An Adaptive Technique for Constructing Robust and High-Throughput Shared Objects

Danny Hendler[1], Shay Kutten[2], and Erez Michalak[2]

[1] Department of Computer-Science, Ben-Gurion University
[2] Department of Industrial Engineering and Management, Technion

Abstract. Shared counters are the key to solving a variety of coordination problems on multiprocessor machines, such as barrier synchronization and index distribution. It is desired that they, like shared objects in general, be robust, linearizable and scalable.
We present the first linearizable and wait-free shared counter algorithm that achieves high throughput without a-priori knowledge about the system's level of asynchrony. Our algorithm can be easily adapted to any other combinable objects as well, such as stacks and queues.
In particular, in an N-process execution E, our algorithm achieves high throughput of $\Omega(\frac{N}{\phi_E^2 \log^2 \phi_E \log N})$, where ϕ_E is E's level of asynchrony. Moreover, our algorithm stands any constant number of faults. If E contains a constant number of faults, then our algorithm still achieves high throughput of $\Omega(\frac{N}{\phi_E'^2 \log^2 \phi_E' \log N})$, where ϕ_E' bounds the relative speeds of any two processes, at a time that both of them participated in E and none of them failed.
Our algorithm can be viewed as an adaptive version of the Bounded-Wait-Combining (BWC) prior art algorithm. BWC receives as an input an argument ϕ as a (supposed) upper bound of ϕ_E, and achieves optimal throughput if $\phi = \phi_E$. However, if the given ϕ happens to be lower than the actual ϕ_E, or much greater than ϕ_E, then the throughput of BWC degraded significantly. Moreover, whereas BWC is only lock-free, our algorithm is more robust, since it is wait-free.
To achieve high throughput and wait-freedom, we present a method that guarantees (for some common kind of procedures) the procedure's successful termination in a bounded time, regardless of shared memory contention. This method may prove useful by itself, for other problems.

1 Introduction

A shared counter is a shared object that holds an integer and supports the *fetch&increment* (FAI) operation for atomically incrementing the counter and returning its previous value. It is desirable that algorithms for shared counters be *linearizable*, robust, and scalable. Linearizability [17] is the most widely-used correctness condition for shared objects. Intuitively, it requires that each operation appears to take effect instantaneously at some moment between its invocation and response. Regarding robustness - *Lock-freedom* is a global progress

guarantee. It requires that some operation must be completed in a finite number of processes' steps. *Wait-freedom* [15] is a stronger guarantee - a process completes its own operation in a finite number of its own steps. Wait-freedom provides strong fault-tolerance [15]: no process can be prevented from completing an operation by undetected failures of other processes, or by arbitrary speed variations. We measure scalability in terms of shared objects throughput [12]. Intuitively, the throughput of a non-empty execution E is the ratio between E's duration and the number of completed FAI instances in E. For example, suppose the hardware supports a primitive of FAI. A counter implementation in which every process simply performs the FAI primitive on a shared base object is both linearizable and wait-free. However, this implementation is not scalable, since its throughput does not grow with N.

This paper presents the first linearizable and wait-free shared counter algorithm that achieves high throughput without any a-priori knowledge of the execution. Our algorithm can be easily adapted to work for any other combinable operation as well.

1.1 Related Work

To allow parallelism, researchers proposed highly-distributed coordination structures such as *counting networks* [1]. Though they are wait-free and scalable, the counting networks of [1] are not linearizable. Herlihy, Shavit, and Waarts demonstrated that counting networks can be adapted to implement linearizable counters [16]. However, the first counting network they present is not lock-free, while the others are not scalable, since each operation has to access $\Omega(N)$ base objects.

Combining is a well-established technique for highly parallel shared objects. Combining was introduced by Gottlieb et al. to be used in switches of a processor-to-memory network [8]. It reduces contention by merging several messages with the same destination. When a switch discovers several memory requests directed to the same memory location, a combined request is created to represent these requests. Separate responses to the original requests are created later from the reply to the combined request. Goodman et al. [7] introduced Combining Tree (CT), that was used to implement a linearizable and scalable counter, but their implementation is not lock-free. Shavit and Zemach [19] introduced *diffracting trees* to replace the static CT with a collection of randomly created dynamic trees. Diffracting trees were used to implement wait-free and scalable shared counters but they are not linearizable. Hoai Ha, Papatriantafilou, and Tsigas introduced another version of adaptive CT [11], but it is not linearizable either.

Chandra, Jayanti, and Tan [2] introduced a construction that implements a large class of objects they call *closed objects*. An object contains a set of operations, such that every operation in the set causes the object a state transition. The object is closed, if any two consecutive state transitions can be replaced with a single state transition that brings the object to the same state. A closed object may support the FAI operation. In their construction, processes com-

bine operations over dynamically created trees. Their algorithm is wait-free and linearizable, but its throughput does not grow with N.

Ellen, Lev, Luchangco, and Moir ([6]) introduced *Scalable Non-Zero Indicator* (SNZI), a shared object that is related to a shared counter, but has weaker semantics. SNZI can replace a shared counter for some applications. They present a linearizable implementation that is scalable and lock-free, and can be fast in the absence of contention. However, their approach does not seem suitable for a full fledged shared counter.

Achieving fault-tolerance using bounded-time locking was proposed by Gray and Cheriton in [9], in the context of caching. They presented the *lease*, that grants its holder control over writes to a covered datum (during the term of the lease). Another fault-tolerance tool is Greenwald's two-handed-emulation [10], which uses the *double-compare-and-swap* (DCAS) primitive to construct lock-free implementations of shared objects. When some process p calls the emulation to execute an operation o of the object, p tries to register a new instance of o to the emulation. Only one operation instance may be registered to the emulation at any moment. Hence, first, p has to assist executing the steps of the currently registered instance o' (if such exists). When o' is completed, the call of its creator process completes, and other process (possibly p) registers its operation instance. DCAS is used to ensure that exactly one process executes successfully the current step of the currently registered operation instance, while the other simultaneous writes have no effect.

Hendler and Kutten integrated combining, bounded waiting and two-handed-emulation, and proposed BWC, a linearizable and lock-free counter implementation [12]. Their algorithm receives as an input an argument ϕ and guarantees a high throughput of $\Omega(\frac{N}{\log N})$ in an N-process execution E in which ϕ is an asymptotically tight upper bound on E's level of asynchrony. Intuitively, BWC is a lock-free variation of CT, in which processes wait for each other, but only up to some bounded number of steps (determined using ϕ). However, in an execution E in which $\phi < \phi_E$, BWC achieves low throughput. When $\phi >> \phi_E$, BWC's throughput decreases by a factor of at least ϕ/ϕ_E compared to the optimum.

The difficulty arising from an unknown level of asynchrony received a lot of attention. Dolev, Dwork and Stockmeyer presented consensus algorithms for a number of partial synchrony models with different timing requirements and failure assumptions [3]. In some of their models, a fixed upper bound on the relative processor speed (denoted by ϕ) is not known in advance, and their protocols overcome this difficulty. However, their approach does not seem to provide a robust solution in our model, since a fixed ϕ may not exists. Even if it did, it is unclear how to broadcast it to all processes without high contention.

Dwork, Lynch and Stockmeyer suggested distributed consensus protocols that are tolerant to some number of failures in various models with different synchrony conditions [5]. They deal with point-to-point models and with models that allow multicasting to some of the processors in an atomic step. However, in their solutions, the level of asynchrony is known in advance.

Like in the case for counters, we are not aware of any linearizable and wait-free *deterministic* stack or queue algorithm that is also scalable. Hendler, Shavit, and Yerushalmi [14] presented an elimination-based *randomized* linearizable stack algorithm that is both lock-free and scalable in practice. Moir et al. [18] used ideas similar to [14] to obtain a queue algorithm that possesses the same properties.

The new algorithm presented in this paper can be viewed as an adaptive version of BWC, that gracefully adapts to the asynchrony bound. Unfortunately, we have not found a way to use BWC in a modular manner, and our algorithm is therefore rather involved (see Sect. 3.3).

The rest of the paper is organized as follows. Section 2 provides the model of the shared system we use. Section 3 explains some key concepts in the new algorithm, by outlining some previous combining techniques. Section 4 describes the new algorithm. Section 5 presents a new technique we use to execute concurrent procedures, that may be useful by itself, for other problems. Section 6 outlines the analysis of the new algorithm. In Sect. 7, we conclude and present directions for future work.

2 Model

We consider a shared-memory system, in which a set of N asynchronous processes communicate by applying to shared variables *read*, *write*, or *read-modify-write* primitives - *compare-and-swap* (CAS) and DCAS. To execute a primitive, a process performs an *invocation* of the operation, 0 or more *stall* steps and a *response* step. From the invocation step and until the response step, the operation is considered to be *pending*. If two or more processes have a pending write/read-modify-write operation on a shared variable v, then exactly one of them receives a response, and the rest are stalled. Similarly to [4], we assume that if a process p has a pending operation on a shared variable v, then p incurs a stall only if another process with a pending operation on v receives a response. Pending operations receive their responses in the order of their invocations.

A *configuration* specifies the value of each shared variable and the state of each process. An *initial configuration* is a configuration in which all the shared variables have their initial values and all the processes are in their initial states. An *execution fragment* is a sequence of steps, in which processes take steps and change states according to their algorithm. A process' state may change based on the response it receives in a response step only. An *execution* is an execution fragment that starts from the initial configuration. We assign times to execution steps as follows. Assigned times constitute a non-decreasing sequence of integers starting from 0. Let s be a step performed by process q, let $E = E_1 s E_2$, and let s' denote q's last step in E_1 (if any). The time assigned to s in E is denoted *time(E,s)*. If s is a response step, then *time(E,s)* is set to *time(E,s')*. If s is a stall step, then *time(E,s)* is set to *time(E,s')+1*. If s is an invocation step, *time(E,s)* is set to the maximum between 0 and (if exists): (1) the time of the last step in E_1; (2) *time(E,s')* plus 1; (3) the time of the last step in E_1 that accesses the same variable as s plus 1.

The *duration* of an execution E is the time of the last step of E. The *throughput* of an execution E is the ratio between the number of FAI operations that complete in E and E's duration.

Let E be an execution fragment. E is *ϕ-synchronous* if, for any E_0, E_1, E_2 such that $E = E_0E_1E_2$, and for any two distinct processes p and q, if p has to invoke some primitive at the end of E_0 and E_1 contains $\phi + 1$ invocation steps by q, then E_1 contains at least one invocation step by p.

An integer ϕ is a *correct asynchrony bound* for an execution E if it is an upper bound of the speed ratio between any two processes in E. *E's level of asynchrony*, denoted by ϕ_E, is the minimal corret asynchrony bound (or infinity if no finite bound exists). The *contention level* of an execution E is the maximum number of consecutive stalls that are incurred by a process in E.

For simplicity of presentation, we assume in the following that N is an integral power of 2. This assumption does not change our results: the solution for a set of N processes is equivalent to the solution for a set of $2^{\lceil \log N \rceil}$ processes (in which only N processes actually make requests).

3 Prior Art Combining Techniques

3.1 Combining Tree [7]

The CT algorithm [7] uses a full binary tree, in which each process owns one of the leaves and a unique *color*. A process p stores a new request in p's leaf, and climbs the tree towards the root. To climb to a new node n, n must be uncolored and p has to color n with p's own color. If p succeeds in coloring n, then p stays at n for some ϵ steps.

Some other process q may reach n during that time and has to wait at n (since n is colored by p). We say that p and q are *buddies*, such that p is the *climbing buddy*, q is the *waiting buddy* and n is their *meeting point*. The climbing buddy combines the requests of both buddies (stores at n a union of the requests that are stored in both child nodes of n) and climbs to bring corresponding responses for both buddies. The waiting buddy waits at the meeting point for the responses.

Process p may reach a node n_1 colored by some process q_1, but after q_1 had already finished forwarding requests into n_1. In that case, we say that p *suffers*, since it has to wait at n_1 until q_1 returns and *uncolors* n_1 (as described later). Only then p can color n_1 and proceed.

Eventually (assuming no faults occur), a union of requests that includes p's request is stored at the root by some process r, who produces corresponding responses. For example, responses for x FAI requests are an interval of x consecutive numbers starting with the current value of a counter at the root (the counter is increased accordingly by x).

Then, r descends towards its own leaf, uncoloring the nodes along the way. At every node n_2 which is a meeting point of r with some waiting buddy q_2 who gave y requests, r gives y responses to q_2. Both buddies then descend in the same manner, propagating responses to the rest of the waiting processes. When returning to the leaf with a response, a process returns that response.

Figure 1 depicts an illustration of a CT example. In this example, three processes wait for responses at non-root nodes, and one process is at the root with a combination of four requests, propagating four responses (responses 0 and 1 to the left child; responses 2 and 3 to the right child).

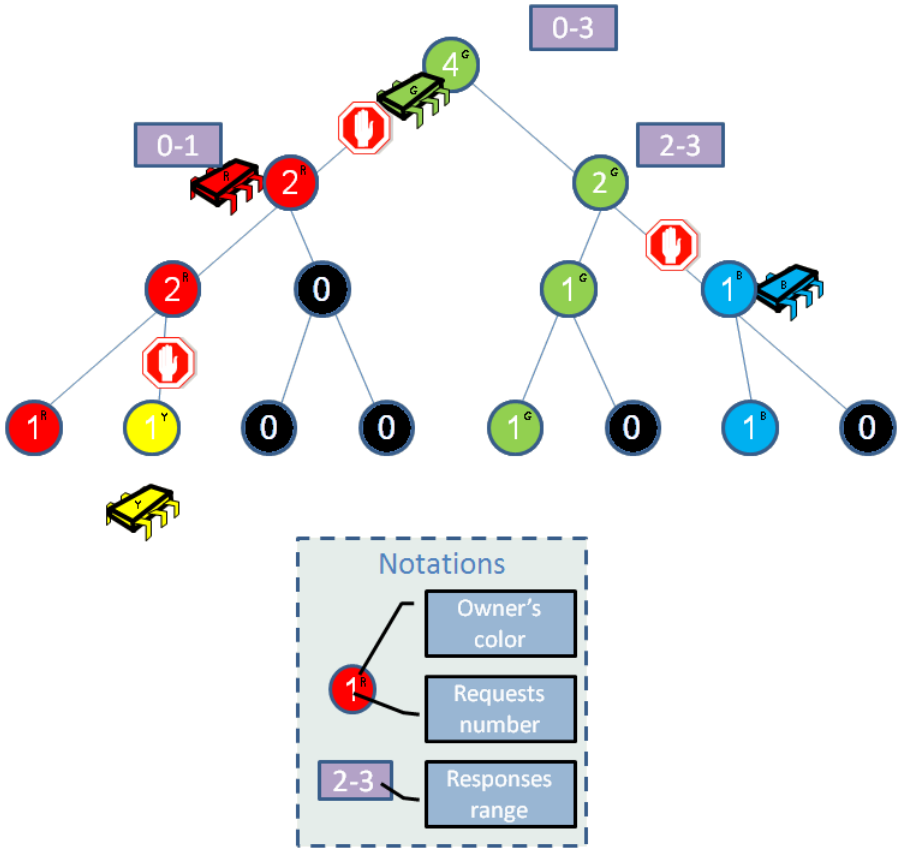

Fig. 1. Combining Tree

3.2 Bounded Wait Combining [12]

CT is linearizable but not lock-free, and does not guarantee high throughput. BWC is a lock-free version of CT that guarantees high throughput in ϕ-synchronous executions (assuming ϕ is known).

In CT, a deadlock can happen if a buddy p waits for one of its buddies indefinitely. In BWC, waiting times are bounded as a function of ϕ. Only if the exe-

cution is not ϕ-synchronous, p's bounded wait may expire, and then p proceeds in an *asynchronous mode of operation* (continues independently, without any waiting for other processes, to forward requests to the root and get responses). Greenwald's two-handed-emulation [10] is used to synchronize multiple processes who operate simultaneously at the same nodes without locks. This makes BWC lock-free but sequential in executions that are not ϕ-synchronous.

Another drawback of CT, is high latency when a process suffers many times in its way towards the root. If the execution is ϕ-synchronous, BWC guarantees that a process suffers at most once (per call). For that, BWC arranges the calling processes in *bunches of buddies*, in the following way: First, processes just try to color their way up towards the root (without storing or forwarding requests yet, or waiting ϵ time for buddies). A process r who colors the root becomes a *leader* of a new bunch (that contains only r at that moment). In the root, r waits a *root wait*, long enough as a function of ϕ to guarantee that every process that suffered (if such exists) catches a meeting point in the current bunch (waits on a node colored by a process who is in r's bunch, or going to be). Then, r descends back towards the leaf, and signals processes (who wait on r's meeting points) to join r's bunch. Similarly, these buddies of r descend back towards their own leaves and signals other processes to join r's bunch. From the leaves, the processes of the bunch start to forward requests like in CT, such that a union of the requests of all the processes in the bunch will be stored at the root, and corresponding responses will be produced and propagated.

3.3 Difficulties in Adapting to the Unknown Asynchrony Bound ϕ_E

Recall that BWC assumes a known bound on the ratio of the speeds of different processes. A first, trivial (but probably incorrect) idea towards getting rid of this assumption seems to be (1) to make multiple guesses of that bound and (2) run multiple copies of BWC modularly, one per guess. We tried that direction, but, unfortunately, did not manage to make it work. Intuitively, this would have violated the linearizability or the counter semantics. In BWC, a process gives its request (for a counter value) to be combined with other requests. When the request is combined, it is not easy to locate it and cancel it. Hence, assume the process gives the request to one copy of BWC. Then other copies cannot handle the same request, otherwise the counter would have been increased several times. Moreover, if those hypothetical copies of BWC communicate with each other, the modularity is lost. We did use multiple guesses of the bound, but we had to do that in a less modular way. Moreover, we had to overcome difficulties in coordinating processes working on different guesses.

For example, we could not use two-handed-emulation as is done by BWC. As long as BWC worked in its synchronous modes, only one process can move requests into each node. In the new algorithm, various processes may access the same node, each working on a different guess. This could have caused starvation, had we stayed with the original two-handed-emulation. The *communal procedures* technique we developed to solve this problem (see Sect. 5) may be useful for future studies and is an independent contribution of this paper.

4 The New Algorithm

We describe our algorithm in two levels. First, we provide a high level description of the algorithm (Sect. 4.1). Then, we provide a more detailed description of the algorithm's modes and transitions (Sect. 4.2). Detailed pseudo-code can be found in the technical report [13].

4.1 High level overview

In the new algorithm, a process p holds a *guess* of ϕ_E, according to which p attempts to perform an algorithm that resembles BWC. If p finds its guess insufficient, then p *restarts* (doubles its guess and tries again). We call the progress of p with a certain guess a *guess iteration*.

In the initial configuration, every node is uncolored and holds the dummy guess 1, while every process starts the algorithm with the minimum guess 2. When p colors a node n, it updates n's guess to p's guess. To color a node n, a process p must have a higher guess than n (even if n is already colored, p paints over with its own color). As soon as a process p_1 detects that its color was removed from a node n_1, it restarts. As a result, requests or responses that belong to other buddies in p_1's bunch are being left at n_1. We say that p_1's bunch was *interrupted*.

First, a process injects a request to its leaf and begins the first guess iteration with a guess 2. Let p be a process that begins a new guess iteration with a guess ϕ. Denote p's leaf by L_p and p's guess by G_p. First, p checks whether a response for p's request resides already at L_p (as a result of a former guess iteration, if such exists). If so, then p returns with that response. Otherwise, if G_p is above our defined *guess threshold*, then p shifts to an *asynchronous mode of operation*. In that mode, p round trips from L_p to the root and back (promoting requests and responses in its way towards their destination), until p finds its response in L_p, and returns. Assume there is no response in L_p and ϕ is lower than the guess threshold. Then, during the guess iteration, p aims to join a bunch of buddies, who later combine together requests and propagate responses. If ϕ is an incorrect guess, then at some point, this bunch may be interrupted by some process with a higher guess. If p detects such interruption, p restarts. Additionally, p bounds the number of steps it spends during the guess iteration with guess ϕ. If ϕ is smaller than the actual ϕ_E, then p may reach the bound and restart. If p restarts while some buddies wait for it, at some point they will restart too (either their bounded wait will expire or they will detect an interruption).

Let us describe the guess iteration progress, as long as p does not restart from reasons mentioned above. To color nodes (see Sect. 3), p climbs from L_p towards the root. Here (unlike CT and BWC), p succeeds to color a node n only if n's guess is lower than G_p. If so, p stores its color and guess into n. Eventually, some process r who guesses ϕ (either p or some other process) colors the root with the guess ϕ, becomes a bunch leader and begins a *root wait* (see Sect. 3.2). After the root wait, r descends to *recruit* to r's bunch processes who guess ϕ and wait on r's colored nodes. The bunch is constructed recursively (every recruited process

descends and recruits processes who guess ϕ and wait on its colored nodes). At every node m in which a process q_1 recruited a process q_2, we call m a *meeting point*, q_1 a *climbing buddy* and q_2 a *waiting buddy*.

Both in the case that p is the bunch leader itself, and the case that p was recruited by some other process, p descends back to L_p. From its leaf, p climbs up its colored nodes. At every non-leaf node n along its way, p forwards new requests from the children of n in p's bunch. If n is a meeting point of p with a waiting buddy q, and p reaches n before q, then p waits until q completes forwarding requests to their meeting point. Then, q waits at n for responses to all of its pending requests.

Eventually, the bunch leader r reaches the root again and forwards y new combined requests. The counter is increased from (say) z to $z+y$, and y responses (from z to $z+y-1$) are produced.

Then, r descends towards L_r. At every non-leaf node n along its way, r propagates new responses to n's children. In addition, r uncolors n and decrements n's guess, to allow recoloring of n later with the guess ϕ. Recursively, every waiting buddy in r's bunch (including p if $p \neq r$) gets responses for all the pending requests in its meeting point, and proceeds similarly towards its leaf. Finally, every process in r's bunch reaches its leaf and returns with the single response that resides there.

4.2 A more detailed description of the algorithm

The algorithm is composed of *modes*. Below, we describe each mode of the algorithm and specify the transition between modes. Figure 2 depicts the mode transitions diagram of our algorithm.

- *START* - Process p injects a new request to L_p, sets its own guess G_p to 2 and colors L_p. Then, p enters the first guess iteration, at the *INIT_RANK* mode.

- *INIT_RANK* - First, p updates L_p's guess to G_p. Second, p remembers L_p as its *top colored node* (the highest node p has colored in p's current guess iteration).
  If there is already a response for p's request in L_p, p switches to *PROP_RESPONSES* to return with the response. Else, if p's guess surpasses the guess threshold, p shifts to an *asynchronous mode of operation* (switches to *ASYNC_UP*).
  Otherwise, p switches to *SLOCK_UP*, to join (or lead) a bunch with the guess G_p.

- *SLOCK_UP* - In switching to this mode, p sets its *timer*. At the beginning of every *SLOCK_UP* iteration, p decreases its timer and performs two tests: A *timer verification* (that p's timer has not expired) and a *color verification* (that p's current node is still colored by p). If a test fails, p *restarts* (doubles its guess and switches back to *INIT_RANK*).

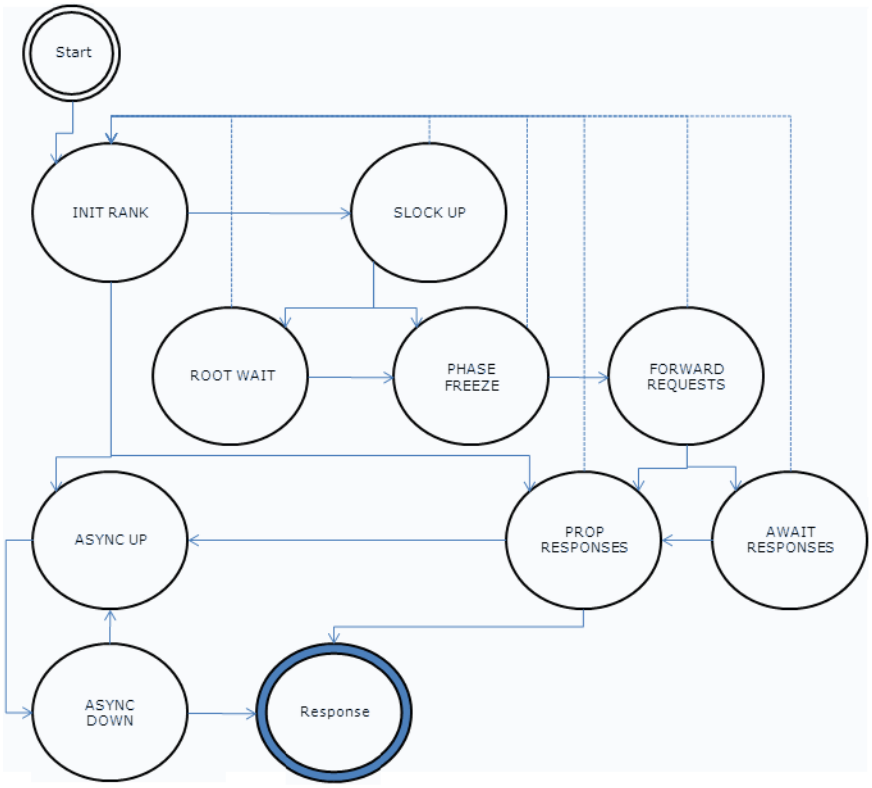

Fig. 2. Mode Transitions; Dashed arrows reflect a restart for a new guess iteration

Starting from L_p, at each non-root node n that p manages to color, p stores two indications: First, that n is ready to join a bunch with guess G_p. Second, that p has not forwarded yet requests to n. Note, that even though p has colored n successfully, some other process (with a guess higher than G_p) may recolor n at any time. Thus, p updates n only if n is colored by p. Using DCAS, p can detect an interruption (and restart if it finds any).
If n's parent (say np) has a guess that is lower than G_p, then p colors np with G_p, updates p's top colored node to be np and ascends to np. Otherwise, if a signal (an indication that p may join a bunch that is currently being formed) is written at n, then p (a waiting buddy at the meeting point n) switches to *PHASE_FREEZE*.
If p colors the root, p becomes a new bunch leader. To give other processes (who guess G_p too) time to climb up to a node that will be recruited to that bunch, p switches to *ROOT_WAIT*.
If it did not switch to any other mode, p stays for another *SLOCK_UP* iteration.

– *ROOT_WAIT* - In switching to this mode, p sets its timer. At every *ROOT_WAIT* iteration, p ensures that the root has not been recolored (if it is, then p restarts). The root wait is completed when p's timer expires, and p switches to *PHASE_FREEZE*.

– *PHASE_FREEZE* - In switching to this mode, p sets its timer. At every iteration, p decreases its timer and performs color and timer verifications (like in the *SLOCK_UP* mode).
Descending down from p's top colored node towards L_p, at each non-leaf node n along the way, p signals each child c of n that is waiting for a signal to join a bunch with guess G_p. When p reaches L_p, it switches to *FORWARD_REQUESTS*.

– *FORWARD_REQUESTS* - At every iteration, p decrements its timer and performs color and timer verifications.
At each non-leaf node n along p's way from L_p to p's top colored node, p forwards requests up to n from every child c of n who got a signal of p's current bunch. If c is colored by a waiting buddy q of p, then p waits at n until q indicates at c the completion of requests forwarding to c. Similarly, p indicates at n when it completes forwarding requests to n.
To handle simultaneous writes of different processes, this critical part is executed using a new technique (see Sect. 5). This technique, called *communal procedures*, guarantees successful completion after some known constant number of steps.
At p's *top colored* node, there may be two cases: (1) p is at the root with responses for its bunch, and switches to *PROP_RESPONSES*; (2) p is not at the root and switches to *AWAIT_RESPONSES* (to wait for responses from p's climbing buddy).

– *AWAIT_RESPONSES* - In switching to this mode, p sets its timer. At every iteration, p decrements its timer and performs color and timer verifications. If p gets all the required responses at its top colored node, p switches to *PROP_RESPONSES*.

– *PROP_RESPONSES* - In switching to this mode, p sets its timer. At every iteration, p decrements its timer and performs color and timer verifications. Starting from its top colored node, at each non-leaf node n in p's way to L_p, p propagates from n to n's children all the responses for the requests (again, using a communal procedure). In addition, p uncolors n and decrements n's guess.
Finally, p reaches L_p and returns with the single response that resides in it.

The *asynchronous mode of operation* consists of the following two modes:

– *ASYNC_UP* - p forwards requests in the path from L_p towards the root without waiting for any other process. At each non-leaf node n along

the way, p forwards the requests of both n's children (using a communal procedure). At the root, p switches to *ASYNC_DOWN*.

– *ASYNC_DOWN* - p propagates responses down the path from the root to L_p. Additionally, p uncolors and resets the guess of the nodes along the way, thus enabling future operations to avoid using high guesses (to stabilize the system, if it becomes more synchronous later on).
During the asynchronous mode of operation new pending requests may arrive to a node at each moment. Therefore, unlike in *PROP_RESPONSES*, p does not wait for responses to *all* of the pending requests at the node. Instead, p descends from a node after responding to up to two pending requests elements. When p reaches L_p, if a response resides at L_p, p returns with that response. Otherwise, p switches back to *ASYNC_UP*.

5 Communal Procedures

Different processes who call a procedure to forward requests to (or propagate responses from) the same node may try to write to the same variables simultaneously. Unlike in BWC, this may happen in the new algorithm not only when some process switches to an asynchronous mode of operation. Had we used two-handed-emulation as in BWC (see Sect. 3.2), starvation could happen and we could not guarantee wait-freedom and high throughput. The new algorithm requires that each call to such procedures is completed successfully after some finite number of steps (bounded by a known-in-advance constant). We present a technique that achieves all that, for some common kind of concurrent procedures we call *communal procedures*.

A procedure f is *communal* if when called by multiple processes simultaneously, a single complete instance of f yields the desired output for all the simultaneous calls. For example, a procedure to forward requests into a node from its children can be implemented as a communal procedure, since a single instance of this procedure can forward requests for all of the simultaneous calls.

A process executes a communal procedure f according to f's *state object*, that simply holds a *step number* variable (initialized to 0). The statements of f are translated into numbered steps (starting from step 1). The state object of f indicates the last step that was executed in the current instance of f. DCAS is used to execute the following step: one DCAS "hand" promotes the step number variable and the other "hand" performs the corresponding step. The DCAS ensures that only one of the simultaneous callers succeeds. When f's instance reaches its last step, to start a new instance, a process allocates a fresh state object (with a step number 0), and tries to update f's reference at the node to point to the new object (using DCAS that promotes the step number, such that only one update succeeds). Garbage collection (or equivalent techniques such as lock-free reference counting or dynamic lock-free objects, see [10]) guarantees that this object will not be recycled while any reference to it exists, and hence will be a unique token describing the particular procedure instance.

Many processes may wish to execute f at the same time. Every process helps executing f's instances only until it guarantees a full execution of at least one instance of f. For example, assume process p calls f at a node n, and finds f's instance at n at step number j. If $j = 0$, then p helps to execute f only until p detects a new instance of f at n. Otherwise ($j > 0$), p stays for another round to execute the steps of the new instance too, but only until p detects a third instance of f at n (because this ensures that the second instance was fully executed during p's call). Note, that the total number of primitives executed by p is bounded by twice the number of steps in f.

We use two communal procedures at every node (detailed pseudo-code appears at the technical report [13]): (1) ForwardRequestsFromChildren() - specifically for the FAI operation, a process p forwards requests to some node n from n's child c, by computing the number of additional requests to forward, and adding this number to n's counter. Later, the propagation of the responses for the requests must be in the order of requests arrivals (to preserve linearizability). Therefore, p enqueues an appropriate pending requests element to n's pending requests queue to record this order. If n is the root, then p also enqueues a responses element to n's responses queue. Specifically for the FAI operation, the responses element consists of the range of responses for the new requests (starting with the previous value of the root's requests counter). (2) PropagateResponsesToChildren() - a process p propagates responses from some node n to n's child c, according to a pending requests element from n's pending requests queue. Assume that such element requires x responses to be propagated down to c. Process p slices a range of x responses from n's responses queue and enqueues that range into c's responses queue.

6 Algorithm Analysis Outline

The complete analysis appears in the technical report [13]. First, let us define some additional terms. An *E-interval* is a sub-sequence of E. We say that $[t_0, t_1]$ is a *FAI-segment of* p, if it is an E-interval such that p starts a FAI operation at t_0 and returns at t_1. Let m be a mode of the algorithm. We say that $[t_0, t_1]$ is an *m-segment of* p, if p switches into mode m at t_0 and stays in this mode until it switches out of m at t_1 (or returns with a response). An *ASYNC-segment of* p starts when p switches from *INIT_RANK* to *ASYNC_UP* and ends when p returns with a response (or infinite if p does not return - though we prove this cannot happen). The outline of the analysis follows.

- Linearizability - First, we prove (Lemma 1) by induction that there are no overflows in the requests/responses queues in the nodes. This ensures that there are no interfering writes between a communal procedure who enqueues into such a queue and a communal procedure who dequeues from it. With that, we prove (Lemma 2) that no field changes its value between the moments a communal procedure reads from it and writes into it. Then, we prove (Lemma 3) that the critical procedures forward requests and propagate

responses as expected. With that, we prove (Lemma 4) that if a process ends the algorithm, it returns with a response. Finally, we prove (Theorem 1) the linearizability of our algorithm, and that it preserves the counter semantics.

- Wait-freedom - First, we bound (Lemma 5) the number of steps performed by a process during a single iteration at *any* mode. Second, we prove (Lemmas 6-9) that an $ASYNC$-segment of execution of a process is finite and its length is a function of N. Finally, we prove (Theorem 2) the wait-freedom of our algorithm, by proving that a process p either completes in a finite time or switches to the asynchronous mode of operation (for which, Lemma 6 proved the execution is finite).
- Scalability - Specifically, we prove that in an N-process execution E, our algorithm achieves the throughput of $\Omega(N/((\phi_E \log \phi_E)^2 \log N))$. If ϕ_E is greater or equal to the guess threshold, the proof is relatively straightforward. Otherwise (Assumption 1: ϕ_E is lower than the guess threshold), we prove that a process returns with a response after at most $\lceil \log_2 \phi_E \rceil$ guess iterations. For that, we prove (Lemma 23) that a process p with a correct guess G does not restart. Specifically, p succeeds in every color and timer verification during that guess iteration, until it returns with a response. A color verification may fail only if some other process has a guess higher than G. Thus, we prove (Lemmas 10-21) p's success at every timer verification, assuming no process guessed higher than G (Invariant 1), and with that we prove (Lemma 22) that no process can be the first to guess higher than G (i.e. Invariant 1 holds under Assumption 1). We bound the number of primitives p applies during a FAI-segment (Lemma 24), and finally, prove (Theorem 3) the above throughput whether or not Assumption 1 holds.
- Additional properties - We prove (Theorem 4) that our algorithm achieves high throughput even after a constant number of process failures. When some process p who guessed G fails, it has no effect on processes who guessed higher than G (because they don't wait on p's colored nodes, and are not p's bunch buddies). At the worst case, if G is a correct guess, then processes who guessed G would have to restart with guess $G+1$, which is also a correct rank.

7 Conclusions

In this paper we presented the first linearizable and wait-free shared counter algorithm that achieves high throughput without any a-priori knowledge of the system's asynchony level.

A communal procedure is a procedure that can be executed concurrently by multiple processes, such that a single complete execution of the procedure is sufficient for each participating process. Another contribution of our paper, which we believe is of independent interest, is the definition and efficient implementation of communal procedures.

The algorithm as described does not adapt when the level of asynchrony decreases during the execution. Such an adaptive version was developed, but

is not presented here because of its considerable additional complication. One direction for future research, is to make the throughput of the algorithm adaptive to the number of participating processes. Another interesting research direction is to establish corresponding lower bounds on the throughput of algorithms with similar properties.

References

1. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. STOC **23** (1991) 348–358
2. Chandra, T. D., Jayanti, P., Tan, K.: A Polylog Time Wait-Free Construction for Closed Objects. PODC **17** (1998) 287–296
3. Dolev, D., Dwork, C., Stockmeyer, L.: On the Minimal Synchronism Needed for Distributed Consensus. JACM **34** (1987) 77–97
4. Dwork, C., Herlihy, M., Waarts, O.: Contention in shared memory algorithms. STOC **25** (1993) 174–183
5. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the Presence of Partial Synchrony. JACM **35** (1988) 288–323
6. Ellen, F., Lev, Y., Luchangco, V., Moir, M.: SNZI: scalable NonZero indicators. PODC **26** (2007) 13–22
7. Goodman, J. R., Vernon, M. K., Woest, P. J.: Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. ASPLOS **3** (1989) 64–75
8. Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M.: The NYU ultracomputer designing a MIMD, shared-memory parallel machine. ISCA **9** (1998) 239–254
9. Gray, C., Cheriton, D.: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. SOSP **12** (1989) 202–210
10. Greenwald, M.: Two-Handed Emulation: How to build Non-Blocking implementations of Complex Data-Structures using DCAS. PODC **21** (2002) 260–269
11. Ha, P. H., Papatriantafilou, M., Tsigas, P.: Self-tuning Reactive Distributed Trees for Counting and Balancing. OPODIS **8** (2004) 213–228
12. Hendler, D., Kutten, S.: Constructing shared objects that are both robust and high-throughput. DISC **20** (2006) 428–442
13. Hendler, D., Kutten, S., Michalak, E.: Technical report for OPODIS 2010. http://ie.technion.ac.il/~kutten/hkm2010
14. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. SPAA **16** (2004) 206–215
15. Herlihy, M.: Wait-free synchronization. TOPLAS **13** (1991) 124–149
16. Herlihy, M., Shavit, N., Waarts, O.: Linearizable Counting Networks. FOCS **32** (1991) 526–535
17. Herlihy, M., Wing, J. M.: Linearizability: a correctness condition for concurrent objects. TOPLAS **12** (1990) 463–492
18. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. SPAA **17** (2005) 253–262
19. Shavit, N., Zemach, A.: Diffracting trees. SPAA **6** (1994) 167–176