Synchronization power depends on the register size

(Preliminary Version)

Yehuda Afek

AT&T Bell Laboratories and
Department of Computer Science
Tel-Aviv University
Israel 69978

Gideon Stupp

Department of Computer Science
Tel-Aviv University
Israel 69978

Abstract

Though it is common practice to treat synchronization primitives for multiprocessors as abstract data types, they are in reality machine instructions on registers. A crucial theoretical question with practical implications is the relationship between the size of the register and its computational power. We wish to study this question and choose as a first target the popular compare&swap *operation (which is the basis for many modern multiprocessor architectures). Our main results are:*

1. *We show that multi-valued consensus among n processes can be solved using a* compare&swap *register that can hold $\approx \frac{\log n}{\log \log n}$ values. That is, $n = (k-1)!$ where k is the number of values in the register, so the register has only $O(\log \log n)$ bits.*

2. *We prove that there is a dependency between register size and processes' ability to solve multi-valued consensus. The key to the proof is a novel method of reducing a multi-valued decision task with limited size* compare&swap *registers to the set-consensus problem [3] with read/write registers, allowing us to build on the recent powerful impossibility results of [2, 9, 18].*

3. *We further use the reduction method to prove a tight tradeoff between the space and time necessary to solve multi-valued consensus with a* compare&swap *register. Specifically, we show that any algorithm for multi-valued consensus among n processes with a k value* compare&swap *register, where $k \geq \frac{\log n}{\log \log n}$, must have a run that accesses the register $\Theta(\log_k n)$ times.*

The results of this paper suggest that a complexity hierarchy for multiprocessor synchronization operations should be based on the space complexity of synchronization registers and not on the number of so called "synchronization objects."

1 Introduction

We consider an asynchronous concurrent system consisting of n processes that communicate via shared memory. It is well known that the type of operations allowed on the shared memory cells greatly effects the kind of tasks that the n processes can solve. The first results of this type [6, 8, 11, 14] proved that if the only operations supported by the hardware are atomic read or write of memory cells (registers) then the system cannot implement a wait-free solution to the consensus problem, even if $n = 2$. (An algorithm is wait-free if each process finishes the algorithm in a finite number of steps regardless of the number of faults and the speed of other processes.) However, if the hardware also supports atomic *test-and-set* operations on single bits in the shared memory (as some old IBM machines do, and some modern machines such as Encore's Multimax, Sequent's Symmetry, DEC's Firefly and 6380 Corollary support) then 2 processes can solve the consensus problem among them but 3 processes cannot [8, 11, 14]. In his seminal paper Herlihy defined a hierarchy on abstract operation types, classifying them according to the number of processes among which these operations can solve consensus [8]. More specifically, an operation type, has consensus number k if any system supporting that operation type and the read/write operation type on an arbitrary size and number of registers, can be used to solve consensus among k processes, but cannot be used to solve consensus among $k+1$ processes. At the bottom level of the hierarchy are the weakest type of operations, with consensus number 1, e.g. atomic read/write of registers, while at the top are operation types such as compare&swap, whose consensus number is ∞.

Herlihy and Plotkin show that any wait-free synchronization task can be solved with any operation in the top level of the hierarchy [11, 17]. That is, they presented a universal construction for any sequentially-specified wait-free task. Jayanti and Toueg later presented simple and bounded universal constructions [12].

In this paper we define the space complexity of synchronization registers as the number of different values that the registers can hold, (which is exponential in the number of bits in the registers). We define the space complexity of an arbitrary synchronization object as the number of states in its sequential specification. We study the effect of the space complexity of the registers on the size of decision tasks they can solve, and on the efficiency with which such solutions are obtained. Thus we refine the class of operations in the top level by adding a space complexity measure to each type of operation.

We demonstrate our results by considering solutions to the multi-valued consensus problem with compare&swap registers. In the multi-valued consensus task each process proposes its own identity as its input and all processes decide on one unique identity as their output decision value. Validity requires that the elected identity must be one that has been proposed. The compare&swap register type which is supported by two contemporary machines (486-based Corollary and 68030-based NEWS) is in the top level of the hierarchy, and is defined as follows: c&s(a $\rightarrow$ b) operation, on register r, is:

c&s(a $\rightarrow$ b)(r)**: return**(v)
  $prev := r$;
  if $prev = a$ then $r := b$ **fi**
  return$(prev)$

Throughout the paper we assume that compare&swap registers are initialized (before any algorithm starts using them) to $\perp$. Also, in all the consensus algorithms throughout the paper, the first step of a process is to write its identity in a read/write register (thus notifying others that it is active in the algorithm). Consider the following sequence of three multi-valued consensus algorithms with decreasing space complexity and increasing time complexity:

1. To reach multi-valued consensus with a compare&swap register that can hold $n+1$ values (one of which is $\perp$) process i simply applies the operation $t :=$ c&s$(\perp \rightarrow i)$, and decides i if $t = \perp$ and t otherwise. This trivial algorithm has space complexity $n+1$ and number of accesses to the compare&swap register 1.

2. Another algorithm, that uses a sequence of $\log n$ compare&swap registers, $cs_1, \ldots cs_{\log n}$, each of which can hold 3 values, $\{\perp, 0, 1\}$ (i.e. the space complexity of the compare&swap operation here is $3^{\log n} = O(n)$), was suggested by Plotkin in [16, 17]. This algorithm proceeds in $\log n$ iterations. After posting its identity in a read/write register process i tries to set the values of the compare&swap registers to the values of its bits, one by one. Let b_i^j be the j-th bit in i's candidate id binary representation (initially, i's id is its candidate id). In the j-th iteration it performs t := c&s$(\perp \rightarrow b_i^j)$ on cs_j. If in any iteration the compare&swap operation fails (i.e., $t \neq \perp$) and $t \neq b_i^j$ then process i chooses an id from the ids registered in the read/write memory whose first j bits agree with the bits that have been decided so far. It then sets its candidate id to the chosen id and continues to the next iteration. By the end of the algorithm each process holds the elected id.

3. The last algorithm suggests the following new algorithm that uses one compare&swap that can hold $2\log n + 1$ values. In iteration j of the new algorithm process i tries to set the compare&swap register to (j, b_i^j). If it succeeds it records the pair (j, b_i^j) in the read/write memory. However, if it fails, it scans the memory for the largest recorded iteration number and adopts that process candidate id and iteration number. In this $O(\log n)$ space algorithm each process performs at most $\log n$ accesses to the compare&swap register.

This sequence of algorithms poses the following questions:

1. What is the optimal space complexity of a system of compare&swap registers that can solve multi-valued consensus (with unbounded read/write memory).

2. What is the optimal number of accesses to the compare&swap register in a multi-valued consensus algorithm if the register space complexity is k.

As for the first question we present a fourth algorithm with a compare&swap register of space complexity $O(\frac{\log n}{\log \log n})$ (and n single-writer multi-reader atomic registers of size $O(\log n)$ bits each, Section 3).

As a first step in the search for a matching lower bound, we prove in Section 5, that a compare&swap register with 3 values cannot solve multi-valued consensus for $n \geq 25$. (3 valued compare&swap has infinite consensus number by Herlihy's definition!) Thus proving that there is a dependency between register size (space complexity) and the processes' ability to solve multi-valued consensus.

We then continue to ask the following open question: Given a system with unbounded read/write memory and a k space complexity compare&swap registers, what is the maximum number of processes for which this system can solve multi-valued consensus. We conjecture that our algorithm is space complexity optimal, i.e., that the answer to the open question is $O(k!)$.

As for question 2 above, in our fourth algorithm each process might access the compare&swap register at most $O(\frac{\log n}{\log \log n})$ times. We prove in Section 4 a matching lower bound on the tradeoff between space and number of accesses to the compare&swap registers of any algorithm: Any algorithm for multi-valued consensus in any system with a k space complexity compare&swap and any number of read/write registers, must have a run in which at least one process accesses compare&swap registers at least $O(\log_k n)$ times.

Although we have presented our results in terms of the commercially available compare&swap operation, we can regenerate them in terms of an arbitrary read-modify-write operation type. The read-modify-write version of the lower bound proofs is considerably more complicated and we have thus chosen the compare&swap for the presentation of the paper.

Our method, called *register reduction*, uses a finite but unbounded space complexity read/write registers to reduce decision tasks that employ bounded size strong registers to the set consensus task that employs only read/write registers (and which is impossible by recent results of Borowsky, Gafni, Herlihy, Saks, Shavit, and Zaharoglou). To understand the uniqueness of the register reduction method let us informally define the valency of a decision task as the maximum valency of its initial state. The *register reduction* method consists of two major steps. In the first step we use $e < n$ processes to emulate the front-ends of the n processes of the decision task in question. Using the emulation we prove that the amount by which each access to a strong synchronization register might reduce the valency of a system (in the worse case) is dependent on the register space complexity. Roughly speaking, we show that in the worse case an access to a strong register (e.g. compare&swap) with space complexity k may reduce the system valency from v to no less than $\frac{v}{k-1}$.

This first step of our technique is used in Section 4 to prove that in order to reduce the system valency from n to 1 with strong registers whose space complexity is k an algorithm must access those registers at least $\log_k n$ times. The reduction technique also implies the following bound on the s-set consensus problem with n different inputs: In a system with read/write registers and compare&swap registers, at least $\log_k \frac{n}{s}$ accesses to the compare&swap are necessary to solve the problem (i.e. bringing down the valency from n to s).

In the second step of our *register reduction* method, we improve on the first step and show that a register whose space complexity is 3 cannot be used again and again to reduce the valency of the system. We show that after some number of accesses to the register its power is vanishing and it behaves like a read/write register, which cannot reduce the system valency any further. The fact that read/write registers cannot reduce the system valency from a to b where $a > b > 1$, also follows from [2]. This step is used to bound the valency of the initial state of a decision task that is solved by a register with space complexity 3.

Related work : Since the first impossibility proof of asynchronous agreement in a fail-stop distributed system, by Fischer Lynch and Paterson [6] there have been many papers extending and generalizing the proof method [3, 4, 5, 12]. Those papers extend it to deal with different models of communication and different models of failure. Some of the recent papers [2, 3, 9] addressed the number of failures that shared memory objects can with stand. This issue of t-resiliency have not been addressed in our paper. However, in [2] Borowsky and Gafni have introduced a simulation technique (different than ours) to address the power of various shared objects (without restriction on their space complexity). In their technique each simulating process tries to simulate all the codes of the simulated algorithm while in our technique we divide the codes among the simulators, each simulating several codes.

The paper proceeds as follows: Model and Definitions are in Section 2, the algorithm is presented in Section 3 while the lower bound proofs are in Sections 4 and 5. Conclusions and discussion are in Section 6.

2 Model and Definitions

Due to space limitations we omit the model section. We use the same model and notation as in [8].

A *consensus protocol* is a system of n processes where each process starts with an input value from some domain D. The processes communicate with one another by applying operations to the shared memory and eventually agree on a common input value and halt. A consensus protocol is required to be: (a) Consistent: distinct processes never decide on distinct values, (b) Wait-free: each process decides after a finite number of steps and (c) Valid: the common decision value is the input to some process.

The sequential specification of a consensus object is that all decide operations return the argument value of the first decide [10, 17]. A wait free linearizable implementation of a consensus object is called a consensus protocol.

A state s of a consensus protocol is said to be *univalent* if there is only one decision value for all possible run fragments that start at s. This means that the system as a whole eliminated all but one input value as a possible decision value.

A *multi-valued consensus* (MVC) protocol is a consensus protocol where the domain D is the processes' names and the input to process i is its own id i (also called leader election).

The *k-set consensus problem* is a generalization of the consensus problem. Informally, a *k-set consensus protocol* is a system of n processes where each process start with an input value from some domain D. The processes communicate with one another by applying operations to the shared memory registers and eventually each decide on a value from a set $D' \subset D$ where $|D'| \leq k$. A k-set consensus protocol is required to be: (a) Consistent: $|D'| \leq k$, (b) Wait-free: each process decides after a finite number of steps and (c) Valid: the decision value of any process is the input to some process.

A compare&swap-(k) object is a compare&swap as defined in the introduction, and whose register can hold k different values, from the set $C = \{\perp, 0, 1, \ldots, k-2\}$. A compare&swap operation is said to *succeed* if the operation has changed the register's value.

3 Multi-Valued Consensus for $(k-1)!$ processes using compare&swap-(k)

In this section we present a multi-valued consensus (MVC) algorithm among n processes, using one compare&swap-($\approx \frac{\log n}{\log \log n}$).

W.l.o.g. assume that $n = (k-1)!$. Let P be the $(k-1)!$ different permutations of $\{0, 1, \ldots, k-2\}$. The algorithm uses a one to one mapping $F : \{1, 2, \ldots, n\} \rightarrow P$, from the processes ids to P. With each id i of process P_i we associate a permutation $L(i)$ of C as follows: $L(i) = \perp || F(i)$.

The structure of the algorithm (given in Figure 1, whose code is given in Figure 1, is similar to that of algorithm 3 from the introduction. As mentioned each process starts the algorithm by posting its id in its swmr atomic register. The algorithm operates in $(k-1)$ phases, in each of which the processes agree on one symbol of $L(l)$, the permutation representing the leader identity, l. In the first phase each process i, proposes the first symbol of $L(i)$ (=**Symbol**$(1,i)$). In general, if in phase p its proposal is agreed upon then in the next phase it proposes **Symbol**$(p+1, i)$ the $p+1$-st symbol of its proposed candidate-id.

However, if another symbol is chosen in phase p, the process selects a new candidate-id, from those posted in the shared memory, and tries to elect it as a leader. The candidate-id selected is such that the first p symbols in its associated sequence agree with those that have been decided so far. At the end of each phase each process records in it's swmr atomic register its current candidate-id and its phase number.

4 The time complexity of MVC with compare&swap-(k)

Theorem 1 *Let B be an algorithm for MVC among n processes that share only one* compare&swap-(k)*, csk, and any number of atomic registers. Then, there must be a run in B such that at least one process performs $O(\log_{k-1}(n))$ operations on csk in this run.*

Proof: Let d be the maximum number such that there is a run of B in which some process performs d c&s() operations on csk.

The proof uses the following claim:

Claim 1 *Given an algorithm B with parameter d as above, then there is a $(k-1)^d$-set consensus algorithm, B', for $\lfloor \frac{n}{d+1} \rfloor$ processes, that uses only atomic registers.*

From [2, 9, 18] we know that the l-set consensus problem cannot be solved with atomic registers for $l < n$. Hence, by the claim $(k-1)^d \geq \lfloor \frac{n}{d+1} \rfloor$, or $d \geq \log_{k-1}(\lfloor \frac{n}{d+1} \rfloor) \geq \log_{k-1}(n) - \log_{k-1} 2(d+1)$. Thus

Operation Multi-valued-consensus(id:value) returns(value)

The **MVC** operation accepts as parameter a processes's id and returns the same value from the processes' domain to all calling processes. The value is chosen from the id's of processes that started executing.

```
shared
  R_i, 1 <= i <= n, n atomic registers each consisting of a pair
                (Phase_i,CandidateId_i).
        -- CandidateId_i holds the current candidate of process i for being the consensus
        -- value. It is initialized to i. Phase_i holds the number of bits that process i
        -- knows that have been agreed upon. It is initialized to 1.

  csk          a compare&swap-(k)
        -- For a given k we show MVC among ≈ k! processes
local
  CurrentV,NextV,Result: symbols;
        -- Used in the c&s() operation, see line 6.
Code:
1   i := id; R_i := (1, id);
        -- Atomically set Phase_i to 1 and CandidateId_i to id.
2   let w be the minimum x s.t. (k-1)!/(k-1-x)! >= n
        -- k phases are needed for deciding among (k - 1)! processes. If there are less
        -- processes (n < (k - 1)!) then less phases are executed.
3   while Phase_i <= w do
4     CurrentV := Symbol(Phase_i,CandidateId_i);
5     NextV    := Symbol(Phase_i+1,CandidateId_i);
6     Result   := c&s(CurrentV → NextV);
        -- Use the compare&swap to decide on the next symbol in the name of the consensus
        -- result (as represented by L(i)).
7     if    (Result = CurrentV)
8     then R_i := (Phase_i + 1,CandidateId_i);
        -- If successful, try to decide on the next symbol in the name of CandidateId_i.
      else
        -- If failed then find another CandidateId whose name prefix equals the string of
        -- symbols that were agreed upon.
9       begin
10      let k be s.t. Phase_k := max_{j ∈ {1,...,n}} Phase_j;
11      if Phase_k > Phase_i then R_i := R_k;
        -- If some process is in a more advanced phase then copy the data of the most
        -- advanced such process.
        else
          begin ;
12        let k be s.t. Phase_k = Phase_i and
              Symbol(Phase_k+1,CandidateId_k) = Result;
        -- If all processes are in the same phase then choose a new candidate from the
        -- valid processes that have the decide symbol as the next symbol in their name.
13        R_i := R_k;
          end ;
        end ;
      od ;
14  return (CandidateId_i);
```

Function Symbol(p:value,i:value) returns(value)

Function **Symbol** computes the p-th symbol in the mapping $L(i)$

```
1  return (the p-th symbol of L(i));
```

Figure 1: Code for MULTI-VALUED CONSENSUS using compare&swap-($\approx \frac{\log n}{\log \log n}$).

$d = \Omega(\log_{k-1}(n))$ because,

$$\lim_{n\to\infty} \frac{d}{\log_{k-1} n} \geq \lim_{n\to\infty} \frac{\log_{k-1}(n) - \log_{k-1} 2(d+1)}{\log_{k-1} n} = 1$$

Proof of Claim: W.l.o.g. we assume that all atomic registers in B are swmr [1, 19, 13, 15]. Each of the $\frac{n}{d+1}$ (w.l.o.g. assume that $d+1$ divides n) processes of B' is assigned $d+1$ of the processes (front ends) of B, $(v_{i_1}, \ldots, v_{i_{d+1}})$ $i = 1, \ldots, \frac{n}{d+1}$. Each process emulates each of its assigned front ends in a particular way to be described. Henceforth, the processes of B' are called *emulators* and each front end of B is called *virtual process*.

We begin by an outline of the proof structure in the following two paragraphs. A more detailed description follows there after.

The intuition of the proof is as follows: Each emulator emulates the run of its virtual processes in B, until one of them reaches a decision state, at which point the emulator adopts that decision value. The emulation proceeds as follows: Each emulator iteratively emulates the front ends of its virtual processes one by one until each is about to perform a successful c&s() operation (some c&s() operations can be immediately identified as unsuccessful and emulated by internally returning the current value assumed to be in the compare&swap). At this point it chooses one of its virtual processes to succeed and advances all others, by failing them in the c&s() operation (because their operation "takes place" right after the successful one which changes the value in the compare&swap thus causing the others to fail). Thus, the emulator assumes a specific value as the next value of *csk*, the compare&swap register. After emulating the successful operation of the chosen virtual process the emulator emulates no more steps of this virtual process, effectively emulating a fail-stop failure of this process. If several emulators choose the same "next value" at about the same time, then effectively only one virtual process from all of them is succeeding in its c&s() operation. Although the emulators do not know which is the successful one, it does not matter since they are all marked dead. The problem arises when two or more emulators choose different "next value" for *csk*. This problem is solved by allowing each of the emulators to assume a different "next value" in *csk*, thus proceeding in different runs of B. The main idea is that at this point virtual processes of emulators that chose different "next value"s proceed to emulate different runs of B. In each run it is assumed that a different value was successfully writtenin the *csk*. While this is not a legal run for all of the virtual processes together, it is legal for each group that chose the same sequence of "next value"s. The virtual processes of each group of emulators "assumes" that all the virtual processes of the other groups have fail-stopped at the splitting point (where one run departs from the other). Roughly speaking this entails that each operation on the compare&swap register might cause each group of emulators to break into at most $k-1$ disjoint subgroups, each continuing a different run of B.

The rest of the proof, given after this paragraph, details the book keeping necessary for the above emulation process. The essence of it is that each emulator keeps a history variable that records the sequence of values it believes the *csk* had. Each write to an atomic register by a virtual process is tagged by the value of the history of the corresponding emulator at the time of the write.

The reduction actually emulates a full-information version of algorithm B [2, 7, 9, 18]. Every atomic register A is replaced by a list that holds all the values that have ever been written to the register (single-writer!). Each value written is tagged with the history of the writing emulator and is appended to the register. As old histories of an emulator are always a prefix of its newer histories, and since we use swmr registers the sequence of histories in a register list are prefixes of each other as well.

Each read verifies that it reads a value from a process which is together with it in the same run by observing the history marks of the values in the read register (which might force it not to take the most recent value of that register).

The history variable of each emulator is a sequence, $a_1, a_2, \ldots$, where $a_i \in C$. Initially all histories contain the singleton $\perp$ which is the initial value of *csk*. The history describes the sequence of values that were written in *csk*. When an emulator emulates an operation of a virtual process on *csk*, it first snapshots all the history variables, and then executes an internal function, calc_h*() on the snapshot. The function calc_h*() chooses from all the history variables one history (called h^*_{calc}) which is maximal in the sense that it is not a prefix of any other history, and that the emulator's own history variable is a prefix of (or equal to) h^*_{calc}. Then, the emulation returns the last symbol of h^*_{calc} as the result of the virtual process operation on *csk*. If the emulated c&s() is assumed successful, then the emulator appends to h^*_{calc} the new value (that was assigned by the c&s() operation) and writes the result to its history variable. A crucial point in our emulation is that of all the emulators that concurrently wrote the same new history, only one emulated process actually succeeds and we don't know which one. But, as they all fail-stop immediately after the operation, it does not matter which one of them succeeded. If the c&s() operation failed, the emulator writes h^*_{calc} as is to it's history variable.

When an emulator emulates a read from register A, it looks for the entry in A whose history tag is the largest prefix of the emulator's history (or equal to it) and returns the value of that entry. However, if there is an entry in the list such that the current history of the emulator is a prefix of that entry's history, then the emulator chooses the largest (latest) such entry, updates its history to the history of that entry and returns the value of that entry. Obviously, the new history of the emulator is an extension of its previous history.

Each virtual process is in one of two states, *alive* or *dead*. Initially all are alive. A virtual process that is marked dead is assumed to have fail-stopped in the emulation, and thus no more steps of its front-end will be emulated.

Emulator q executes the front-ends of it's alive virtual processes one by one. In each it proceeds until either the front-end reaches a decision, or a c&s() operation that succeeds (assuming the csk value is the last symbol in h^*_{calc}).

In the former, q terminates returning the same decision value as the virtual process. In the later case, successful c&s(), it does not execute it (meaning not returning the result to the virtual process nor updating the history variable) but switches to any other alive virtual process whose next operation is not a successful c&s(), and continues to emulate it. In this procedure the emulator might return to an alive virtual process more than once because its history may change while executing other virtual processes, thus making some c&s() non-successful. Note that when returning to a previously stopped c&s() the snapshot is re-executed.

If the next step of all the alive virtual processes of the emulator is a successful c&s() then the emulator performs the following procedure: (1) one virtual process, v_x, whose next operation is, say, c&s($a \rightarrow b$) is arbitrarily chosen, (2) v_x is marked dead, (3) b is appended to the end of h^*_{calc} and the result is written to the history variable, (4) the next operation of all alive virtual processes (which is of the form c&s($a \rightarrow *$)) is emulated by assuming the c&s() failed and returning b to the calling virtual processes (no reads or writes of the history variables are needed in this step as we use $h^*_{calc}||b$).

This procedure is repeated until the emulator reaches a decision state with one of its virtual processes. In any run of B, each emulator might kill at most d virtual processes (as there are at most d c&s() operations). Since each emulator has at least $d+1$ virtual processes, the emulators must reach a decision state.

We now show that the emulation is correct, that is, that it implements a $(k-1)^d$-set consensus between the $\frac{n}{d+1}$ emulators.

Let $R' = \pi'_1\pi'_2\ldots$ be a run of the emulation B' where each π'_i is either an internal operation or an external operation of one of the emulators. Let $R'_k = \pi'_1\ldots\pi'_k$ be a prefix of R'. For any operation π'_i in R', let $h_{\pi'_i}$ be the value of the history variable of the emulator executing π'_i in R'_{i-1} (i.e. in the state of B' at the end of R'_{i-1}).

The string $h_{\pi'_i}$ is called *maximal history* if there is no π'_j s.t. $h_{\pi'_i}$ is a prefix of $h_{\pi'_j}$ in R'. We denote such history h^*. Note that h^*_{calc} is a computation of h^* at a point in the run. We denote a list of symbols from C (the legal values of the compare&swap) as α.

Some of the operations in every R' are bookkeeping operations of the emulation. Others, are operations that directly correspond to operations of the front ends of the virtual processes.

The operations of B' that are operations of the front ends of B are called *virtual operations*. There are three types of virtual operations:

1. *Read* operations of shared memory variable x into some internal memory r, $\pi' = (r := \text{read}(x))$ that corresponds to a read operation in the front end of the virtual process, $r := \text{read}(x)$.

2. *Write* operations of value v to shared memory variable x, $\pi' = (\text{write}_x(h', v))$ that correspond to a write operation in the front end of the virtual process, $\text{write}_x(v)$.

3. *History operations.* There are two kinds of history operations: a *history read*, π' =(r:=SNAPSHOT), where SNAPSHOT is an atomic snapshot of all the history variables, and a *history write*, $\pi' = \text{write}_{h'}(v)$ where h' is the emulator's history variable. A history read and write pair corresponds to a c&s() operation of the type r :=c&s($a \rightarrow b$) (in case of a successful c&s() operation the pair of history read and write corresponds to several c&s() operations, one for each active virtual process). Each emulator first executes a history read and then calculates h^*_{calc} using calc_h*() as defined before. If the last symbol of h^*_{calc} is not a ($h^*_{calc} = \alpha m,\ m \neq a$) then h^*_{calc} is written to the history variable by the history write ($\pi' = \text{write}_{h'}(h^*_{calc})$) and the pair of history read and write corresponds to a failed c&s($a \rightarrow b$) operation. If the last symbol of

h^*_{calc} is a ($h^*_{calc} = \alpha a$) and there is another virtual process whose next operation is not c&s($a \rightarrow *$) then the emulator stops executing the current virtual process and continues with the other virtual process. Note that if the next operation of the new virtual process is c&s(), we do not need to re-execute a history read but can continue the run as in the failure case. The pair of history read (performed while emulating the previous virtual process) and history write (performed while emulating the new virtual process) are mapped to a single failed c&s(). If however, the next operation of the new virtual process was not a c&s() operation then the history read is degenerate (as it does not have a matching history write) and is skipped over. The last case is when the next operation of all virtual processes is c&s($a \rightarrow *$). In this case, $h^*_{calc}||b$ is written by the history write (π' $=$write$_{h'}(h^*_{calc}||b)$) and the virtual process is marked dead (thus emulating a fail-stop). Also, the next operation of all other virtual processes is emulated by returning b as the current value of the compare&swap. The pair of history read and write corresponds in this case to all the c&s() operations of all the virtual processes where the first operation is c&s($a \rightarrow b$).

Let $R'|_h = \pi'_1 \pi'_2 \ldots$ be a subsequence of R' such that every operation π'_i is a virtual operation and for each π'_i, $h_{\pi'_i}$ in R' is a prefix of h. We define $\rho^{R'|_h}$ to be a run of B that corresponds to $R'|_h$ in that every operation π' of $R'|_h$ is mapped to it's corresponding operation π and every history read and write pair is mapped to a c&s() operation.

The order of the operations in $\rho^{R'|_h}$ is inductively defined: assume $R'_k|_h = \pi'_1 \ldots \pi'_k$ is already mapped to $\rho^{R'_k|_h} = \pi_1 \ldots \pi_l$. Then, in the full paper we show how to map π'_{k+1}.

Lemma 1.1 *For any run R' of B' and any h^*,*

1. *$\rho^{R'|_{h^*}}$ is a legal run of B.*

2. *The value in the csk register after $\rho^{R'|_{h^*}}$ is the last value in the string h^*.*

Proof: Omitted due to space limitations.

To complete the proof of the claim, we first note that every emulator decides on the value that its first virtual process decides upon. Also, in every run R' all virtual processes whose operations correspond to the same $\rho^{R'|_{h^*}}$ decide on the same value. But, as there are at most $(k-1)^d$ different h^* and as every virtual process must belong to some such ρ, the emulation implements a $(k-1)^d$-set consensus between $\frac{n}{d+1}$ emulators. ∎

5 The Power of compare&swap-(3)

In this section we combine an extension of the emulation technique with the FLP technique, to prove an impossibility result. That is, we prove that compare&swap-(3) cannot solve multi-valued consensus among more than 25 processes.

Theorem 2 *There is no algorithm for MVC among n processes, $n \geq 25$, that share only one* compare&swap-(3)*, cs3, and any number of atomic registers.*

Proof: Assume to the contrary that there is such an algorithm, B. The proof proceeds by first proving the existence of a special run prefix in any such algorithm and second using that run prefix to construct an infinite run (similar to the infinite runs constructed by FLP type proofs), thus showing that B is not wait free in contradiction to the assumption.

Claim 2 *If there is such an algorithm B, then there is a prefix of a run of B that ends in state s^* such that*

1. *the next operations of at least two processes in s^*, P_i and P_j, are c&s($o_i \rightarrow m_i$), and c&s($o_j \rightarrow m_j$) respectively, $o_i \neq o_j$ (o stands for old value).*

2. *there are at least two runs of B starting at s^* in which neither P_i, nor P_j take steps and each of these runs reach a different decision value.*

Proof of Claim: We employ the emulation technique from Theorem 1 to prove the claim. That is, if such a B exists and there is no such state s^* then there is an algorithm that solves 4-set consensus among five processes using only read/write registers, contradicting the impossibility result on set consensus.

Given B and $n \geq 25$ front ends, assign $d \geq 5$ virtual processes to each of five emulators. Each emulator follows the same rules as in Theorem 1 to execute its 4-set consensus algorithm, with the following exception: Whenever the next step of all the live virtual processes of an emulator is a successful c&s() operation, and there are at most two symbols in h^*_{calc}, ($\perp m_1$) (in which case it must have at least 3 live processes) the emulator selects *two* virtual processes whose next operation is the same, c&s($o \rightarrow m$), $o \neq m$. The emulator marks one of these processes as dead and suspends the other. One of them is assumed to have died right after taking the step, while the other is suspended just before taking that step. There must be such two because there are only two possible values for m. Then, the emulator continues as in Theorem 1 by appending

m to h^*_{calc}, advancing each live virtual process by one operation (which must be a failing c&s($o \rightarrow *$)), and proceeding with the emulation process.

If the next operation of all live virtual processes of an emulator are successful c&s(), and there are three symbols in h^*_{calc} then the emulator chooses one live virtual process (which it must have) and simulates its run as if it runs in isolation from the current state to a decision state.

Obviously, there will never be more then three symbols in h^*_{calc} since no emulator will ever execute a history write of the fourth symbol.

First, we show that lemma 1.1 holds for this version of the emulation. The proof is omitted due to space limitations.

We can now prove Claim 2 by contradiction.

Assume the first item of Claim 2 is false. Let R' be any run of B', h^* any maximal history of R'. If there are three symbols, $\perp, m_1, m_2$, in h^* then there are at least two virtual processes marked dead before executing the operation c&s($\perp \rightarrow m_1$), and two marked dead before executing the operation c&s($m_1 \rightarrow m_2$). From this follows that in $\rho^{B'|_{h^*}}$ there is at least one process whose next operation is c&s($\perp \rightarrow m_1$),$m_1 \neq \perp$ and one whose next operation is c&s($m_1 \rightarrow m_2$),$m_2 \neq m_1$ (as one of each kind of compare&swap might have already been executed). From the assumption we know that such a state cannot be reached which means that all maximal histories in all runs of B' are of length less then three.

1. As there is at most one history write operation, there are at most two virtual processes marked dead in each emulator. Since each emulator has five virtual processes it will never enter an infinite loop. Also, since B is wait free, so is B'.

2. As there are at most two different maximal histories, the emulation emulates at most two different runs of B which means B' solves 2-consensus among 5 processes using atomic registers only.

From this follows that there must be a state s^* for which point 1 of the claim holds.

Assume the first item of Claim 2 holds but the second item is false. Let R'_k be a prefix run of B' such that there is a h^* in R'_k of three symbols, $\perp, m_1, m_2$. As mentioned before, there is in $\rho^{B'_k|_{h^*}}$ at least one process whose next operation is c&s($\perp \rightarrow m_1$) and one whose next operation is c&s($m_1 \rightarrow m_2$).

1. Since every emulator that reads or writes a history of three symbols decides by doing internal operations, there are at most four dead virtual processes in each emulator which means the emulation do not enter an infinite loop. As B is wait-free, so is B'.

2. From the assumption we know that $\rho^{B'_k|_{h^*}}$ is univalent. It immediately follows that $B'_k|_{h^*}$ is univalent (since no matter which run of B you follow, the same decision value is chosen). Every emulator that reads (or writes) a history of three values immediately decides (by internally computing the only valent value). As there are at most four different maximal histories of three symbols, the emulation emulates at most four different runs of B which means B' solves 4-set consensus among 5 processes using atomic registers only.

Hence, the claim follows. ∎

Following Claim 2 we assume a state s^*. Then by an FLP type argument [6, 8] there is a state s^{**} such that (1) s^{**} in poly-valent, (2) in s^{**} P_i and P_j are as in the claim and, (3) there are two other processes P_{k_1} and P_{k_2} in s^{**} such that the next operation of each is an operation on the $cs3$ register which takes the system to two different univalent state. W.l.o.g. assume that the operations of P_{k_1} and P_{k_2} are c&s($w_0 \rightarrow w_1$) and c&s($w_0 \rightarrow w_2$) respectively, $w_1 \neq w_2$. We can further assume w.l.o.g. (by the claim) that the operation of one of P_i or P_j (assume P_i) is either c&s($w_1 \rightarrow w_2$), or c&s($w_1 \rightarrow w_0$). Then, in the former case (P_i: c&s($w_1 \rightarrow w_2$)), a fifth process (one other than P_i, P_j, P_{k_1} and P_{k_2}) cannot distinguish between the run s^{**}, P_{k_2} and the run s^{**}, P_{k_1}, P_i. In the latter case (P_i: c&s($w_1 \rightarrow w_0$)), a fifth process cannot distinguish between the run s^{**}, P_{k_2} and the run $s^{**}, P_{k_1}, P_i, P_{k_2}$. Thus state S^{**} is not possible and by the FLP type argument algorithm B is not wait-free. ∎

Remark: A much more complicated proof gives a tighter bound, that compare&swap-(3) can not be used for solving MVC for 4 or more processes.

6 Conclusions

This paper addresses the dependency of the size of a shared memory object and it's ability to solve decision tasks. We examine that ability both in terms of computability (how hard a problem such an object can solve) and of complexity (how long will it take to solve a given problem using that object). While giving an optimal tradeoff between the space of a strong object (compare&swap) and the complexity of hard decision problems that use it, we succeeded only in showing that the computability issue is relevant, by giving a specific result (that compare&swap-(3) can not be used for solving multi-valued consensus between more than 25 processes). This leaves open the following question: "given a system with unbounded read/write memory and k space complexity compare&swap registers, what is the maximum number of processes, n_k, for which this system can solve multi-valued consensus" ([14, 8] proved that $n_1 = 1$, $n_2 = 2$, here we showed that $n_3 \leq 25$, we conjecture that $n_k = O(k!)$).

In reaching our results, we developed a new method for reducing multi-valued consensus algorithms between n processes that share strong synchronization objects to l-set consensus algorithms between $e < n$ processes that share only read/write registers (which immediately implies that $e = l$). Our use of the reduction method exemplifies its flexibility. Not only do we use it in itself for proving the time complexity lower bound, but we actually show that certain states must exist in runs of consensus related algorithms in the computability section.

Several extensions come to mind. For example, all our results can be generalized to RMW objects and in fact, perhaps to any linearizable object. Herein, we focus on algorithms that use one copy of the strong register. We claim that one copy of any shared memory register of k values is as strong as several copies of the same object of k_i values each, where $k \geq \prod k_i$. This result is immediate for a general RMW object. The generalization of our register reduction method to systems with several strong objects is one direction to proceed in (perhaps by managing a separate set of history variables for each strong register). Extending our complexity results to k-set algorithms is straightforward.

Acknowledgments: We are in debt to Nir Shavit for his encouragement and insightful discussions. In fact, the question of the dependency on the space complexity was first raised for a take-home exam in Nir's course in Spring of 1992. We also thank Yishay Mansour, Manor Mendel, Michael Merritt and Michael Saks for many helpful discussions.

References

[1] B. Bloom. Constructing two-writer atomic registers. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 249–259, 1987.

[2] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[3] S. Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proc. of the Ninth ACM Symp. on Principles of Distributed Computing (PODC)*, pages 311–324, August 1990.

[4] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:228–323, April 1988.

[6] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[7] G. N. Frederickson and N. Lynch. The impact of synchronous communication on the problem of electing a leader in a ring. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 493–503, 1984.

[8] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[9] M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[10] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] M. 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.

[12] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms: Springer-Verlag LNCS*, November 1992.

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

[14] 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.

[15] G. L. Peterson and J. E. Burns. Concurrent reading while writing ii : The multi-writer case. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 383–392, October 1987.

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

[17] S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. of the 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, Edmonton, Alberta, Canada, August 1989.

[18] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[19] A. K. Singh, J. H. Anderson, and M. G. Gouda. The elusive atomic register revisited. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 206–221, 1987.

From afek Fri Sep 10 11:19:39 1993 Return-Path: ¡afek¿ Received: from gemini.math.tau.ac.il by math.tau.ac.il (5.67/math.tau-921104) id AA22845; Fri, 10 Sep 93 12:19:38 +0300 Received: by gemini.math.tau.ac.il (5.67/math.sub-st921020) id AA16300; Fri, 10 Sep 93 11:19:43 +0200 Date: Fri, 10 Sep 93 11:19:43 +0200 From: afek Message-Id: ¡9309100919.AA16300@gemini.math.tau.ac.il¿ To: stupp Subject: focs.bbl Status: OR

References

[1] B. Bloom. Constructing two-writer atomic registers. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 249–259, 1987.

[2] E. Borowsky and E. Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[3] S. Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. In *Proc. of the Ninth ACM Symp. on Principles of Distributed Computing (PODC)*, pages 311–324, August 1990.

[4] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:228–323, April 1988.

[6] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[7] G. N. Frederickson and N. A. Lynch. The impact of synchronous communication on the problem of electing a leader in a ring. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 493–503, 1984.

[8] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, January 1991.

[9] M. Herlihy and N. Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[10] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] 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.

[12] P. Jayanti and S. Toueg. Some results on the impossibiltiy, universality, and decidability of consensus. In *Proceedings of the 6th International Workshop on Distributed Algorithms: Springer-Verlag LNCS*, November 1992.

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

[14] 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.

[15] G. L. Peterson and J. E. Burns. Concurrent reading while writing ii : The multi-writer case. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 383–392, October 1987.

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

[17] S. A. Plotkin. Sticky bits and universality of consensus. In *Proc. of the 8th ACM Symp. on Principles of Distributed Computing*, pages 159–175, Edmonton, Alberta, Canada, August 1989.

[18] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proc. 25th ACM Symp. on Theory of Computing*, May 1993.

[19] A. K. Singh, J. H. Anderson, and M. G. Gouda. The elusive atomic register revisited. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 206–221, 1987.