.

6. Conclusion

The current state of knowledge about optimality of time-stamp systems and multiwriter variables is the following. We consider implementations in terms of atomic multireader subvariables. For concurrent (and hence for sequential) [IL]-time-stamp systems, the upper bound per subvariable is $\Theta(n)$ control bits, according to [DS]. For sequential (and hence for concurrent) [IL]-time-stamp systems, the lower bound per subvariable is n control bits by [IL]. For sequential weak time-stamp systems the analogous upper bound per multiwriter subvariable is $2 \log n$ by the argument we gave, and $\log n + 1$ by [CS], while the lower bound is again, trivially, $\log n$. But for concurrent weak time-stamp systems, defined in analogy with concurent [IL]-time-stamp systems in [DS], the cited upper bound of $\Theta(n)$ control bits per subvariable must *a fortiori* also hold, while the best lower bound known is the trivial $\log n$ control bits per subvariable. This is the case which is relevant (rather, equivalent) to wait-free atomic multiwriter variable implementation. And it is here that the gap between upper bound and lower bound is wide open, while in all other cases it is essentially closed.

Acknowledgement.

We thank John Tromp for helpful comments.

References

[AKKV] B. Awerbuch, L. Kirousis, E. Kranakis and P.M.B. Vitányi, ‘‘A proof technique for register atomicity’’, In: Proc. 8th Conf. Found. Software Techn. & Theoret. Comp. Sci., Lecture Notes in Computer Science, Vol. 338, pp. 286-303, Springer Verlag, 1988.

[Bl] B. Bloom, ‘‘Constructing Two-writer Atomic Registers,’’ IEEE Transactions on Computers, **37**(1988), pp. 249-259

[BP] J. Burns and J. Pachel, ‘‘Uniform Self-Stabilizing Rings’’, ACM TOPLAS (1989), 330-344.

[CS] R. Cori, E. Sopana, ‘‘Some combinatorial aspects of time-stamp systems’’, Manuscript, Labri, Universite Bordeaux 1, France, June 1990.

[DS] D. Dolev and N. Shavit, ‘‘Bounded concurrent time-stamp systems are constructible, Extended Abstract’’, 21th ACM Symp. on Theory of Computing, 1989, 454-466.

[Dij] E.W. Dijkstra, ‘‘Self-stabilizing systems in spite of distributed control’’, CACM, 17:11(1974), 643-644.

[IL] A. Israeli and M. Li, ‘‘Bounded Time-Stamps’’, Proc. 28th IEEE Symp. on Foundations of Computer Science, 1987, pp. 371-382.

[LV] M. Li, P.M.B. Vitányi, ‘‘A very simple construction for atomic multiwriter register’’, Techn. Rept. TR-01-87, Aiken Computation Laboratory, Harvard University, November 1987. Published as: ‘‘How to share atomic concurrent wait-free variables’’, Proc. International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, Vol. 372, Springer Verlag, 1989, 488-505.

[La] L. Lamport, ‘‘On Interprocess Communication Parts I and II’’, Distributed Computing, Vol. 1, 1986, pp. 77-101.

[Pe] G.L. Peterson, ‘‘Concurrent reading while writing’’, ACM Transactions on Programming Languages and Systems, vol. 5, No.1, 1983, pp. 46-55.

[Sc] R. Schaffer, ‘‘On the correctness of atomic multi-writer registers,’’ Tech. Rept. MIT/LCS/TM-364, MIT Lab. for Computer Science, June 1988.

[VA] P.M.B. Vitányi and B. Awerbuch, ‘‘Atomic Shared Register Access by Asynchronous Hardware’’, Proc. 27th IEEE Symp. on Foundations of Computer Science, 1986, pp. 233-243. (Errata, **Ibid.**, 1987.)