Constructing Shared Objects that are Both Robust and High-Throughput

Danny Hendler* and Shay Kutten**

Faculty of Industrial Engineering and Management,
Technion, Haifa, Israel

Abstract. Shared counters are among the most basic coordination structures in distributed computing. Known implementations of shared counters are either blocking, non-linearizable, or have a sequential bottleneck. We present the first counter algorithm that is both linearizable, non-blocking, and can provably achieve high throughput in semisynchronous executions. The algorithm is based on a novel variation of the software combining paradigm that we call *bounded-wait combining*. It can thus be used to obtain implementations, possessing the same properties, of any object that supports combinable operations, such as stack or queue. Unlike previous combining algorithms where processes may have to wait for each other indefinitely, in the bounded-wait combining algorithm a process only waits for other processes for a bounded period of time and then 'takes destiny in its own hands'.
In order to reason rigorously about the parallelism attainable by our algorithm, we define a novel metric for measuring the throughput of shared objects which we believe is interesting in its own right. We use this metric to prove that our algorithm can achieve throughput of $\Omega(N/\log N)$ in executions where process speeds vary only by a constant factor, where N is the number of processes that can participate in the algorithm.
We also introduce and use *pseduo-transactions* - a technique for concurrent execution that may prove useful for other algorithms.

1 Introduction

At the heart of many distributed systems are *shared objects* - data structures that may be concurrently accessed by multiple processes. The most widely-used correctness condition for shared objects is *linearizability*, introduced by Herlihy and Wing [14]. Intuitively, linearizability requires that each operation appears to take effect instantaneously at some moment between its invocation and response. *Lock-free* implementations of shared objects require processes to coordinate without relying on mutual exclusion. They are considered more robust, as they avoid the inherent problems of locking, such as deadlock, convoying, and priority inversion.

* Supported in part by the Technion Aly Kaufman Fellowship and by the Israeli ministry of science and technology.
** Supported in part by the Technion foundation for the promotion of research.

A shared counter is an object that holds an integer and supports the *fetch&increment* operation for atomically incrementing the counter and returning its previous value. Shared counters are among the most basic coordination structures in distributed computing. Consequently, efficient implementations of shared counters received considerable attention in the literature. In spite of these efforts, existing counter implementations are either non-linearizable, blocking, or inherently sequential. This paper presents the first counter algorithm that is both linearizable, nonblocking, and highly parallel.

If the hardware supports the *fetch&increment* primitive, then the simplest way to implement a counter, shared by N processes, is by using the following trivial algorithm: all processes share a single base object on which they perform the *fetch&increment* operation to get a number. Although this central counter is both linearizable and nonblocking (it is, in fact, *wait-free* [11]), it has a sequential bottleneck. Specifically, the worst-case time complexity of this implementation is $\Omega(N)$: if all processes attempt to apply their operations simultaneously to the central counter, the last to succeed incurs a delay linear in N while waiting for all other earlier processes to complete their operations.

Fich et al. [4] proved an $\Omega(N)$ time lower bound on obstruction-free [12] implementations of a wide class of shared objects, that includes counters, stacks and queues. This lower bound establishes that no nonblocking counter algorithm can improve on a central counter in terms of worst-case time complexity. This does not preclude, however, the existence of nonblocking counter algorithms that achieve better worst-case time complexity in semisynchronous executions. Indeed, the worst-case time complexity of our algorithm in such executions is $O(\log N)$, yielding maximal throughput of $\Omega(N/logN)$.

To allow parallelism, researchers proposed using highly-distributed coordination structures such as *counting networks*. Counting networks were introduced by Aspnes et al. [2]. Though they are wait-free and allow parallelism, the counting networks of [2] are non-linearizable. Herlihy et al. demonstrated that counting networks can be adapted to implement wait-free linearizable counters [13]. However, the first counting network they present is blocking while the others do not provide parallelism, as each operation has to access $\Omega(N)$ base objects.

A well-established technique for constructing highly parallel shared objects is that of *combining*. Goodman et al. [5] and Yew et al. [19] used combining for implementing *fetch&add*. In both these algorithms, the current value of the counter is stored at the root of a binary tree. A process applies its operation starting from its leaf and climbing along the path to the root. Whenever two processes meet at an internal node, they combine their operations by generating a single request for adding the sum of both requests. One of these processes proceeds in climbing the tree while the other is blocked and waits at the node. When a process reaches the root, it adds to the central counter the sum of all the requests with which it combined and then starts a process of propagating responses back to the blocked processes. Combining trees can be used to implement linearizable counters and allow high parallelism but are blocking.

Shavit and Zemach introduce *diffracting trees* [18] to replace the "static" tree used in software combining with a collection of randomly created dynamic trees. Diffracting trees can be used to implement shared counters that are linearizable and allow parallelism but are blocking. Hoai Ha et al. introduce another version of (blocking) adaptive combining trees [8].

We introduce a variation on the software combining paradigm, that we call *bounded-wait combining*. Unlike in previous software combining algorithms, where processes may have to wait indefinitely for other processes, in bounded-wait combining, a process only waits for other processes for a bounded period of time and then 'takes destiny in its own hands'. As long as process speeds do not differ by more than some known fixed factor, processes wait for each other, eliminating contention for memory and achieving high parallelism. When the execution is asynchronous, however, processes fall back to an asynchronous modus operandi where high parallelism cannot be guaranteed but progress is. The result is *the first implementation of a linearizable counter that is both nonblocking and provably achieves high parallelism in semisynchronous executions.*

Our algorithm uses Greenwald's *two-handed emulation* mechanism [7] to implement a construct we term *pseudo-transactions*. Pseudo-transactions are weaker than ordinary transactions in that they are not atomic but they permit higher parallelism. Though they cannot replace transactions in general, we believe pseudo-transactions may prove useful also for other algorithms.

Geenwald's two-handed emulation is known to be sequential. However, We use it to implement pseudo-transactions by allowing different processes to operate on different two-handed emulation objects *in parallel*. The two-handed emulation mechanism uses the double compare-and-swap (DCAS) primitive. DCAS can be emulated efficiently by single-word compare-and-swap (CAS) by using, e.g., the algorithm of Harris et al. [9].

Bounded-wait combining can be adapted to work for any *combinable operation* [6, 15] and can thus be used to implement nonblocking and highly parallel linearizable stacks and queues. We are not aware of any other stack or queue algorithm to possess these properties.

Hendler et al. presented the *elimination-backoff stack*, a nonblocking linearizable stack algorithm [10]. Their empirical results show that their algorithm achieves high parallelism in practice; nevertheless, it does not provide any deterministic guarantee of parallelism. Moir et al. used ideas similar to these of [10] to obtain a queue algorithm that possesses the same properties [17].

In order to be able to reason rigorously about the parallelism attainable by our algorithm, we define a novel metric for the throughput of shared objects that may be interesting in its own right. By throughput, we mean the ratio between the number of operations that complete in an execution and the execution's duration. The key to this metric is a definition of time that assigns identical times to events that access different base objects and may be executed concurrently. To the best of our knowledge, this is the first formal metric for the throughput of shared objects. We use this metric to prove that our algorithm can achieve maximal throughput of $\Omega(N/\log N)$ in semisynchronous executions.

Model and Definitions. We consider a standard model of an asynchronous shared memory system, in which a finite set of asynchronous processes communicate by applying operations to shared objects [3].

Shared objects are implemented from *base objects*, such as read/write registers, provided by the system. A *configuration* specifies the value of each *base object* and the state of each process. An *initial configuration* is a configuration in which all the base objects have their initial values and all processes are in their initial states. To apply their operations, processes perform a sequence of *steps*. Each step consists of some local computation and one shared memory *event*, which is an application of a synchronization primitive (such as *read*, *write*, or *read-modify-write*) to a base object.

An *execution* is a sequence of events that starts from an initial configuration, in which processes apply events and change states (based on the responses they receive from these events) according to their algorithm.

An *operation instance* is an application of a specific operation with specific arguments to a specific object made by a specific process. If the last event of an operation instance Φ has been applied in an execution, we say that Φ *completes in* E. We define by *completed(E)* the number of operation instances that complete in E. We say that a process p is *active* after execution E if p is in the middle of performing some operation instance Φ, i.e. p has applied at least one event while performing Φ in E, but Φ does not complete in E. If p is active after E, then it has exactly one *enabled* event, which is the next event p will apply.

An execution E is *k-synchronous* if the speeds of any two processes that participate in it vary by a factor of at most k. Formally, we say that E is k-synchronous if, for any two distinct processes p and q and for any execution $E = E_0E_1E'$, if p has an enabled event after E_0 and E_1 contains $k + 1$ events by q then E_1 contains at least one event by p.

In addition to *read* and *write*, our algorithm uses the *compare-and-swap* (CAS) and the and *double-compare-and-swap* (DCAS) primitives. (The algorithm can be implemented on systems that support only read, write and CAS by using a DCAS emulation from CAS. See, e.g., [9].)

CAS(w, old, new) writes the value *new* to memory location w only if its value equals *old* and, in this case, returns *true* to indicate success; otherwise it returns *false* and does not change the value of w. *DCAS* operates similarly on two memory locations.

The rest of the paper is organized as follows. Section 2 provides an overview of the algorithm. Section 3 describes the synchronous part of the algorithm in more detail. Section 4 describes the procedures that forward operation requests and dispatch responses. Section 5 introduces our throughput metric. Concluding remarks are brought in Section 6

2 An Overview of the BWC Algorithm

In this section, we provide a high-level description of the *bounded-wait combining* (henceforth, BWC) algorithm. It can be used to provide a linearizable,

```
constant LEFT=0, RIGHT=1, FREE=⊥
structure Range {int from, int till}, typedef RRQ queue of Ranges
structure Reqs {int dir, int num}, typdef REQQ queue of Reqs
structure Node {
 Node* parent,                    Node* children[2],                  int reqs initially 0,
 int reqsTaken[2] initially {0,0}, REQQ pending initially EMPTY, RRQ resp initially EMPTY,
 boolean inPhase initially false, boolean collected initially false, int slock initially FREE,
 int phaseTop initially null
}
Node* nodes[2N-1]
```

Fig. 1. The structure of BWC combining-tree nodes.

nonblocking and high-throughput implementation of any combinable operation [6, 15]. For the sake of presentation simplicity, however, we describe it in the context of implementing a shared counter, supporting the *Fetch&increment* operation.

The algorithm uses a binary tree denoted $\mathcal{T}$. The value of the counter is stored at $\mathcal{T}$'s root. Each leaf of $\mathcal{T}$ is statically assigned to a single process. Every node of $\mathcal{T}$ consists of an instance of the *Node* structure shown in Figure 1, with the *parent* and *children* fields storing pointers to a node's parent and children, respectively. (Other node fields are described in the appropriate context.)

The BWC algorithm is parameterized: it uses an *asynchrony tolerance* parameter k that determines the extent to which processes are willing to wait for other processes. Each node contains a *synchronous lock*, which we simply call an *slock*. Whenever node n's *slock* equals the id of some process q, we say that n is *owned by* q; otherwise, we say that n is *free*. A leaf node is always owned by the process to which it is assigned. *Slocks are respected by processes in k-synchronous executions, but are disregarded in asynchronous executions* (i.e. in executions that are not k-synchronous).

For simplicity of pseudo-code presentation, we assume that variable scoping is dynamic, i.e. called procedures are in the scope of the calling procedure's local variables. The pseudo-code of the main procedure is shown in Figure 2. Initially, the algorithm behaves 'optimistically', in the sense that it operates under the assumption that the execution is k-synchronous. Whenever a process executes this part of the algorithm (implemented by the *SynchPhase* procedure, see Section 3), we say that it operates in a *synchronous mode*. As long as process speeds do not vary 'too much', processes operate in synchronous modes only and the algorithm guarantees low memory contention and high throughput.

While operating synchronously, computation proceeds in *synchronous phases*. In phase i, a subset of the participating processes construct a subtree $\mathcal{T}_i$ of $\mathcal{T}$, that we call a *phase subtree*. For every process q that participates in phase i, $\mathcal{T}_i$ contains all the nodes on the path from q's leaf node to the root.

Participating processes then use $\mathcal{T}_i$ as an ad-hoc combining tree. Each process q, participating in phase i, owns a path of nodes in $\mathcal{T}_i$ starting with q's leaf node and ending with the highest node along the path from q's leaf to the root whose *slock* q succeeded in acquiring. We denote q's path in $\mathcal{T}_i$ by $\mathcal{P}_i^q$.

After $\mathcal{T}_i$ is constructed, it is used as follows. Each participating process q starts by *injecting* a single new *operation request* at its leaf node. Processes then cooperatively perform the task of forwarding (and combining) these requests up $\mathcal{T}_i$. Process q is responsible for the task of forwarding requests from the sub-trees rooted at the nodes along $\mathcal{P}_i^q$. It may have to wait for other processes in order to complete this task. If the execution remains k-synchronous then, eventually, all collected requests arrive at the highest node of $\mathcal{P}_i^q$. If that node is not the root, then q now has to wait, as the task of forwarding these requests farther up $\mathcal{T}_i$ is now the responsibility of another process whose path ends higher in $\mathcal{T}_i$.

Finally, the operation requests of all participating processes arrive at the root. Once this occurs, the single process r whose path $\mathcal{P}_i^r$ contains the root increases the counter value stored at the root by the total number of requests collected. Process r then initiates the task of dispatching operation responses (which are natural numbers in the case of the *Fetch&Inc* operation) down $\mathcal{T}_i$. We call r the *phase initiator*. If the execution remains k-synchronous then eventually a response arrives at the leaf of each participating process.

A challenge in the design of the algorithm was that of achieving high throughput in k-synchronous executions while guaranteeing progress in *all* executions. To that end, the BWC algorithm is designed so that processes can identify situations where processes with which they interact are 'too slow' or 'too fast'. A detailed description of the *SynchPhase* procedure appears in Section 3.

After waiting for a while in vain for some event to occur, a process q may conclude that the execution is not k-synchronous. If and when that occurs, q falls back on an *asynchronous mode*. Once one or more processes start operating in asynchronous modes, high contention may result and high throughput can no longer be guaranteed but progress *is*.

The asynchronous part of the BWC algorithm is simple and its pseudo-code is omitted from this extended abstract for lack of space and can be found in the full paper. We now provide a short description. When process q shifts to an asynchronous mode, it injects an operation request at its leaf node, if it hadn't done so while operating synchronously. Process q then climbs up the tree from its leaf to the root. For every node n along this path, q forwards the requests of n's children to n. After it reaches the root, q descends down the same path to its leaf node, dispatching responses along the way. Process q *does not respect node* slock*s while traversing this path in both directions*. However, it releases the

```
Fetch&Inc()
1  boolean injected=false
2  int rc=SynchPhase()
3  if (rc ≠ ASYNCH)
4     return rc
5  else
6     return AsynchFAI()
```

Fig. 2. The main procedure of the BWC algorithm.

slock of every node it descends from. (This guarantees that if the system reaches quiescence and then becomes semisynchronous, processes will once more operate in synchronous modes.)

Process q keeps going up and down this path until it finds a response in its leaf node. We prove that the BWC algorithm guarantees global progress even in asynchronous executions, hence it is nonblocking.

Actual operation-requests combining and response-propagation is performed by the *FwdReqs* and *SendResp* combining procedures (see Section 4). The BWC algorithm allows different processes to apply these procedures to different nodes of $\mathcal{T}$ concurrently. In asynchronous executions, it is also possible that multiple processes concurrently attempt to apply these procedures to the same node. E.g., multiple processes may concurrently attempt to apply the *FwdReqs* procedure to node n for combining the requests of n's children into n.

The correctness of the algorithm relies on verifying that each such procedure application either has no effect, or procedure statements are applied to n in order without intervening writes resulting from procedures applications to other nodes. Moreover, the data-sets accessed by procedures that are applied concurrently to different nodes are, in general, not disjoint: a procedure applied to node n may have to read fields stored at n's parent or children. To permit high throughput, these reads must be allowed *even if other processes apply their procedures to these nodes concurrently.* It follows that procedures applied to nodes cannot be implemented as transactions. The BWC algorithm satisfies these requirements through an infrastructure mechanism that we call *pseudo-transactions.* We now describe the pseudo-transactions mechanism in more detail.

2.1 Pseudo-Transactions

Transactions either have no effect or take effect atomically. In contrast, concurrent reads *are* allowed while a pseudo-transaction executes but intervening writes are prohibited. Intuitively, pseudo-transactions suffice for the BWC algorithm because the information stored to node fields 'accumulates'. Thus reads that are concurrent with writes may provide partial data but they never provide inconsistent data. A formal definition of pseudo-transactions follows.

Definition 1. *We say that a procedure* P *is applied to an object* n *as a pseudo-transaction if each application of* P *either has no effect or the statements of* P *are applied to* n *in order and no field written by a statement of* P *is written by a concurrently executing procedure.*

The following requirements are met to ensure the correctness, liveness, and high-parallelism of the BWC algorithm:

1. **Combining correctness**: the combining procedures *FwdReqs* and *SendResp* are applied to nodes as pseudo-transactions.
2. **Node progress**: progress is guaranteed at every node. In other words, after some finite number of statements in procedures applied to n are performed, some procedure applied to n terminates.

The BWC algorithm meets the above requirements by using the following two mechanisms. First, We treat each node (in conjunction with the *FwdReqs* and *SendResp* combining procedures) as a separate object. *We apply Greenwald's two-handed emulation to each of these objects separately* [7]. A detailed description of two-handed emulation is beyond the scope of this paper, and the reader is referred to [7]. We provide a short description of the emulation in the following.

Greenwald's two-handed emulation uses the *DCAS* operation to ensure that the statements of an applied procedure execute sequentially. To apply a procedure to object n, a process first tries to register a procedure-code and procedure operands at a designated field of n by using *DCAS*. Then, the process tries to perform the read and write operations of the procedure one after the other. Each write to a field of n uses *DCAS* to achieve the following goals: (1) verify that the write has not yet been performed by another process, (2) increment a virtual "program counter" in case the write can be performed, and (3) perform the write operation itself.

In addition to using two-handed emulation, we have carefully designed the node structure so that applications of the *FwdReqs* or *SendResp* combining procedures to different nodes never write to the same field (see Section 4 for more details).

Two-handed emulation guarantees that a combining procedure applied to node n either has no effect (if its registration failed) or its statements are performed with no intervention from other procedures applied to n. As procedures applied to different nodes never write to the same field, combining procedures are applied as pseudo-transactions and requirement 1. above is satisfied.

Applying two-handed emulation to an object n results in a nonblocking implementation, *on condition that procedures applied to other nodes cannot fail DCAS operations performed by a procedure applied to n.* Since procedures applied to different nodes never write to the same field, none of them can fail the other. Thus, Requirement 2. is also satisfied.

3 The Synchronous Modus Operandi

This part of the algorithm is implemented by the *SynchPhase* procedure (see pseudo-code in Figure 3) which iteratively switches over the *mode* local variable. Variable *mode* stores a code representing the current synchronous mode.

In the following description, q is the process that performs *SynchPhase*. The local variable n stores a pointer to the *current node*, i.e. the node currently served by q. Initially, n points to q's leaf. In some of the modes (*UP*, *FORWARD_REQUESTS* and *AWAIT_RESP*), q may have to wait for other processes. Before shifting to any of these modes, the local variable *timer* is initialized to the number of iterations q should wait before it falls back to an asynchronous mode. In the *ROOT_WAIT* mode, q waits so that other processes can join the phase it is about to initiate. In all of these cases, *timer* is initialized to some appropriately selected function of k and $\log N$. In the specification of these wait-

ing times, M denotes the maximum number of events applied by a process in a single iteration of the *SynchPhase* procedure which is clearly a constant number.

We prove in the full paper that this selection of waiting periods guarantees that no process shifts to an asynchronous mode in k-synchronous executions. We now describe the synchronous modes.

UP: This is q's initial mode. Process q starts from its leaf node and attempts to climb up the the path to the root in order to join a phase. To climb from a non-root node to its parent m, q first verifies that m is free (statement 15), in which case it performs a *CAS* operation to try and acquire m's *slock* (statement 16). If it succeeds, q sets its current node n to m and reiterates. If m is not free, q checks the *m.inPhase* flag (statement 18) which indicates whether or not m is part of the current phase's subtree. If it is set then q checks whether n was added to the phase subtree by m's owner (statement 19). It both conditions hold, then q managed to join the current phase and all the nodes along the path from its leaf to n will join that phase's subtree. In this case, q shifts to the *PHASE_FREEZE* mode and stores a pointer to its highest node along this path (statements 20, 21). Otherwise, if q acquired the root node *slock*, then q is about to be the initiator of the next phase. It stores a pointer to the root node, sets the number of iterations to wait at the root to $\Theta(\log N)$, and shifts to the *ROOT_WAIT* mode (statements 12 - 14). If none of the above conditions hold, q decrements *timer* and, if it expires, concludes that the execution is asynchronous and returns the *ASYNCH* code (statements 23, 24).

ROOT_WAIT: q waits in this mode for the iterations timer to expire in order to allow other processes to join the phase subtree. While waiting, q performs a predetermined number of steps, in each of which it applies a single shared memory event. Finally, q shifts to the *PHASE_FREEZE* mode (statements 25-28).

PHASE_FREEZE: in this mode, q freezes the nodes along its path from *phaseTop* to its leaf and the children of these nodes. Freezing a node adds it to the phase's subtree. To determined which of n's children is owned by it, q uses the *LEAF_DIR* macro that, given an internal node n and a process id, returns *LEFT* or *RIGHT* (statement 29). For every node n, if the child of n not owned by q is not free, q sets that child's *inPhase* flag, sets n's *inPhase* flag, and descends to n's child on the path back to its leaf (statements 31-33, 40). When q gets to the parent of its leaf, it injects a single request to its leaf, sets the iterations counter, sets a flag indicating that a request was injected, and shifts to the *FORWARD_REQUESTS* mode (statements 35-38).

FORWARD_REQUESTS: in this mode, q forwards and combines requests along the path starting with the parent of its leaf and ending with *q.topPhase*. For each node n along this path, q checks for each child *ch* of n whether *ch* is in the current phase's subtree and whether requests need be forwarded from it (statement 42). If so and if *ch*'s *collected* flag is set, requests from the subtree rooted at *ch* were already forwarded and combined at *ch*. In this case q calls the *FwdReqs* procedure to forward these requests from *ch* to n and sets the local

```
SynchPhase()
1  Node* n=LEAF(myID), int phaseTop, int mode=UP, int timer=Mk(k + 13) log N + 1
2  boolean chCollected[2]={false,false}
3  do forever
4     switch(mode)
5        case UP: UpMode()                                  6  case ROOT_WAIT: RootWait()
7        case PHASE_FREEZE: PhaseFreeze()                   8  case FORWARD_REQUESTS: ForwardRequests()
9        case AWAIT_RESP: AwaitResp()                       10 case PROP_RESP: PropResp() od

UpMode()
11 if (ROOT(n))
12    phaseTop=n
13    timer=2M(k + 1) log N
14    mode=ROOT_WAIT
15 else if (n.parent.slock=FREE)
16    if (CAS(n.parent.slock, FREE, myID))
17       n=n.parent
18 else if (n.parent.inPhase)
19    if (n.inPhase)
20       phaseTop=n
21       mode=PHASE_FREEZE
22 else
23    timer=timer-1
24    if (timer=0) return ASYNCH

RootWait()
25 if (timer > 0)
26    read n.slock
27    timer=timer-1
28 else mode=PHASE_FREEZE

PhaseFreeze()
29 int whichChild=LEAF_DIR(n, myID)
30 Node *ch=n.children[whichChild]
31 if (n.children[1-whichChild].slock ≠ FREE)
32    n.children[1-whichChild].inPhase=true
33 n.inPhase=true
34 if (ch=LEAF(myID))
35    ch.reqs=ch.reqs+1
36    injected=true
37    timer=M(k + 1) log N
38    mode=FORWARD_REQUESTS
39 else
40    n=ch

ForwardRequests()
41 (for i=LEFT; i≤RIGHT; i++)
42    if (n.children[i].inPhase ∧¬ chCollected[i])
43       if (n.children[i].collected)
44          FwdReqs(n, i)
45          chCollected[i]=true
46       else
47          timer = timer -1
48          if (timer = 0) return ASYNCH
49          else continue do-forever (statement 3)
50 n.collected=true
51 if (n ≠ phaseTop)
52    n=n.parent
53 else if (ROOT(n))
54    mode=PROP_RESP
55 else
56    timer=3Mk log N
57    mode=AWAIT_RESP

AwaitResp()
58 if (¬ EMPTY(n.resp))
59    mode=PROP_RESP
60 else
61    timer=timer-1
62    if (timer=0) return ASYNCH

PropResp()
63 if (n = LEAF(myID))
64    Range r=DEQ_R(resp)
65    if (RLEN(r) > 0) return r.first
66    else return ASYNCH
67 SendResp(n)
68 n.inPhase=false, n.collected=false, n.slock=FREE
69 n=n.children[LEAF_DIR(n,myID)]
```

Fig. 3. Pseudo-code for the synchronous part of the algorithm with asynchrony tolerance k.

flag *chCollected* corresponding to *ch* in order to not repeat this work (statements 43-45). If *ch*'s *collected* flag is not set, q decrements *timer* and continues to wait for that event to occur; if the timer expires, q returns the *ASYNCH* code (statements 47 - 49). If and when q succeeds in forwarding requests from each of n's children that is in the phase, it sets n's *collected* flag and climbs up. Eventually, it shifts to either the *PROP_RESP* mode or the *AWAIT_RESP* mode, depending on whether or not it is the current phase's initiator (statements 50-57).

AWAIT_RESP: In this mode, q, when not the initiator of the current phase, awaits for the owner of node n's parent to deliver responses to n. If and when

this event occurs, q shifts to the *PROP_RESP* mode (statements 58, 59). If *timer* expires, q returns the *ASYNCH* code (statement 62).

PROP_RESP: In this mode, q propagates responses along the path from *q.phaseTop* down to its leaf node. For each node n along this path, q propagates n's responses to its children and then frees n (statements 67-69). Eventually, q descends to its leaf. If a response awaits there, it is returned as the response of *SynchPhase*. Otherwise, q returns *ASYNCH* (statements 63-66).

4 The Combining Process

The combining process is implemented by the *FwdReqs* and *SendResp* procedures. The pseudo-code of these procedures appears in Figure 4. As discussed in Section 2.1, the code actually performed is a transformation of the code shown in Figure 4 according to Greenwald's two-handed emulation technique, as indicated by the *2he* attribute. The *FwdReqs* procedure forwards requests from a child node to its parent. The *SendResp* procedure dispatches responses from a parent node to its children. The two procedures use the following node fields.

reqs: For a leaf node n, this is the number of requests injected to n. If n is an internal node other than the root, this is the number of requests forwarded to n. If n is the root, this is the current value of the counter.

reqsTaken: this is an array of size 2. For each child m of n, it stores the number of requests forwarded from m to n.

pending: this is a queue of *Reqs* structures. Each such structure consists of a pair: a number of requests and the direction from which they came. This queue allows a process serving node n to send responses in the order in which the corresponding requests were received. This allows maintaining the linearizability of the algorithm. In k-synchronous executions, the *pending* queue contains at most 2 entries. In asynchronous executions it contains at most n entries, as there are never more than n simultaneous operations applied to the counter.

resp: this is a producer/consumer queue storing response ranges that were received at n and not yet sent to its children. The producing process (the one that serves the parent node) enqueues new response ranges and the consumer process (the one that serves the child node) dequeues response ranges. The producer and consumer processes never write to the same field simultaneously. The *resp* queue contains at most a single range in k-synchronous executions. In asynchronous executions it contains at most n ranges. We now describe the pseudo-code of these two procedures that are performed as pseudo-transactions. Here, q is the process executing the code.

The *FwdReqs* procedure receives two parameters. A pointer n to the node to which requests need to be forwarded, and an integer, *dir*, storing either *LEFT* or *RIGHT*, indicating from which of n's children requests need to be forwarded to n. Let m denote the child node of n that is designated by the *dir* parameter. Process q first initializes a pointer to m (statement 1). Then q computes the number of requests in m that were not yet forwarded to n and stores the result in *delta* (statement 2). If there are such requests, q proceeds to forward them.

```
2he FwdReqs(NODE* n, int dir)
1  Node* child = n.children[dir]
2  int delta=child.reqs - n.reqsTaken[dir]
3  if (delta > 0)
4     n.reqs=n.reqs+delta
5     n.reqsTaken[dir]=n.reqsTaken[dir]+delta
6     ENQ_REQS(n.pending, <dir,delta>
7     if (ROOT(n))
8        ENQ_R(n.resp, n.reqs-delta+1, delta)
```

```
2he SendResp(NODE* n)
9  do twice
10    if (¬ EMPTY(n.resp))
11       Range fResp=FIRST_R(n.resp)
12       int respsLen=RLEN(fResp)
13       RRQ fReqs=FIRST_REQS(n.pending)
14       int reqsLen=REQS_LEN(fReqs)
15       int send=min(respLen, reqsLen)
16       int dir=REQS_DIR(fReqs)
17       ENQ_R(n.children[dir].resp, fResp.start, send)
18       DEQ_R(n.resp, send)
19       DEQ_REQS(n.pending, send)
20 od
```

Fig. 4. Pseudo-code for the combining procedures

Forwarding the requests is implemented as follows. Process q first increases n's *reqs* field by *delta* (statement 4) to indicate that *delta* additional requests were forwarded to n. It then increases n's *reqsTaken* entry corresponding to m by *delta* (statement 5). This indicates that *delta* additional requests were forwarded from m to n. Finally, q adds an entry to n's *pending* queue specifying that *delta* more responses need to be sent from n to m. If n is the root node, then the operations represented by the forwarded requests are applied to the central counter at the root when *n.reqs* is increased by statement 4. In that case, q immediately adds the corresponding range of counter values to n's queue of response ranges (statements 7-8).

The *SendResp* procedure receives a single parameter - n - a pointer to the node from which responses need be sent. Process q executes the loop of statements 9-20 twice. If the responses queue is not empty, the length of its first range is computed and stored to the *respLen* local variable (statements 11, 12). Then the length of the next requests entry is computed and stored to *reqsLen* (statements 13, 14). The minimum of these two values is stored to *send* (statement 15). This number of responses is now sent in the direction specified by the *dir* field of the first requests entry (statements 16, 17). Finally, *send* responses and requests are removed from the *n.resp* and *n.pending* queues, respectively. The loop of statements 9-20 is executed twice. This is enough to propagate the responses for all the requests forwarded to a node in a synchronous phase.

For presentation simplicity, the *reqs* and *reqsTaken* fields used in the pseudo-code of Figure 4 are unbounded counters. To ensure the correctness of the combining process, however, it is only required that the difference between the values of these fields be maintained. This difference cannot exceed N, since this is the maximum number of concurrent operations on the counter. It follows that the code can be modified to use bounded fields that count modulo $2N + 1$.

5 A Metric for the Throughput of Concurrent Objects

We now present a metric for quantifying the throughput of concurrent objects. This metric allows us to reason about the throughput of concurrent objects

rigorously. The key to the metric is a definition of time that assigns identical times to events that may be executed concurrently.

Let E be an execution. Consider a subsequence of events of E applied by some process p. As processes are sequential threads of execution, the times assigned to the events applied by p must be strictly increasing. Consider a subsequence of events in E, all of which access the same base object o. Here it is less obvious how to assign times to these events. If the value of o is never cached, then the times assigned to these events must be strictly increasing as accesses of o are necessarily serialized. However, if o may be cached, then it *is* possible for o to be read concurrently by multiple processes accessing their local caches. Thus an alternative assignment of times, where consecutive reads of the same base object may be assigned identical times, is also possible.

For the analysis of the BWC algorithm, we chose to apply the stricter definition where time must strictly increase between accesses of the same object. We define the throughput of an execution as the ratio between the execution's duration and the number of operation instances that complete in it. It follows that the throughput of an algorithm can only increase if the alternative (less strict) assignment of times were to be used. Formal definitions of execution time and throughput follow.

The subsequence of events applied by process p in E is denoted by $E|p$. The subsequence of events that access a base object o in E is denoted by $E|o$. We assign times to execution events. Assigned times constitute a non-decreasing sequence of integers starting from 0. Times assigned to the subsequence of events applied by any specific process p are monotonically increasing. Similarly, times assigned to the subsequence of events that access any specific object o are monotonically increasing. This is formalized by the following definitions.

Definition 2. *Let e be an event applied by process p that accesses base object o in execution E and let $E = E_1 e E_2$. The* synchronous time *assigned to e in E, denoted by* time(E,e)*, is defined to be the maximum of the following numbers:*

- *the synchronous time assigned to the last event of E_1 (or 0 if E_1 is empty),*
- *the synchronous time assigned to p's last event in E_1 plus 1 (or 0 if $E_1|p$ is empty),*
- *the synchronous time assigned to the last event in E_1 that accesses o plus 1 (or 0 if $E_1|o$ is empty).*

Definition 3. *The* synchronous duration *of an execution E, denoted by* time(E)*, is 0 if E is the empty execution or $time(E, e_l) + 1$ otherwise, where e_l is the last event of E. The* throughput of a *(non-empty)* execution *E is defined to be* completed(E) / time(E)*.*

Based on the above definitions, we prove the following theorem.

Theorem 1. *The throughput of the BWC algorithm with asynchrony tolerance parameter k in k-synchronous executions in which all processes start their operations concurrently is $\Omega(N/\log N)$.*

To prove theorem 2, we first show that the selection of the waiting times with which the *timer* variable is set guarantees that processes always operate in synchronous modes in k-synchronous executions. We then show that as long as all processes operate in synchronous modes and the execution is k-synchronous, every operation instance completes in $O(\log N)$ time. The proof is omitted from this extended abstract for lack of space and can be found in the full paper.

We also prove the following theorem in the full paper.

Theorem 2. *The BWC algorithm is a nonblocking linearizable counter implementation.*

Intuitively, the linearizability of the BWC algorithm follows from the fact that the counter value is stored at the root node and that responses are dispatched from each node n in the order in which the corresponding requests were received at n. We show that, as long as all processes operate in synchronous modes, the BWC algorithm is wait-free. If some processes fall back on asynchronous modus operandi, then we show that the **Node progress** property (see Section 2.1) guarantees overall progress.

6 Concluding Remarks

In this paper we present Bounded-Wait Combining (BWC), the first nonblocking linearizable counter algorithm that can provably achieve high parallelism in semisynchronous executions. Bounded-Wait Combining can be used to obtained implementations, possessing the same properties, of objects such as stack and queue. We define a novel metric of the throughput of concurrent algorithms and use it to analyze our algorithm. We also introduce and use *pseudo-transactions* - a concurrent execution technique that, though weaker than ordinary transactions, permits higher parallelism. We believe that both our throughput metric and the pseudo-transactions construct may prove useful in the design and analysis of other algorithms.

Our algorithm guarantees the nonblocking property in *all* executions. However, to guarantee high throughput, it is required that processes know in advance an upper bound on the ratio between the fastest and slowest processes. (The requirement for knowledge of timing information is similar to that made in the *known bound* model [1, 16].) We believe the BWC algorithm can be extended to adjust adaptively to an unknown system bound on the speed ratio. Also, the tree used by the BWC algorithm is static and of height $\Theta(\log N)$, regardless of the number of processes participating in the computation. It follows that the time it takes to apply an operation is $\Theta(\log N)$ even if a process runs solo. It would be interesting to see whether the algorithm can be made 'dynamic' from this respect while maintaining its properties. We leave these research problems for future work.

Acknowledgements. We thank the anonymous reviewers for many useful comments.

References

1. R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM J. Comput.*, 26(2):539–556, 1997.
2. J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.
3. H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. Wiley, 2004.
4. F. E. Fich, D. Hendler, and N. Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *FOCS 2005*, pages 165–173.
5. J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *ASPLOS*, pages 64–75, 1989.
6. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer designing a mimd, shared-memory parallel machine. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 239–254, New York, NY, USA, 1998. ACM Press.
7. M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas. In *PODC 2002*, pages 260–269, 2002.
8. P. H. Ha, M. Papatriantafilou, and P. Tsigas. Self-tuning reactive distributed trees for counting and balancing.
9. T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC 2002*.
10. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04*, pages 206–215, New York, NY, USA. ACM Press.
11. M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, January 1991.
12. M. Herlihy, V. Luchango, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS 2003*, pages 522–529.
13. M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.
14. M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, June 1990.
15. C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. *ACM TOPLAS*, 10(4):579–601, 1988.
16. N. A. Lynch and N. Shavit. Timing-based mutual exclusion. In *IEEE Real-Time Systems Symposium*, pages 2–11, 1992.
17. M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA'05*, pages 253–262, New York, NY, USA.
18. N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.
19. P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, 1987.