Delimiting the Power of Bounded Size Synchronization Objects

(Extended Abstract)

Yehuda Afek
Department of Computer Science
Tel-Aviv University
Israel 69978

Gideon Stupp
Department of Computer Science
Tel-Aviv University
Israel 69978

Abstract

Theoretically, various shared synchronization objects, such as compare&swap and arbitrary read-modify-write registers, are *universal* [10, 20]. That is, any sequentially specified task can be solved in a concurrent system that supports these objects and a large enough number of shared read/write registers. Are these objects indeed almighty? Or, are there other considerations that have to be kept in mind when analyzing their computation power. In this paper we show that progressively larger objects of these types are more powerful (larger in the number of different values they can hold). This provides a refinement of Herlihy's hierarchy.

We consider a shared memory system with unbounded read/write memory and a size k compare&swap register. Let n_k be the maximum number of processes that can elect a leader in such a system (in a wait-free manner). In [1] we present an election algorithm for $O(k!)$ processes in such a system, i.e. showing that n_k is at least $O(k!)$. However, on the lower bound side only n_1, n_2, and n_3 were shown to be bounded [1, 10, 18], while for $k > 3$ it was not known whether such a bound exists. Here we prove that for any k, n_k is bounded by $O(k^{(k^2+3)})$ that is, at most $O(k^{(k^2+3)})$ processes can elect a leader in such a system [1]. Hence, the more values a strong shared memory object can hold the stronger it is!

The proof of the lower bound (lower bound on space, which is an upper bound on number of processes) combines several techniques that were recently developed with novel new techniques, which are interesting on their own.

[1] This gives a lower bound on space because it implies that $O(k^{(k^2+3)})$ processes or more need $\Omega(k)$ size compare&swap register to elect a leader wait-freely.

1 Introduction

This paper addresses the relationship between the size of a shared synchronization object and its ability to solve synchronization tasks. Strong synchronization objects (e.g. compare&swap, or load-link-store-conditional) are in reality complex machine instructions on shared memory addresses. Some of the strong objects are classified as being universal [10]. However, this classification assumes that we have an unlimited number of such objects. The question is [1, 16]: What is the relationship between the size of a synchronization task and the size of the objects its solution requires. In particular, can bounded size synchronization objects solve any task? i.e., are bounded size strong objects universal as well?

We consider an asynchronous concurrent system that consists of n processes that communicate via shared memory. It is by now well known that the type of operations supported on the shared memory cells greatly effects the kind of tasks that the n processes can solve. In [9, 10, 13, 18] it is proved that if only atomic read or write operations are supported by the hardware then the system cannot wait-freely reach consensus (or solve the leader election 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 single bit atomic *test-and-set* operations are supported by the hardware (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 [2]) then 2 processes can elect a leader and solve the consensus problem, but 3 processes can solve neither [10, 13, 18]. Herlihy continued this sequence and defined a hierarchy on abstract operation types, classifying them according to the number of processes among which these operations can solve consensus (and thus leader election) [10]. At the top level of Herlihy's hierarchy are operation types such as compare&swap, whose consensus number is ∞. Moreover, Herlihy showed that given these operation types any sequentially specified problem can be solved [10] (Jayanti and Toueg have later modified his construction to be bounded [15]).

In this paper we show that the top level of the hierarchy is farther refined by a space complexity parameter. That is, the more values the register in the top level can hold, the more powerful it is.

To demonstrate this we choose the popular compare&swap object, which is being used in many new multiprocessor machines. The synchronization task we consider is the leader election problem, or in other words, the multi-valued consensus problem. We present a lower bound on the size of a compare&swap register that may be used to elect a leader among n processes. However, instead of a lower bound on space (size) we provide an upper bound on the number of processes that can elect a leader with a compare&swap register that can hold at most k different values. Clearly these lower bound and upper bound imply each other. Specifically we show that at most $\eta = O(k^{(k^2+3)})$ processes can elect a leader using a k size compare&swap.

In the leader election task each process proposes its own identity for election and all processes elect one identity as their leader. Validity requires that processes choose an identity only if it was proposed. The compare&swap register type is at the top level of Herlihy's 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($prev$)

Although our results are presented in terms of the commercially available compare&swap register, we see it as a test case and believe that the results can be generalized to an arbitrary read-modify-write register type.

The lower bound proof in this paper is a non-trivial extension of the *register reduction* technique that we have introduced in [1]. In this method we reduce a decision task that employs a bounded size strong register to the set-consensus task in a system that supports only read/write registers (which is impossible by the results of Borowsky, Gafni, Herlihy, Saks, Shavit, and Zaharoglou). Clearly, the reduction process may use an unbounded but finite number of read/write registers (just as an NP-complete reduction proof may use any polynomial time overhead). That is, we show that if there is a leader election algorithm A between η processes in a system that uses a size k compare&swap register then, there is an l-set-consensus algorithm B among m processes, $l < m$, in a system that supports only read/write registers.

As is the case in reduction based NP-complete proofs, the crux of the proof is in the reduction itself, that is in the emulation algorithm. Clearly, the less restrictive the emulation algorithm is, the stronger the proof we get. In [1] we present an emulation algorithm that places severe restrictions on the leader election algorithm, such as bounded time (that is, bounded number of accesses to the compare&swap register), and unbounded number of processes contending for the leadership.

In this paper we overcome the above limitations by introducing a new emulation algorithm. The new emulation algorithm is capable of emulating a run of A in which there is a bounded number of processes, such that the emulation produces an impossible set-consensus algorithm even if there is an unbounded number of steps in the run. This emulation is detailed in Section 3.

Related work: Since the first impossibility proof of asynchronous agreement in a fail-stop distributed system, by Fischer Lynch and Paterson [9] there have been many papers extending and generalizing the proof method [6, 7, 8, 15]. Those papers extend the method to deal with different models of communication and different models of failure. Some of the recent papers [4, 6, 11] addressed the number of failures that shared memory objects can withstand. This issue of t-resiliency have not been addressed in our paper. However, in [4] 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.

In the past year several other papers that address questions similar to ours have appeared. In [14, 16] Jayanti, Kleinberg and Mullainathan consider several models, one of which is similar to ours, and ask how many copies of a particular type of object are necessary to reach binary consensus among n processes. Jayanti studies this question when different types of objects are allowed, thus bringing up the question of the robustness of Herlihy's hierarchy (H_m^r). Kleinberg and Mullainathan show that if n processes can elect a leader with one copy of object O (without any other registers!) then this object can solve binary consensus among at most $\lfloor n/2 \rfloor$ processes. In both of these papers, when the system was allowed to have an unbounded number of read/write registers the authors study the effect of the number of objects on the power of the system. Each object they have considered cannot solve consensus for an arbitrary number of processes (i.e., its consensus number was bounded by some k). While here we consider an object (compare&swap) whose consensus number is ∞, even when it can hold only three values. In another paper, [5] Burns Cruz and Loui consider the effect of the size of registers on their ability to solve leader election. However, Burns et. al. make two strong assumptions, (1) that each read-modify-write register may be written at most once, and (2) that the system is equipped only with read-modify-write registers (there are no read/write registers). Thus, the proof is simpler, since the state of the system is rendered by the state of the strong registers. Under these assumptions, Burns et. al. prove that a k value read-modify-write register can elect a leader among at most $k-1$ processes (compared with the $O(k!)$ in our model) and in general if there are several such registers then the number of processes is the product of the registers sizes (where the size of a register is the number of values it can hold).

The paper proceeds as follows: Model and Definitions are in Section 2, the lower bound is described in Section 3. Due to space limitations not all the proofs of the lemmas are included.

2 Model and Definitions

Due to space limitations we omit most of the model section. We use the same model and notation as in [10].

The *Leader Election* (LE) problem (or Multi valued consensus) is a variation of the consensus problem in which the inputs domain is the processors' names and the input of processor i is its own identity. A *LE protocol* is a system of n processes where each non failed process starts with its identity as the input value. The processes communicate with one another by applying operations to the shared memory registers and eventually elect (decide on) a common input identity and halt. A LE protocol is required to be: (a) Consistent: distinct processes never elect distinct identities, (b) Wait-free: each process elects a leader after a finite number of steps and (c) Valid: the common identity elected is the identity of a process that have proposed itself for leadership.

The sequential specification of a LE object is that all elect operations return the identity of the processor that applied the first operation [12, 20]. A wait free linearizable implementation of a LE object is called a LE protocol.

The *k-set consensus problem* is a generalization of the consensus problem [6]. Informally, a *k-set 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 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 $\Sigma = \{\perp, 0, 1, \ldots, k-2\}$. A compare&swap operation is said to *succeed* if the operation changes the register's value.

3 The impossibility of LE among $O(k^{(k^2+3)})$ processes with compare&swap-(k)

Theorem 1 *There is no leader election algorithm among $\eta = O(k^{(k^2+3)})$ processes using one compare&swap-(k) and any number of atomic registers.*

Proof outline: As we did in [1] we employ the reduction by emulation idea. That is, assume by way of contradiction that there is such a LE algorithm, A. We show how to use A to construct an l-set consensus algorithm, B, among $m > l$ processes that uses only read/write registers. Such an algorithm is impossible by [4, 11, 21]. In this paper we present a new emulation for the reduction which is much more sophisticated and involved than the one we have in [1].

Claim 1 *If there is a LE algorithm* A *among $O(k^{(k^2+3)})$ processes using one* compare&swap-(k) *and any number of atomic registers, then there is a $(k-1)!$-set consensus algorithm* B *for $m = (k-1)!+1$ processes, that uses only atomic registers.*

Proof of Claim: The rest of the paper is the proof of this claim. W.l.o.g. we assume that all atomic registers in A are swmr [3, 17, 19, 22]. Each of the m processes in B is assigned η/m of the processes (front ends) of A. Each process of B emulates each of its assigned front ends in a particular way to be described. Henceforth, the processes of B are called *emulators* and each process of A is called *virtual process* (v-process). The steps of a v-process in A are simulated only by the emulator that owns it. During the emulation, an emulator suspends and releases the emulation of its v-processes depending on the run's progress. However we assign enough v-processes to each emulator so that it will always have a v-process with which it may proceed in the emulation. A detailed formal proof is given after the proof outline (see Figures 3,4,5,6 for the emulation code).

3.1 The emulation

In the emulation process, each emulator iteratively scans the state of the emulation, chooses one of its v-processes and emulates one step of that v-process code in A. Each emulator repeats this process until one of its v-processes reaches a decision state, at which point the emulator adopts that decision value as its output in the set-consensus algorithm B, and leaves the emulation.

In the body of each iteration an emulator does some book-keeping operations and emulates either a read/write operation, or, if the next step of all its v-processes is a c&s() operation, a compare&swap operation for at least one v-process. Read/write operations of the processes in algorithm A are emulated by using such read/write operations on registers shared among the emulators. In emulating a c&s() operation we distinguish between two cases: successful c&s() (that changes the value in the compare&swap) and unsuccessful c&s(). The emulation of unsuccessful c&s() operations is as simple as the emulation of read operations, since such operations do not effect future operations of other v-processes. Successful c&s() operations are emulated by recording in a special data structure, built out of read/write memory, the sequence of changes that have taken place in the compare&swap register; This sequence is called *history*. In each step of the emulation each emulator reads the history and assumes that the last value in the history is the current value of the compare&swap. At this point it may for example simulate a c&s() operation that fail on such a value. However the simulation of successful c&s() operations is much more complicated and will be described in the sequel. In any event, the history (sequence of values that the compare&swap register takes) is the back bone of the constructed run. All the emulated operations are carefully arranged relatively to points in this history.

Another way to view the emulation process is that it is an algorithm by which a set of processes (the emulators) cooperatively construct a legal run of A. In the beginning all the emulators start to build one legal run of A, common to all their v-processes. During the construction the set of m emulators may split into groups such that each group of emulators continues with the construction of a different run of A. The constructed runs of different groups have the same prefix which is their run up to the splitting point.

The construction of such a set of runs of A would be rather simple [1] if each time emulators of the same group emulate a different successful c&s() operation they would each continue to construct a different run of A. Each such run assumes that the compare&swap took a different next_value. In the current proof (as oppose to the one in [1]) groups of emulators split only on the first time a value is used in the compare&swap register. That is, a group of emulators split when some members of the group simultaneously emulate successful c&s() operations that update the compare&swap register with different values non of which has been assigned to the register before. Each such sub-group assumes that a different new_value is appended to the history. We call the sequence of new_values observed by each emulator, its *label*. Therefore there are at most $(k-1)!$ different labels (all the labels start with $\perp$), i.e. at most that many different groups or that many constructed runs. The labels are used to distinguish between different emulated runs. Each group may in the end decide on a different value in the set-consensus (because a different process is elected in A in the different emulated runs).

3.1.1 Emulating successful c&s() operations:

In each iteration an emulator first computes the history and deduces from it the current value in the c&s(), a. Then, only if the next operation of all its active v-processes is a successful c&s() (i.e., c&s($a \rightarrow \cdot$)), it emulates a successful c&s() operation. Among all of these c&s() operations it selects to emulate c&s($a \rightarrow b$) which is the most populous operation, i.e., the one with the largest number of v-processes that have it as their next step. The value b is then added to the history and the operation is emulated as described next.

Before emulating a successful c&s($a \rightarrow b$) operation, each emulator suspends the emulation of many v-processes ($\frac{\eta}{mk^2}$ of them) whose next operation is c&s($a \rightarrow b$). Thus each v-process can be in one of two states: active, or suspended. Once enough v-processes are suspended on different operations we can emulate a successful operation on the compare&swap as follows:

Assume that at least two emulators in the same group have observed the same last value, a, in the history. Let each try to emulate a successful c&s(), one emulates c&s($a \rightarrow b$) and the other emulates c&s($a \rightarrow c$) (assuming that the values a, b, *and* c have already occurred in the history). Then, if there are other v-processes suspended on c&s($b \rightarrow a$) and on c&s($c \rightarrow a$), we may safely assume that both emulators succeed each in the corresponding c&s() by assuming that either of the following history segments occurred: $\ldots abac$ (i.e., c&s($a \rightarrow b$), c&s($b \rightarrow a$), c&s($a \rightarrow c$)) or, $\ldots acab$ (i.e., c&s($a \rightarrow c$), c&s($c \rightarrow a$), c&s($a \rightarrow b$)), where the return to value a in each is due to the release of a suspended v-process. Thus, both operations are successfully emulated in the same run of A without introducing additional splitting. Note that at this point it does not matter which of the two histories is the one emulated as long as each may be provided by the suspended v-processes. At this point, each emulator updates the history data structure about its corresponding successful c&s(). Which of the above two histories is actually emulated depends on where each of them updated the history data structure. In any event, the v-processes that are required to support the occurrence of those sub-histories (one doing c&s($b \rightarrow a$) and the other c&s($c \rightarrow a$)) remain suspended until a future step as explained next.

Even when one of the sub-history sequences is chosen (e.g. $\ldots abac$) the emulators do not release (activate) the corresponding v-processes (those doing c&s($b \rightarrow a$), c&s($a \rightarrow b$), and c&s($a \rightarrow c$), in the example) because several emulators might concurrently decide to release a v-process on the account of the same transition (e.g. $b \rightarrow a$) in the history. We must ensure that exactly one v-process that is suspended on c&s($b \rightarrow a$), is released for this transition.

We overcome this difficulty in the following way: an emulator releases a v-process that is suspended on c&s($b \rightarrow a$) operation only when there are at least m transitions from a to b in the history for which there is yet no corresponding operation in the run. Where an operation is in the run only if a v-process was simulated performing that operation. Then, even if all the m emulators concurrently release a v-process that performs c&s($a \rightarrow b$), there are enough transitions in the history to match all of them. This is a peculiarity of our scheme; the history is constructed while the construction of the corresponding run (the release of the corresponding v-processes) is lagging behind.

In the main iteration step of the emulation, each emulator also has the possibility of doing the following function (called CanRebalance in Figures 3, 5) : for each possible transition in the compare&swap (e.g. $a \rightarrow b$) compute the difference between the number of times that that transition occurs in the history and the total number of v-processes that executed such an operation in the run. If the difference is more than m, and the emulator has one v-process suspended on that transition, that v-process may be reactivated and moved to take a step in the emulated run (which must be a successful c&s()). The exact matching between released v-processes and transitions in the history is not important. All that we prove is that there is a correct matching of all the successful c&s() operations (those that have been released) with transitions in the history. By correct matching we mean that each released v-process could have been doing the corresponding c&s() operation when the transition occur. Thus, in the proof we do not show a specific run of A that was emulated, but rather we prove that there is at least one run of A that the emulation has emulated.

The mechanisms described in the last paragraph are a key difference between the current proof and the one in [1]. Using the proof technique of [1], it is impossible to prove an upper bound on the number of processes that reach election with a compare&swap-(k), due to the fundamental obstacles in that proof.

3.1.2 A more detailed explanation

The implementation of the above idea becomes complicated because there may be more than two emulators in one group each perceiving a different view of the history. Each may try to emulate a different c&s() operation and our algorithm should paste all of their actions together into one legal run. Moreover, while in the toy example above we kept one suspended v-process to close a two edge cycle for each c&s() operation, (e.g. c&s($b \rightarrow a$) closes a cycle with c&s($a \rightarrow b$)), in the algorithm we sometimes keep several such processes that together with the emulated c&s() operation close a longer cycle. To manage these and other activities in the emulation we use two major data structures, (1) the *v-processes graph*, $vp-graph$ and (2) a tree T.

1. *vp-graph:* is a complete directed graph on k nodes, each corresponding to one value from Σ. For each link ($a \rightarrow b$) in the graph each emulator keeps (in single writer multi reader memory) a list of its v-processes that have ever been suspended on a c&s($a \rightarrow b$) operation. When the emulator decides to release a suspended v-process, it does not remove it from the list, but marks it "released". To each v-process in the list we attach a copy of the history as observed by its emulator at the time the process was suspended. Thus each appearance of a particular v-process in the list is with a different history.

2. *T:* is a tree shared data structure in which the history of the emulation is maintained (see Figure 1). Each node of T is a tree by itself. Each of these trees, denoted by t, maintains the history of a group of emulators that simulate the same run of A. Each vertex of a tree t corresponds to a single symbol from Σ in the history. Initially all the emulators are in one group maintaining its history in a tree $t^{\perp}$ located at the root of T. The small trees are shared among all the emulators such that any subset of them may update the tree simultaneously. Moreover, after any set of updates, a sequence of symbols may be derived from the tree, such that this sequence is the history of the intended run. Recall that when a c&s() operation updates the compare&swap register to a value that has not yet occurred in the emulated run, all the emulators that agree on that operation form a group that continue to simulate a run suffix of their own. I.e., the emulators are split into groups according to the new_value each has updated in the compare&swap register. We label each tree t by the sequence of "first_values" that have led the emulators to it, i.e., $t^{\perp}$ is at the root of T, and $t^{\perp 0}, t^{\perp 1}, t^{\perp 2}, \ldots t^{\perp (k-2)}$ are at the second level of T from the root, etc. The sequence of "first_values", which is called *label*, is equivalent to the history after removing from it all the symbols, except the first occurrence of each. Thus, when the emulators concurrently do a successful update of the compare&swap, with different values that have not yet occurred in their run, they

each continue to update the history in a different tree (t). The label of an emulator is the label of the small tree that corresponds to the run that this emulator is currently emulating. The depth of T is at most k and an internal node in depth i has $k-i$ children (each leaf of T corresponds to a different permutation of Σ that starts with $\perp$, i.e. there are $(k-1)!$ leaves).

The history of a run whose emulators label is l is the concatenation of the depth-first-search traversals of all the trees t that are on the path from the root of T to t^l. The detailed description of the structure of each tree t is given in the sequel.

Each emulator starts its iteration by reading all the data structures, from which it computes the most updated history and an *excess-graph*. The *excess-graph* has the same set of nodes and edges as the *vp-graph*. Each of its edges is labeled with the number of v-processes that have ever been suspended on it minus the number of corresponding transitions in the history. That is, the excess on edge $(a \rightarrow b)$ is the number of v-processes suspended on operation c&s$(a \rightarrow b)$ and which have not been matched (consumed) by a corresponding transition in the history.

After reading the state and computing the excess graph the emulator first checks to see if it has $\frac{\eta}{mk^2}$v-processes whose next step is c&s$(a \rightarrow b)$ and it has no v-processes suspended on the corresponding edge in the *vp-graph*. For each such edge the emulator then suspends $\frac{\eta}{mk^2}$v-processes. After that the emulator is ready to advance the emulation of one v-process by one of the following steps: Either

1. emulating a step of a v-process whose next step is not a successful update of the compare&swap, or
2. choosing a v-process that may be suspended in an exchange for a v-process that is already suspended on the same c&s() operation and may be released, or
3. adding a new symbol to the history assuming a transition to this symbol is possible by observing the excess graph.

After performing either of the above the emulator checks if any of its v-processes is ready to decide. If so it stops, otherwise it starts a new iteration.

R/W registers: To facilitate operation (1), in particular the emulation of read and write operations, each register of A is implemented in B by a long list of values. In a write operation we append the new value to the list of all previous values the register has seen (note registers are single writer multi reader). In addition, each value written is tagged by the label of the emulator at the time of the write. In an emulated read operation from register r, the latest value in r whose label is either a prefix or an extension of the reading emulator's label, is returned.

Releasing a successful c&s() operation: As for operation (2) a suspended v-process may be released only after verifying that the c&s() operation it is executing can be safely matched with a transition in the history (as explained in 3.1.1). The matching is computed by listing the entire history against all the v-processes that have ever been suspended, and carefully matching suspended operations to the transitions. (see Figure 5 for the details).

Updating the compare&swap history: Next we describe how an emulator updates the history (operation (3)) in case neither operation (1) nor operation (2) are possible. But first we have to describe the structure of the "small" trees, t's, which are at each node of T.

The structure of tree t: The depth of each tree t^l is at most k, but the degree of each of its internal nodes is P where P is a large constant that depends on the time complexity of A. Essentially each node of t^l corresponds to one symbol in the history and they are pasted together in a depth-first-search order. However, many times the pasting of a symbol to its corresponding parent (or child) in the tree is via a short sequence of values (symbols). Thus, each symbol in each node has two additional fields, *ToParent* and *FromParent* each containing a short sequence of values that the compare&swap register had gone through when it was changed from the node's symbol to the parent's value and vice versa. Moreover, since each of the m emulators may try to update a node in the tree concurrently, we place an m tuple record in each node. Each part of the record is exclusively written by a different emulator (single writer multi reader). All the non-empty parts of a record are considered siblings in the tree, and are treated as such for any purpose. The history of a run whose emulators label is l is the concatenation of the depth-first-search (DFS) traversals of all the trees t that are on the path from the root of T to t^l. The history ends at the right most leaf of the last of these trees. For each edge traversed in the DFS we put in the history the two end nodes, and the path between them as given by the *FromParent* or by the *ToParent* fields, depending on the direction in which the edge is traversed. (see Figure 4). Note that in total the value associated with each node may appear several times in the history due to a single occurrence in the tree.

The difficulty in implementing operation (3), of adding a new value to tree t^l of a run (i.e. to its history), stems from the following scenario: Two emulators

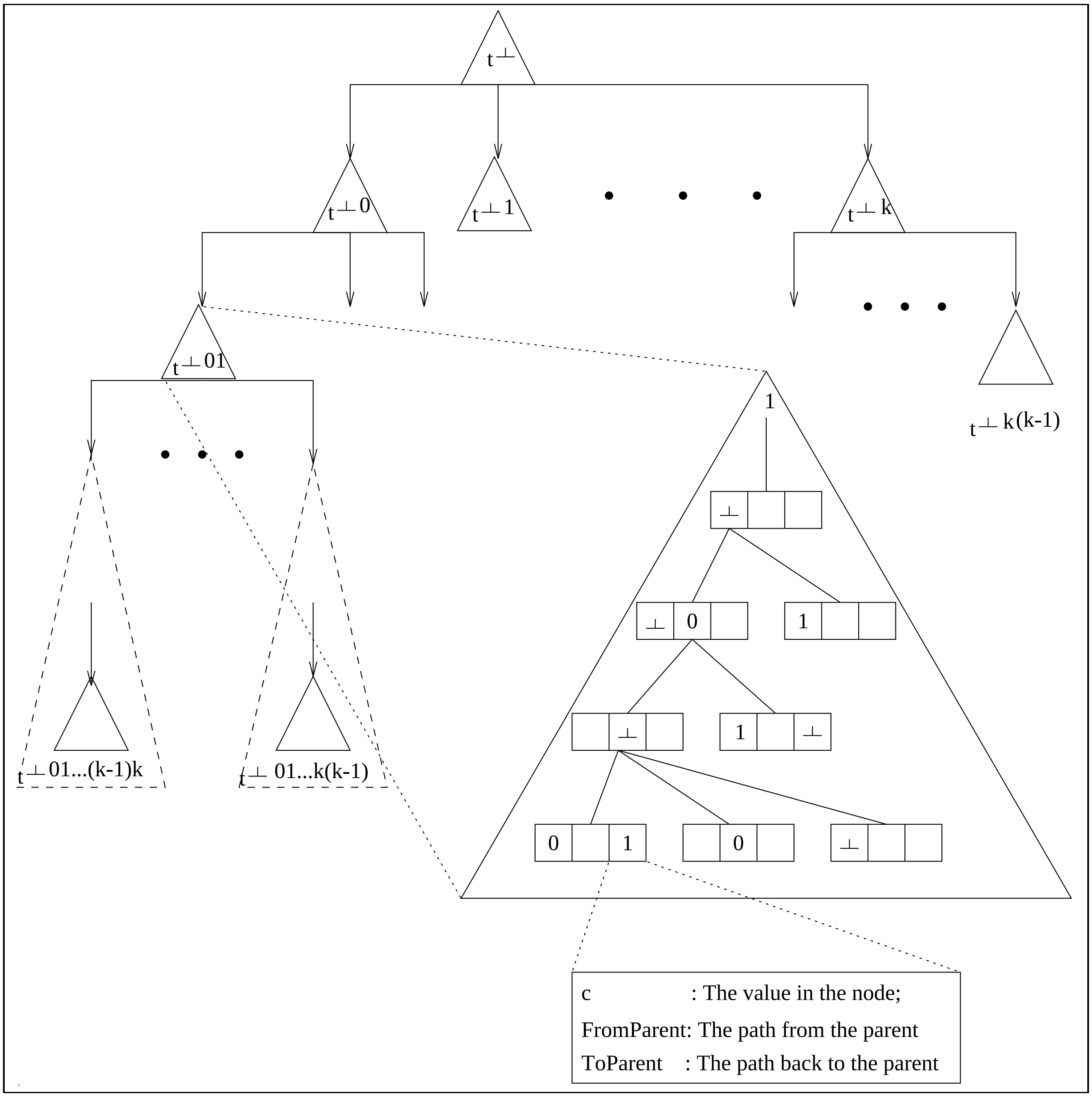

Figure 1: The structure of the Tree T

that simulate the same run may read the history (the trees) at different times, thus assuming different values in the compare&swap, but both may update the history, by updating t^l, concurrently (at the same time). We enable the concurrent updates by proving the following key invariant of each tree t^l:

Invariant: *for each node a in the tree there is a path, in the excess-graph, to each of its ancestors in the tree, such that the excess on each edge on the path is large, (at least m^{j+3}, where j is the ancestor's depth in t^l).*

Now an emulator carries out operation (3) in the following way: Let a be the last symbol in the history read by the emulator. The emulator computes a value b, such that the next c&s() of the greatest number of its active v-processes is c&s($a \rightarrow b$) (hence this emulator must have $\frac{\eta}{mk^2}$v-processes currently suspended on the ($a \rightarrow b$) edge). Then the emulator goes over the ancestors of the node that corresponds to a in t^l, starting from a. It finds the first of these ancestors, f, to which b may be attached as a child while maintaining the key invariant above. It then adds b to the history by attaching it as a child to node f. The resulting history is now as follows, from a we add the sequence that leads to f (by the invariant), and from f we add the sequence that leads from f to b in the excess graph. In the appendix we give the lemma that ensures us that such an ancestor can always be found. The proof of this lemma hinges on the following combinatorial question and its solution:

Combinatorial question: Consider the following process in a complete directed graph on k nodes with m agents that are initially placed in the nodes of the graph. In the process each agent can repeatedly do one of the following two actions:

1. **Move:** in such a step an agent moves from its current node v to some other node u, painting the $v \rightarrow u$ edge.
2. **Jump:** in this step an agent relocates itself in a new node u in the graph. This step is possible only if since the last time the agent has visited node u (or if the agent has never visited node u) another agent has *moved* to u.

The question is then, what is, if any, the maximum number of **moves** the agents can do before the painted edges contain a cycle.

Lemma 1.1 *The solution to the above question is m^k.*

Proof: (Due to Noga Alon) Let G be the graph after the longest possible run that does not contain a painted cycle. Since there is no cycle in the graph we can topologically sort the nodes from $k-1$ to 0 such that painted edges go from high numbered nodes to low numbered nodes.

Associate a weight w_i with each agent that is located in node i such that $w_i = m^j$. Let ϕ_s, be a potential function, equal to the sum of the weights of all agents at state s of the system.

At the beginning of the run, ϕ_0 is at most m^k. It can be easily verified that each **move** of an agent reduces the potential of the system by at least 1 even if all other emulators **jump** upwards as a result of that **move**. Thus the claim follows, since the minimal potential that can be reached before a cycle is closed is 0. ∎

4 Conclusions

This paper resolves the main question that was raised and left open in [1]. Which is, "given a system with unbounded read/write memory and k space complexity compare&swap register, is there a maximum number of processes, n_k, for which this system can solve multi-valued consensus".

In [5] Burns Cruz and Loui proved that a compare&swap-(k) alone can elect a leader between at most $k-1$ processes. An immediate conclusion from the results in [5] and here, is that adding read/write registers to the compare&swap register increases its power. Here we proved that these increase is exponentially limited.

We believe that the results presented herein can be extended to hold for arbitrary read-modify-write registers of size k, and to systems with a number of copies of the strong object.

Although we managed to prove here that n_k is bounded by $O(k^{(k^2+3)})$ there is still a gap between that and the $(k-1)!$ processes algorithm we have presented in [1]. Our conjecture is that $n_k = \Theta(k!)$).

Acknowledgments: We thank Noga Alon for the proof of Lemma 1.1. We also thank Yishay Mansour, Manor Mendel, and Michael Saks for helpful discussions.

References

[1] Y. Afek and G. Stupp. Synchronization power depends on the register size. In *Proc. of the 34th IEEE Ann. Symp. on Foundation of Computer Science*, pages 196–205. IEEE Computer Society Press, November 1993.

[2] B. N. Bershad. Practical considerations for lock-free concurrent objects. Technical Report CMU-CS-91-183, Carnegie Mellon University, September 1991.

[3] B. Bloom. Constructing two-writer atomic registers. In *Proc. of the 6th ACM Symp. on Principles of Distributed Computing*, pages 249–259, 1987.

[4] 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.

[5] J. E. Burns, R. I. Cruz, and M. C. Loui. Generalized agreement between concurrent fail-stop processes. In A. Schiper, editor, *Proc. of the 7th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 725*, pages 84–98. Springer Verlag, Berlin, September 1993.

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

[13] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 291–302, 1988.

[14] P. Jayanti. On the robustness of Herlihy's hierarchy. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pages 145–158, August 1993.

[15] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proc. of the 6th Int. Workshop on Distributed Algorithms: Lecture Notes in Computer Science, 647*, pages 69–84. Springer Verlag, November 1992.

[16] J. M. Kleinberg and S. Mullainathan. Resource bounds and combinations of consensus objects. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pages 133–144, August 1993.

[17] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[18] 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.

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

[20] 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.

[21] 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.

[22] A. K. Singh, J. H. Anderson, and M. G. Gouda. The elusive atomic register revisited. In *Proc. of the 6th ACM Symp. on Principles of Distributed Computing*, pages 206–221, 1987.

A Correctness Proof of the Emulation

We now show that the emulation is correct, that is, that it implements a $(k-1)!$-set consensus among $(k-1)!+1$ emulators. Let s_i be a global state of the emulation system including all emulators and shared data structures. Let $E = s_0\pi_1 s_1 \pi_2 s_2 \ldots$ be an execution of the emulation B where each π_i is either an internal operation or an external operation of one of the emulators. Let $R = \pi_1\pi_2\ldots$ be the corresponding run of the execution E. Let $R_k = \pi_1 \ldots \pi_k$ be a length k prefix of R. For any operation π_i in R, let l_{π_i} be the value of the label of the emulator executing π_i (in state s_{i-1}). The string l_{π_i} is called *maximal label* if there is no π_j s.t. l_{π_i} is a prefix of l_{π_j} in R. We denote such a label l^*.

Some of the operations in every run 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 A are called *virtual operations*. There are several 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, $\pi = (\text{write}_x(l, v))$, where l is a label, that correspond to a write operation in the front end of the virtual process, $\text{write}_x(v)$.

3. *History operations.* There are three kinds of history operations: a *history read*, $\pi = ((T', G'){:=}\text{SnapShot}(T, G))$, which is executed in the main routine (Emulation), *history update*, π =(attach new son to node), which is executed in the update routine (UpdateC&S) and a *history response*, π =(return result to c&s() operation), which is executed in the EmulateSimpleOp routine, the rebalancing routine (CanRebalance) and the update routine (UpdateC&S). Usually, the triplet of history read, history update and history response correspond to one c&s() operation. Sometimes, many history responses are associated with one pair of history read and history update to emulate many c&s() operations that occur one after the other.

While only virtual operations are mapped to operations of A, we are also interested (for the sake of the induction) in the next two operations.

1. *Suspend* a virtual process, $\pi = (suspend)$, by which an emulator marks a virtual processes as inactive. A virtual processes is suspended just before executing a c&s() operation. Operations of this virtual processes will not be emulated until it is activated again.

2. *Activate* a virtual process, $\pi = (activate)$. An emulator activates a virtual processes only when it is safe to assume the c&s() operation of the virtual process has succeeded. Note that the label, l_π of this operation is the label saved when the virtual process was suspended.

Let $R|_l{=}\pi_1\pi_2\ldots$ be a subsequence of R such that every operation π_i is a virtual operation, a Suspend or an Activate operation, and for each π_i, l_{π_i} in R is a prefix of l. We define $\rho^{R|_l}$ to be a run of A that corresponds to $R|_l$. That is, every virtual operation π of $R|_l$ is mapped to its corresponding operation(s) in A (the exact mapping is described in the proof of lemma 1.2). From the next lemma it follows that all emulators deciding in the same emulated run, $R|_{l^*}$, decide on the same value. Since there are at most $(k-1)!$ possible different maximal labels and since every non faulty emulator decides in some $\rho^{R|_{l^*}}$, the emulation implements $(k-1)!$-set consensus.

Definition 1 *for each state s in the execution of the emulation* B *we define:*

- *Let $s^s_{(a\to b)}$ be the number of successful c&s$(a \to b)$ operations that were emulated in the run up to state s.*

- *Let $p^s_{(a\to b)}$ be the number of transitions from a to b that are written in the history at state s.*

- *Let $d^s_{(a\to b)} = (p^s_{(a\to b)} - s^s_{(a\to b)})$; That is, the unmatched transitions in the history from a to b.*

- *Let $f^s_{(a\to b)}$ be the number of virtual processes of all emulators that have been suspended and not yet released, and whose next operation is a c&s$(a \to b)$.*

- *Let $w^s_{(a\to b)} = (f^s_{(a\to b)} - d^s_{(a\to b)})$. That is, the number of suspended virtual processes that are yet not demanded by the history and thus can be used in future transitions.*

- *Let G^s be the viable excess graph* [2] *at state s. That is, a directed weighted graph where the weight of each edge (a,b) is $w^s_{(a\to b)}$.*

- *Let G^s_x be a graph obtained from the excess graph by removing all edges whose weight is smaller than x.*

- *Let C_x be a maximal strongly connected component in G^s_x.*

Let $\gamma_1 = 0$ and $\gamma_x = \sum_{i=2}^{x} m^i$ where m is the number of emulators.

Definition 2 *A stable component, SC, of the excess graph G_x is a C_1 component of G_1 where, if $|C_1| = j$ then for all $k - j + 2 \le i \le k$ SC can be partitioned to at most $i - (k - j + 1)$ maximal components $C_{\gamma_{(k-j+i)}}$. A single node ($j = 1$) is also a stable component.*

Lemma 1.2 *For any run R of* B *and any l^*,*

1. *$\rho^{R|_{l^*}}$ is a legal run of* A.

2. *The value in the c&s() after $\rho^{R|_{l^*}}$ is the last value in $h(R|_{l^*})$, the history as computed by the algorithm (ComputeHistory) at the final state of $R|_{l^*}$. More ever, $h(R|_{l^*})$ is the list of changes to the* compare&swap*that occurred during the run.*

[2] In Lemma 1.3 we actually consider a graph induced by a subset of the nodes. This subset is, informally, the set of values in l.

3. *The excess graph G_x, induced by values that have already showed up in the history $h(R|_{l^*})$ can be described as a group of 0 or more stable set components connected by a one way path of weight γ_k or more.*

Proof:

Obviously what we really want to prove is the first point. We will use the second point of the lemma during the induction to explain why the c&s() operations we emulate succeed or fail in $\rho^{R|_l}$ according to what we anticipate in the emulation. We use the third part of the lemma to explain why for every possible emulated run there are always enough suspended v-processes to explain all the transitions in the compare&swap.

The proof proceeds by induction on $R = \pi_1\pi_2\ldots\pi_k$, a prefix run of B.

For $R_0 = \emptyset$ the claim trivially holds.

Assume the claim holds for R_k, we will prove the claim for $R_{k+1} = R_k\pi_{k+1}$.

If π_{k+1} is not a virtual operation or a Suspend or an Activate then $R_{k+1}|_l = R_k|_l$ for any l. Furthermore, if for some l^* (maximal label in R_k), $l_{\pi_{k+1}}$ is not a prefix of l^* then, by definition, $R_{k+1}|_{l^*} = R_k|_{l^*}$.

Thus, it remains to prove the inductive step for the cases in which for some maximal label of R_k, l_1^*, $l_{\pi_{k+1}}$ is a prefix of l_1^*.

We begin the proof by considering the cases where π_{k+1} corresponds either to a virtual read operation or to a virtual write operation.

Since $\rho^{R|_{l_1^*}}$ is legal, the invocation part of π_{k+1} is legal (because it is decided by the emulator using the same protocol that the real processes use in B). For example, if π_{k+1} is a read operation on shared variable x then there is a run $\rho^{R_{k+1}|_{l_1^*}} = \pi'_1\ldots\pi'_{k+1}$ such that π'_{k+1} is an emulated read operation of the shared variable x. It remains to argue that the response part is correct. We consider two different cases according to the type of π'_{k+1}.

1. If π_{k+1} is a write operation. Then the response is always the same (empty) and so is immediately correct.

2. If π_{k+1} is a read operation on variable x then from the emulator's protocol, we know that the response to π_{k+1} is chosen to be the value written to x by some operation π_j, $1 \leq j \leq k$ where either l_{π_j} is a prefix of $l_{\pi_{k+1}}$ or vice versa. Furthermore, the largest such l is chosen. As x is a swmr shared register the histories of consecutive writes to x are extensions of each other. The fact that we chose the largest l means that for any maximal history l^* in R_k, if l_{π_j} is a prefix of l^* then π'_j was the last write to x in $\rho^{R_k|_{l^*}}$.

   (a) Assume l_{π_j} is a prefix of $l_{\pi_{k+1}}$. As $l_{\pi_{k+1}}$ is a prefix of l_1^*, so is l_{π_j} which means that $\pi_j \in R|_{l_1^*}$. From the induction $\rho^{R|_{l_1^*}}$ is legal and so the value written by π'_j is legal. As it is the last value written to x in $\rho^{R|_{l_1^*}}$ it is a legal response to π_{k+1}. This means that $\rho^{R_{k+1}|_{l_1^*}}$ is legal or that point 1 of the claim is correct.

   Points 2 and 3 of the lemma are trivially correct since neither l_1^* (in $R_{k+1}|_{l_1^*}$) nor the value of the compare&swap (in $\rho^{R_{k+1}|_{l_1^*}}$) has changed.

   (b) Assume $l_{\pi_{k+1}}$ is a prefix of l_{π_j}. If l_{π_j} is a prefix of l_1^* the case is identical to 2a. Thus, we assume l_{π_j} is not a prefix of l_1^*. Note that the opposite, that is that l_1^* is a prefix of l_{π_j}, is impossible because l_1^* is maximal in R_k. From the definition of the emulation, the emulator executing π_{k+1} changes the value in its label to l_{π_j}. This means that all the following shared operations of that emulator will not be part of any $R_n|_{l_1^*}$ where $R_n = R_{k+1}\pi_{k+2}\ldots\pi_n$. Such a condition corresponds to a fail-stop in A of all the processes that the emulator emulates immediately after π'_{k+1}. As such a condition is legal, the run $\rho^{R_{k+1}|l_1^*}$ is legal and so point 1 of the claim holds. Note that it does not matter what the response of π_{k+1} was because in any $\rho^{R_n|_{l_1^*}}$ the process emulating π'_{k+1} fail-stopped and, π'_{k+1} being a read operation, its response does not effect any extension of $\rho^{R_{k+1}|_{l_1^*}}$. Although the value in the history variable of the emulator executing π_{k+1} has changed, l_1^* did not change and so (as the compare&swap did not change) points 2 and 3 of the lemma also hold.

   Two points are of interest. One is that although for any run that is defined by l^* the process p_i that executed π'_{k+1} has fail-stopped, the emulation of v_i continues (but in a different run, one that corresponds to, say, l_2^*). The second is that there might be an "illegal" run where *all* processes p_i in some run $\rho^{R|_{l^*}}$ fail-stopped. But, as the emulations continue in other runs this case does not influence the integrity of the emulation.

Now, we consider the case of history operations. Basically, when a c&s($a \rightarrow b$) is emulated the emulator first executes a history read, π_r, to decide what is the current value of the compare&swap (the last symbol in $h(R|_{l^*})$), then it executes a history update, π_u, to update the compare&swap and finally, a history response, π_s, is executed to emulate the return value of the c&s() operation. However, there are cases where the history update is not executed (when the c&s() has failed) and cases where one history read matches several history responses (when several c&s() operations are emulated one after the other).

Assume $\pi_{k+1} = \pi_r$ a history read. A history read by itself is not mapped to an operation in A, nor does it effect G_x. However, it might change the emulator's label. When ever an emulator calls ComputeHistory its label might, as a side effect, change. This happens if t^l is no longer a leaf in T. In such a case, l is extended until t^l is a leaf in T. If the new label is a prefix of l_1^* then little has changed. If the new label is not a prefix of l_1^* then all active virtual processes of the emulator are assumed to fail stop at this point of the run, which is legal. Note that :

- From point 2 of the claim we know that $h(R_k|_{l^*})$ is correct.
- The label of the Activation operation is the label that was written at the time of suspension and so such operations can still participate in the run.

In any case, since no mapping is done nor does G_x change, the lemma is correct by induction.

Assume $\pi_{k+1} = \pi_u$ a history update, where π_r is the corresponding history read and $\rho^{R_k|_{l^*}} = \pi'_1 \dots \pi'_{k'}$. History updates can occur in two contexts. One, if a value that has not yet been written to the compare&swap is written (line 12 of UpdateC&S) and the other if a value that has been used is written (line 9of UpdateC&S).

1. In the case of a new value, A new node of T, $t^{\perp a_1 \dots a_{i-1} a_i}$ is marked as active (by setting a bit). If, between π_r and π_u another emulator marked the new node as active then no mapping is needed. Assume otherwise. Let v_{p_1} be the rightmost leaf in $t^{\perp a_1 \dots a_{i-1}}$, $v_{p_2} \dots v_{p_j}$ the nodes in the path from v_{p_1} to the root. Let $x_1 \dots x_r$ be the concatenation : v_{p_1}.ToParent$||v_{p_2}$.c$||v_{p_2}$.ToParent$|| \dots ||v_{p_j}$.c (if v_{p_j} is the root of then this string is empty). If $\rho(R_k|_{l^*}) = \pi'_1 \dots \pi'_{k'}$ then we add at the end of the run
 the following operations : $\pi'_{k'+1}$ =c&s(v_{p_1}.c $\rightarrow x_1$), $\pi'_{k'+2}$ =c&s($x_1 \rightarrow x_2$),…, $\pi'_{k'+r-1}$ =c&s($x_{r-1} \rightarrow x_r$). Since $x_r = a_{i-1}$ we now need to add c&s() operations from a_{i-1} to a_i (which are mapped just as the previous are). Indeed, these transitions are not represented by the history. But, as there are at most k such cases for each kind of transition, there will always be enough suspended virtual processes to accommodate the transition.

   If no mapping is done then the claim is correct by induction. Assume the mapping was needed.

   (a) From point 2 of the claim we know that x_1 was the last value in the compare&swap. This means that after the mapping the value in the compare&swap will be a_i which is the last value in $h(R'_{k'+j}|_{l^*})$, which means point 2 is correct.

   (b) A new node was added to the chain of stable components, which is a stable component of size 1. The weight of the path is at least γ_k since the emulator suspends at least as much and they were never used in the history up to now. Note that the stable sets stays stable since we use at most one suspended virtual process on each edge of the last stable component.

   (c) To prove correctness of $\rho^{R_{k+1}|_{l^*}}$ we note that 1) $x_1, \dots, x_j, a_i$ are in the same SC and so there are suspended virtual processes to accommodate all the transitions and 2) γ_k virtual processes were suspended on c&s($a_{i-1} \rightarrow a_i$). This means ρ is legal.

2. If a previously used value is emulated to be written to the c&s(), then a new node is attached to the currently active tree, $t^{\perp a_1 \dots a_i}$. If the new node, v_i is the rightmost leaf (as is the case if no other emulator emulated an update since π_r) then the mapping is as follows: Assume v_{p_1} was the rightmost leaf before v_i was attached, $v_{p_2}, v_{p_3}, \dots, v_{p_j}$ the nodes in the path between v_{p_1} and the parent of v_i (it is possible that v_{p_j} is the parent in which case the list is empty). Let $x_1, x_2, \dots, x_r$ be the concatenation v_{p_1}.ToParent$||v_{p_2}$.c$||v_{p_2}$.ToParent$|| \dots ||v_{p_j}$.c (if v_{p_j} is the parent of v_i then this string is empty). If $\rho(R_k|_{l^*}) = \pi'_1 \dots \pi'_{k'}$ then we add at the end of the run
 the following operations : $\pi'_{k'+1}$ =c&s(v_{p_1}.c $\rightarrow x_1$), $\pi'_{k'+2}$ =c&s($x_1 \rightarrow x_2$),…, $\pi'_{k'+r}$ =c&s($x_r \rightarrow v_i$.c).

   If the new node, v_i, is not the rightmost node then let v_{p_j} be its parent and p_{j-1} be the node that is before v_{p_j} in a DFS order of the tree $t^{\perp a_1 \dots a_i}$ where each node is counted each time it is traversed. Assume $\rho(R_k|_{l^*}) = \pi'_1 \dots \pi'_{k'}$ and let π'_o be the c&s($x \rightarrow v_{p_i}$.c) operation that was mapped as a result of the transition from $v_{p_{j-1}}$.c to v_{p+j}.c. Let $x_1 \dots x_r$ be the concatenation v_{p_i}.FromParent$||v_{p_i}$.c$||v_{p_i}$.ToParent. We map the cycle of changes right after π'_o by inserting it before π'_{o+1}.

   We prove correctness of the claim for each case separately:

   (a) In the first case, we map operations of virtual processes that are suspended to the end of the previous run. We need to show that there were enough suspended virtual processes to accommodate for the new csz operations.

This is shown by lemma 1.3. Again note that we might later choose another virtual process to execute the operation and switch between the virtual processes. Such an operation is legal if both virtual processes did not as yet execute a history response. As for point 2 of the claim, we know from the assumption that the value in compare&swap after $\rho(R_k|_{l^*})$ was v_{p_j}.c. This means all the c&s() operations we mapped have succeeded and so the history still represents the list of changes to the compare&swap and v_{p_i}.c is the value in the compare&swap after $\rho(R_{k+1}|_{l^*})$. Point 3 is proven in lemma 1.3.

(b) As for the second case, to prove part 1 of the claim we need to show that by inserting the cycle in the middle of $\rho(R_k|_{l^*})$ we do not make the operations after it illegal. But, since it is a cycle and since there are no other operations of the processes that executed that cycle in $\rho(R_{k+1}|_{l^*})$ after the cycle, the addition is transparent to the run. Other then this, the proof of the claim is exactly as the previous case.

Assume $\pi_{k+1} = \pi_s$ a history response, where π_r is the corresponding history read and (if exists) π_u the corresponding history update. History responses can occur in several places of the algorithm and with several connotations. We will examine each case separately.

1. First, we examine the history response at line 9 of CanRebalance. While this operation is not mapped to a new operation of A, we may change a c&s() operation in $\rho^{R_k|_{l^*}}$ so that this process will execute it instead of the one that is currently executing it. This happens because while executing the history update we did not care which of the suspended virtual processes actually executed the operation and so chose one of the possible. However, now we need a specific virtual processes (the one which receives the history response) to succeed and so we switch processes. The new mapping is executed by taking all successful responses in $R_{k+1}|_{l^*}$ and map each one to a corresponding c&s() operation in $\rho^{R_{k+1}|_{l^*}}$ so that each occurs between the time the virtual process was suspended and the time the successful response occurred.

   To prove point 1 of the claim we need to show that such a mapping exists. Before an emulator decides to execute a successful history response it first checks if, using this mapping, there are at least m (number of emulators) places to map the history response to. That is, if the history response is for a c&s($a \rightarrow b$) operation, returning a, then after all successful history responses are mapped to all transitions in the history, there should still be m transitions from a to b in the history. Since every emulator executes this procedure, even if all emulators decide together to return a successful response for a c&s($a \rightarrow b$) there will be enough transitions in the history to accommodate them. From this follows the correctness for point 1. Points 2 and 3 of the claim are immidiatly correct since the operation does not change the history nor the graph G_x.

2. The next case of a history response is at line 15 of UpdateC&S. In this case, failed responses (x) are returned to all active virtual processes of an emulator. This happens only after a history update was executed, to value x. Assume $R_k|_{l^*} = \pi'_1 \ldots \pi'_{k'}$, π'_o the c&s() operation that was mapped as a result of the history update. We map the c&s() operation that failed (returning x) immidiatly after π'_o by inserting the operation before π'_{o+1}.

   Since $\rho^{R_k|_{l^*}}$ was legal, the value in the compare&swap after π'_o was x and so the response is legal. Since failed c&s() operations do not effect any processes but the one executing it and since there are no more operations of that processes in $\rho^{R_k|_{l^*}}$ after π'_o, $\rho^{R_{k+1}|_{l^*}}$ is legal. Again, points 2 and 3 of the claim are immediate.

3. The case of failed c&s() operations (in EmulateSimpleOp) is exactly the same as the previous case. The only difference is that π'_o is an operation mapped due to a history update of another emulator.

Last, we consider the cases of Suspend and Activate operations.

Assume π_{k+1} is a Suspend operation to virtual process v_i which was about to execute the operation c&s($a \rightarrow b$).

- Since Suspend operation is not mapped to operations in A$\rho^{R_{k+1}|_{l^*}} = \rho^{R_k|_{l^*}}$ and, from the assumption it is legal.
- Since Suspend operation does not change the history part two of the claim is trivial.
- If b is part of the stable set of a its ok. Assume b is in another stable components. Since there is a path of weight γ_k between the stable components, and since at least γ_k vp are frozen together, there is a cycle of γ_k weight between the SC's. Assume the size of the biggest SC was s_1. Then, for all $1 < i \leq s_1$ there are at most $i-1$ $c_{\gamma_{k-s_1+i}}$ meaning there are at most:

  1 component of size $C_{\gamma_{k-s_1+1}}$

2 components	of size	$C_{\gamma_{k-s_1+2}}$
$s_1 - 1$ components	of size	$C_{\gamma_{k-s_1+1}}$

Looking at lemma 1.3 we see that for all $2 \leq i \leq (s_1 - 1)$ there are at most $i - 1$ components $C_{\gamma_{(k-s_1+i-1)}}$ from which immediately follows that the lemma holds. Also, there are at most $s_1 - 1$ C_{γ_k} components which means that for all $(s_1 - 1) \leq i$ the lemma is correct.

Assume π_{k+1} is an Activate operation to a virtual process v_i which has executed operation c&s($a \rightarrow b$). Since activations are not mapped points 1 and 2 of the claim are correct by induction. Activating a virtual process decreases f^s but increases s^s thus not changing w^s or the excess graph G_x, so point 3 too is correct by induction

∎

Definition 3 *A super stable component, SSC, of the excess graph G_x is a C_1 component of G_1 where, if $|C_1| = j$ then for all $k - j + 3 < i \leq k$ SSC can be partitioned to at most $i - (k - j + 2)$ maximal components $C_{\gamma_{(k-j+i)}}$. A C_1 component of two nodes ($j = 2$) is always a super stable component.*

Lemma 1.3 *If at state s of the emulation C is a super stable component then in any run of the algorithm from state s which includes only history updates of values in C , C stays a stable component.*

From the algorithm we know that an update operation c&s($a \rightarrow b$) of emulator e occurs only after the emulator first suspends γ_k virtual processes that are about to execute c&s($a \rightarrow b$) and rebalances all virtual processes it can. This means that if the run could have been linearized so that an emulator atomically reads the history, decides which value to update and updates it then will always be suspended virtual processes to accommodate each transaction in the history. In fact, even if the run was asynchronous but we could guarantee that transitions in the history occurs only as direct update operation of an emulator (that is, that a transition aa_1 in the history occurred because some emulator emulated a successful c&s($a \rightarrow a_1$) operation) there wouldn't be a problem since γ_k the number of suspended operations on each such transition, is bigger than m (the number of emulators) and before any emulator can re-execute the same update it must refill the γ_k suspended. Lets look at the following run. Assume the current history is $\perp$. Assume e_1 decides to update the history to $\perp a$ (suspending γ_k virtual operations that are about to execute c&s($\perp \rightarrow a$) just before) and e_2 decides to update the history to $\perp 1$ (again, suspending γ_k virtual processes). Since both must continue in the same run, we have to choose a run that will accommodate both. Assume we choose the history $\perp 0 \perp 1$ (the second $\perp$ is not needed in this part of the proof but is here for the sake of completeness). Assume that after the operation one of the emulators (say e_1) updated the history back to $\perp$ (suspending γ_k virtual processes that are about to execute c&s($\perp \rightarrow 1$)) and the same run continues. Our history will be $\perp 0 \perp 1 \perp 0 \perp \ldots$ but we suspend new virtual processes only on c&s($\perp \rightarrow 0$), c&s($\perp \rightarrow 1$) and c&s($1 \rightarrow \perp$). This means we continuously use previously suspended virtual processes of the operation c&s($0 \rightarrow \perp$) without suspending new ones!

To solve this problem, we use the structure of the tree t.

Claim 2 *An update operation in C_{γ_x} will be mapped in the history before an operation in $C_{\gamma_{x-1}}$ unless there was another operation in C_{γ_x} after the first.*

The claim means that when the repository of the c&s($0 \rightarrow \perp$) will be depleted enough to make 0 a different component in $C_{\gamma_{x-1}}$ the order in the history will change so if e_1 is about to update the history to $\perp 0$ e_1 can change the history many times form $\perp$ to 1 and still the merged history will be $\perp 1 \perp 1 \perp 1 \perp 0$. Indeed, we didn't as yet refill the the suspended virtual processes that are about to execute c&s($0 \rightarrow \perp$) but in order to repeat the previous scenario we will have to return the history to $\perp$ thus refilling the path. The need for a full tree arises from the fact that there might be several emulators concurrently emulating a move to different components in $C_{\gamma_{x-1}}$. Then, only the last (whichever it is) will be refilled in the scenario as above. When the other edges will be further depleted, the move to them will come after the move to the first due to the tree structure.

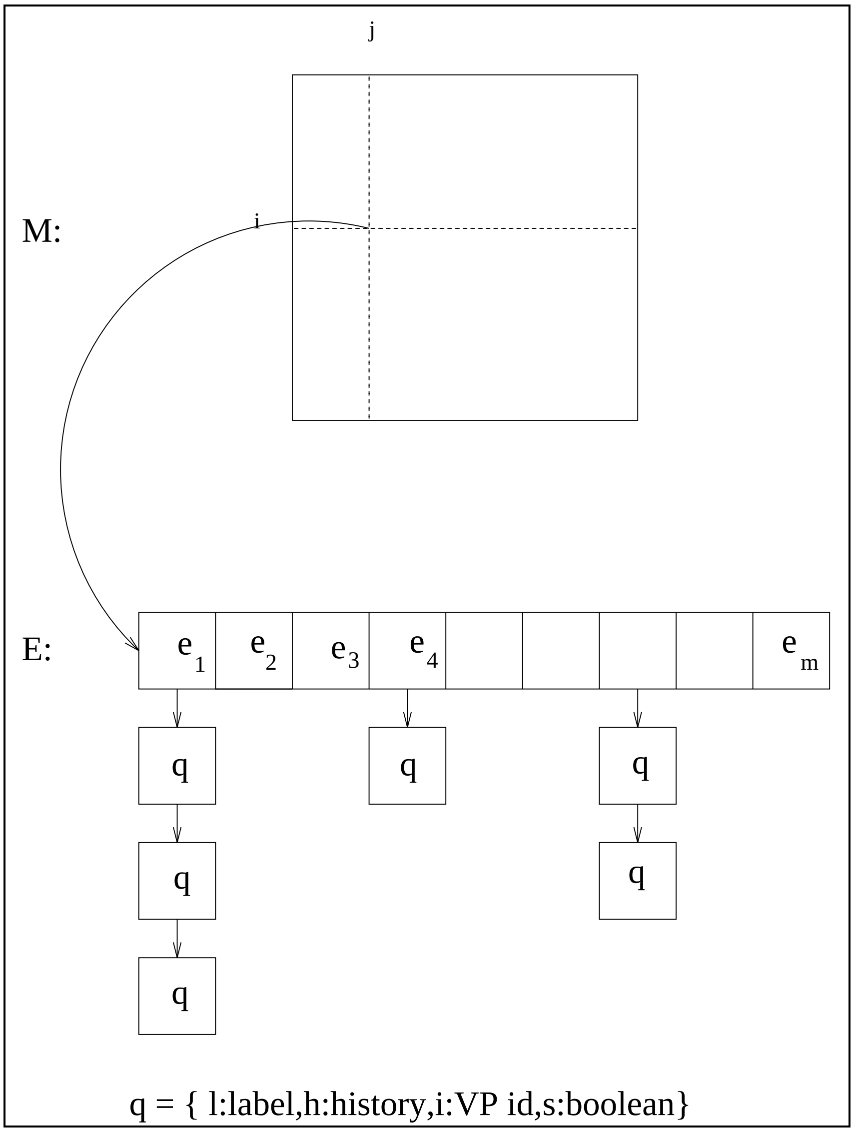

Figure 2: The data structure representing the vp-graph G

Emulation(j:id) **returns(decision value)** — Code for emulator E_j

shared:

- T tree; — See text for a description of T.
- G Graph; — The vp-graph as represented by the matrix M,E (see Figure 2).

local:

- T',G' Local counterparts of T, G;
- h' list initialized to $\{\perp\}$; — h' represents the history of changes made in the compare&swap, as viewed by E_j.
- l' list initialized to $\{\perp\}$; — l' represents the label that uniquely identifies the run in which E_j is participating.
- cs symbol initialized to $\perp$; — The cs holds what E_j believes to be the current value in the compare&swap (last symbol of h').
- V_j The set of virtual processes assigned to E_j;

code:

1 **while** no virtual process decided **do**
  begin
2 (T', G') :=SnapShot(T, G); — Atomically read all shared memory data.
3 h':= **ComputeHistory**(T', l', cs); — ComputeHistory computes from T' and l' the history of changes in the compare&swap. The variable cs gets the last symbol in h'.
4 **if** there are $\frac{\eta}{mk^2}$active virtual processes in V_j whose next step is c&s$(a \rightarrow b)$, and there are no virtual processes of V_j that are suspended on c&s$(a \rightarrow b)$
5 **then** suspend $\frac{\eta}{mk^2}$virtual processes from those found in step 4 (by appending new entries to $M[a,b]->E[j]$). — This line is executed at most once for each pair a, b.
6 **if** there is some active virtual process in V_j whose next operation is neither a c&s() operation nor a c&s$(a \rightarrow b)$ s.t. $cs \neq a$
7 **then EmulateSimpleOp**$(\ldots)$; — EmulateSimpleOp emulates operations that do not require updates to the compare&swap.
  else
8 **if CanRebalance**(G', h', l') **then** do nothing;
9 **else UpdateC&S**(T', G', h', l');
  end ;
10 **return** (the value decided by one of the virtual processes);

Figure 3: Code of the emulation process for emulator E_j

ComputeHistory(T':tree,l':label,cs:symbol) returns(list,label)

let NewL be the longest path from the root of T' such that l' is a prefix of that path;
l' :=NewL;
x :=””;
let x' be the first symbol in l' which corresponds to the first vertex of T;
repeat — DFS of one small tree
  x:=$x||x'$;
  if $x \neq l'$ **then**
    let h' be the concatenation of nodes of t^x in DFS order so that:
      if we enter a node, w from its parent **then** h':=$h'||w.FromParent||w.c$;
      if we return to a node w from its son **then** h':=$h'||w.c$;
      if we leave a node w back to its parent **then** h':=$h'||w.ToParent$;
  else — This is the last node on the path l' in T, $t^{l'}$.
  begin
  let cs be the rightmost leaf of $t^{l'}$;
  Compute h' in the same way as above, but only up to (and including) cs.
  end ;
  h:= $h||h'$;
  x':= the next symbol of l';
until $x = l'$;
return h;

This routine computes the history of changes to the compare&swap given a state T' of T. The computed history is composed only from symbols that are in l'. In the last tree we concatenate only its sequence up to the right most leaf. Otherwise, all our histories would end at the root of t^l.

Figure 4: Code of the ComputeHistory subroutine

CanRebalance(G':vp-graph,h':history,l':list) returns(boolean)

1 **let** $v_{i_1}, \ldots, v_{i_p}$ be the list of suspended virtual processes of E_j sorted in ascending order by the place in the history that the virtual process has been suspended ($p = \frac{\eta}{mk^2}$).

2 **let** U be the set of all successful c&s() operations such that the label associated with each is a prefix of l' (i.e. all of those for which $M[x, y].E_i.q.l$ is a prefix of l').

3 We match as many operations in U as possible, to transitions in the history. Each operation c&s($a \rightarrow b$) may be matched to a transition in the history from a to b that occurred after the corresponding virtual process was suspended.

4 **for** $x := 1$ to p **do**

  begin

5   Assume the operation that v_{i_x} is about to execute is c&s($a \rightarrow b$).

6   **if** (1) there are m or more unmatched transitions in h' from a to b, and (2) v_{i_x} was suspended before their occurrence, and (3) there is at least one active virtual process of E_j, v_q whose next operation is c&s($a \rightarrow b$)

  then

    begin

7   mark v_q as suspended by appending a new entry to the list $M[a, b]->E[j]$;

8   reactivate v_{i_x} by changing the status bit, s, of the corresponding entry in the suspended list to true;

9   emulate one step of v_{i_x}, by returning a as the result of the c&s() operation;

10   **return** (True);

    end ;

  end ; Of for loop.

11   **return** (False);

Figure 5: Code for CanRebalance

1 **UpdateC&S(G':vp-graph,h':history,T',cs)**
local:
2 G : full directed graph of $\{\perp, \ldots, k-2\}$ nodes;
code:
First, we compute the excess graph G.
3 **for** every pair $a, b \in \{\perp, \ldots, k-2\}$ **do**
Compute the weight (excess) of edge (a, b),$w_{(a \to b)}$, following the definition of $w^s_{(a \to b)}$. I.e., the number of globally suspended virtual processes whose next operation is c&s($a \to b$), plus all the previously successful c&s($a \to b$) operations (by counting the number of q blocks in $M[a, b]$), minus the number of transitions from a to b in h';
4 **let** CurrentParent be the node containing cs;
5 Assume most of the active virtual processes of v_j are about to execute c&s($cs \to x$);
6 **let** w be the width of the cycle whose minimum excess is the largest in G, and such that both CurrentParent and x are on that cycle;
7 **let** D be the distance of CurrentParent from the root;
8 Threshold :=$\sum_{g=1}^{D} gm^g$;
9 **if** $w \geq$ Threshold attach x to CurrentParent;
else
begin
10 CurrentParent:= parent of CurrentParent;
11 **if** CurrentParent $= \emptyset$ — We are at the root.
12 **then** mark tree $t^{a_1 \ldots a_l x}$ as active; — x must have never been used in the emulated run.
13 **else goto** 6
14 **end** ;
15 Emulate one step of all the active virtual process in V_j by failing their corresponding c&s() operation, and returning x to each;
16 **return** ();

Figure 6: Code for UPDATEC&S