Bounded-Wait Combining: Constructing Robust and High-Throughput Shared Objects*

Danny Hendler† Shay Kutten‡

December 20, 2008

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, nonblocking, and can provably achieve high throughput in *k-synchronous* executions – executions in which process speeds vary by at most a constant factor k. 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 a stack or a 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 achieves throughput of $\Omega(N/\log N)$ in k-synchronous executions, where N is the number of processes that can participate in the algorithm.

Our algorithm uses two tools that we believe may prove useful for obtaining highly parallel non-blocking implementation of additional objects. The first are "synchronous locks", locks that are respected by processes only in k-synchronous executions and are disregarded otherwise; the second are "pseduo-transactions" - a weakening of regular transactions that allows higher parallelism.

*A preliminary version of this paper appeared in [11].

†Department of Computer Science, Ben-Gurion University. Supported in part by a grant from the Israel Science Foundation.

‡Faculty of I&M, Technion. Supported in part by a grant from the Israel Science Foundation.

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 [16]. Intuitively, linearizability requires that each operation appear 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.

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, however, 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 in *k-synchronous executions* – executions in which process speeds vary by at most a constant factor of k.

If the hardware supports the *fetch&increment* synchronization primitive, then the simplest way to implement a counter, shared by N processes, is by using the following straightforward implementation: all processes share a single base object on which they perform the *fetch&increment* operation to get a number. Although this implementation is both linearizable and lock-free (it is, in fact, wait-free [13]), it has a sequential bottleneck.

To allow parallelism, researchers proposed using highly-distributed coordination structures such as *counting networks*. Counting networks were introduced by Aspnes, Herlihy, and Shavit [1]. Though they are wait-free and allow high parallelism, the counting networks of [1] are non-linearizable. Herlihy, Shavit, and Waarts demonstrated that counting networks can be adapted to implement wait-free linearizable counters [15]. 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*. Combining was invented by Gottlieb et al. to be used in switches of a processor-to-memory network [8]. It reduced contention and allowed parallelism by merging several messages with the same destination. When a switch discovers several memory requests directed to the same memory location, a new, combined, request is created to represent these requests. Separate responses to the original requests are later created from the reply to the combined request.

Goodman et al. [7] and Yew et al. [22] implemented the combining technique in software for performing *fetch&add*. In both these algorithms, the current value of the counter is stored at the root of a binary tree. To apply its operation, a process starts from its leaf and climbs 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 may allow high parallelism. Though combining trees guarantee progress when the system is semi-synchronous, they are blocking in asynchronous systems, where processes may fail-stop.

Shavit and Zemach introduced *diffracting trees* [19] 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 wait-free and may provide high throughput but are non-linearizable. Hoai Ha, Papatriantafilou and Tsigas introduced another version of adaptive combining trees [10]. Non-blocking combining trees can be obtained by applying one of a few published general techniques for transforming blocking objects into semantically equivalent non-blocking objects (see, e.g., [14, 9]). However, these transformations result in objects that are essentially sequential since all operations need to access a central location.

Shavit and Zemach introduced *combining funnels* [20], a scheme for dynamically constructing combining trees with depth logarithmic in the number of processes concurrently accessing the data structure. They also present a combining-funnels-based linearizable wait-free counter that can provide high throughput but is blocking. Wattenhofer and Widmayer presented the *counting pyramide* [21], a similar adaptive variation of combining for message-passing systems.

Recently, Ellen et al. introduced Scalable NonZero Indicator (SNZI), a shared object that is related to a "regular" shared counter, but has weaker semantics [5]. A SNZI object supports *Arrive* and *Depart* operations, that are similar to increment and decrement, and a *Query* operation that returns an indication as to whether or not there is a surplus of *Arrive* operations as compared to *Depart* operations. SNZI can replace a shared counter for some applications. They present a scalable, linearizable nonblocking implementation that can be fast in the absence of contention, a combination of properties that might be impossible to provide by a regular shared counter.

Herlihy, Luchangco and Moir introduced the concept of *obstruction-free* implementations. An implementation is obstruction-free, if it guarantees that a process completes its operation in the absence of *step contention* [3]. Any nonblocking implementation is also obstruction-free. Fich, Hendler and Shavit [6] proved a linear lower bound on the worst-case number of *memory stalls* [4] incurred by a single operation for linearizable obstruction-free implementations of a wide class of objects, that includes counters, stacks, and queues. Their result implies that the worst-case operation latency of these implementations is no better than that of a sequential implementation. The lower bound is obtained, however, in asynchronous executions and does not preclude the existence of nonblocking linearizable implementation that provide low latency (hence also high throughput under high contention) in semi-synchronous executions. Indeed, our algorithm does exactly that.

We introduce a variation on the software combining paradigm, that we call *bounded-wait combining*. 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 takes contention in accessing shared memory into consideration while assigning 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 the *latency* of a *fetch&increment* operation performed by our algorithm in executions where process speeds vary by at most a constant factor is $\Theta(\log N)$ (Theorem 1), *regardless of the level of contention*. This implies that the maximum throughput attainable by our algorithm is $\Theta(N/\log N)$. This throughput is achieved in execution in which all processes participate. On the down side, similarly to counting networks and to static combining trees, throughput falls down to $\Theta(1/\log N)$ when only a constant number of processes participate in the execution.

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'. Technically, this is accomplished by having processes wait on *synchronous locks* (shortened to *slock*s). As long as process speeds do not vary 'too much', processes wait for each other on slocks, eliminating contention for memory and achieving high parallelism. When the execution is asynchronous, however, processes fall back to an asynchronous modus operandi where slocks are ignored. In such executions, high parallelism cannot be guaranteed but progress is. The result is the first implementation of a linearizable counter that is both nonblocking and achieves high parallelism in k-synchronous executions. The challenge in devising our algorithm was to provide high throughput in k-synchronous executions while guaranteeing progress in asynchronous executions where processes may fail-stop.

Our bounded-wait combining algorithm uses what we call *pseudo-transactions*. "Regular" transactions either have no effect or take effect atomically. In contrast, a pseudo-transaction is visible even before it terminates; however, it guarantees that write operations are performed in order without interruption from other pseudo-transactions. As we show, this allows our algorithm to achieve high-parallelism while maintaining correctness. Technically, pseudo-transactions are implemented by applying Greenwald's *two-handed-emulation* technique [9] *in parallel* on every combining-tree node and by ensuring that nodes' data is always consistent.

Bounded-wait combining can be easily adapted to work for any *combinable operation* [8, 17] and can thus be used to implement linearizable stacks and queues that are both nonblocking and highly parallel in k-synchronous executions. We are not aware of any other *deterministic* stack or queue algorithm to possess these properties. Hendler, Shavit, and Yerushalmi presented an elimination-based *randomized* linearizable stack algorithm that is both nonblocking and may achieve high parallelism [12]. Moir et al. used ideas similar to these of [12] to obtain a queue algorithm that possesses the same properties [18].

1.1 Model and Definitions

We consider a shared-memory system, in which a set of N asynchronous processes communicate by applying *read*, *write*, or *read-modify-write* operations to shared variables.

An *object* is an instance of an abstract data type. It is characterized by a domain of possible values and by a set of *operations* that provide the only means to manipulate it. An *implementation* of an object shared by a set of processes provides the following two parts: (1) a specific data-representation for the object from a set of shared variables that is available for the specific system, each of which is assigned an initial value; (2) algorithms for each process to apply each operation to the object being implemented. To distinguish between operations applied to the implemented object and operations applied to shared variables, we call the former *high-level operations* and the later *primitive operations*. 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 processes are in their initial states.

To apply their high-level operations, processes perform a sequence of *steps*. Steps taken by processes are of three kinds.

1. *Invocation*: this is an initiation of a *read*, *write*, or *read-modify-write* primitive operation to a shared variable. Once initiated, and until a response is received, we say that the primitive operation is *pending*. Each process may only have one pending primitive operation at a time. A process may perform some local computation before performing an invocation step.

2. *Response*: a process receives the response to a pending primitive operation.

3. *Stall*: a process with a pending operation is delayed due to contention with other processes that have simultaneous pending primitive operations to the same shared variable.

In other words, to execute a primitive operation, a process performs an invocation of the operation. It then executes 0 or more stall steps (depending on the number of primitive operations on the same shared variable that precede it) and, when the operation completes, performs a response step.

The concept of memory stalls was introduced by Dwork et al. to model the delays caused by contention in accessing shared memory [4]. Similarly to [4], we assume that if a process p has a pending operation on shared variable v, then p incurs a stall only if another process with a pending operation on v receives a response. We also assume that pending operations receive their responses in a first-in-first-out order of their invocations.

We note that, similarly to [4], our definitions imply that memory stalls are also caused by *read* operations and not just by contention caused by write or read-modify-write operations. In cache-coherent systems, however, it *is* possible for the same shared variable to be read concurrently by multiple processes accessing their local caches. Thus, an alternative, less strict, definition of stalls, in which multiple reads of the same base object complete without stalls, is also possible. For the analysis of our algorithm, we choose to apply the stricter definition, in which time must strictly increase between different accesses of the same shared variable. 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 (see formal definitions in Section 2). Hence, the throughput of an implementation (and, specifically, of our algorithm) can only *increase* if the alternative (less strict) definition of stalls were to be used.

An *execution fragment* is a (finite or infinite) sequence of steps. An *execution* is an execution fragment that starts from an initial configuration, in which processes take steps and change states, based on the responses they receive from the operations they perform, according to their algorithm. A process' state may change after a response step but does not change after an invocation step or a stall step.

An *operation instance* is an application of a specific high-level operation with specific arguments to a specific object made by a specific process. If the last invocation step of an operation instance Φ has been applied in an execution E and its response has been received in E, 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 invoked at least one primitive operation while performing Φ in E, but Φ does not complete in E. If p is active after E, then either it has a pending invocation or it has exactly one *enabled* primitive operation, which is the next primitive operation p will invoke when it is scheduled next. Note, that we assume that a process must complete one operation instance before it is allowed to start executing another.

Let E be an execution. We say that E is *k-synchronous* if, for any E_0, E_1, E' such that $E = E_0E_1E'$, and for any two distinct processes p and q, if p has an enabled operation after E_0 and E_1 contains $k+1$ invocation steps by q then E_1 contains at least one invocation step by p. In other words, E reflects a "k-fair" scheduling, in which the difference in the speed with which the enabled operations of any two processes are scheduled is at most k. We define the *contention level* of an execution E as the maximum number of consecutive stalls that are incurred by a process in E.

Compare-and-swap (CAS) is an example of a read-modify-write operation. *CAS(w, expected, new)*

writes the value *new* to shared variable w only if w's current value equals *expected* and, in this case, returns *true* to indicate success; otherwise, it returns *false* and does not change the value of w. *Double compare-and-swap* (DCAS) operates similarly on two shared variables.

The rest of the paper is organized as follows. We describe our metric for throughput in Section 2. Section 3 provides an overview of the algorithm. Section 4 describes the synchronous part of the algorithm in more detail. Section 5 describes the procedures that forward operation requests and dispatch responses. In Section 6, we describe the asynchronous part of the algorithm. We prove correctness, progress and throughput properties of our algorithms in Section 7. In Section 8, we analyze the throughput of prior-art counter algorithms. Section 9 concludes with a discussion of our results and mentions a few open questions.

2 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 takes memory contention into consideration while assigning identical times to steps by different processes that access different objects and may be executed concurrently. Let us first describe how times are assigned to the steps taken by a process to perform a primitive operation.

Let E be an execution and let α be the sequence of steps performed in E by some process q to apply some primitive operation ρ. α is composed of a step invoking ρ, followed by i stall steps, for some $i \geq 0$; if ρ completes in E, α terminates with a step returning ρ's response to q. Let t_0 be the time assigned to the invocation step. The j'th stall step, for $j \leq i$, is assigned time $t_0 + j$. The time assigned to the response step, if it exists, is identical to the time of the step that precedes it in α. Specifically, the above definition implies that, in the absence of contention, the invocation and response are assigned the same time; otherwise, the response is delayed for some $j > 0$ time units, where j is the number of stalls between the invocation and response.

As processes are sequential threads of execution, the times assigned to consecutive primitive operations applied by the same process must be strictly increasing. Similarly, and in accordance with our definition of stalls in Section 1.1, the times assigned to consecutive primitive operations applied to the same shared variable are also strictly increasing.

The subsequence of steps applied by process q in E is denoted by $E|q$. The subsequence of steps that access a shared variable v in E is denoted by $E|v$ (this includes stall steps caused by accessing v). We assign times to execution steps. Assigned times constitute a non-decreasing sequence of integers starting from 0. The following definition formalizes the assignment of time that we use.

Definition 1 *Let s be a step performed by process q that accesses variable v in execution E, let $E = E_1 s E_2$, and let s' denote the last step performed by q 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. *Otherwise (i.e. s is an invocation step),* time(E,s) *is defined to be maximum of the following numbers:*

- *the time assigned to the last step in E_1 (or 0 if E_1 is empty),*
- *the time assigned to q's last step in E_1 plus 1 (or 0 if $E_1|q$ is empty),*
- *the time assigned to the last step in E_1 that accesses v plus 1 (or 0 if $E_1|v$ is empty).*

We define the duration of a (non-empty) execution as the time assigned to its last step plus 1. We define the throughput of an execution E as the ratio between the number of operation instances that complete in E and E's duration.

Definition 2 *The* 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 step of E. The* throughput of a *(non-empty)* execution *E is defined to be $\texttt{completed}(E)/\texttt{time}(E)$.*

We use the above definitions in the next section to prove that our algorithm can achieve throughput of $\Omega(n/logn)$.

Based on Definition 2, we can now define the latency of an operation instance. In Section 7, we prove that the latency of operation instances in k-synchronous executions is $O(\log n)$ (Theorem 1).

Definition 3 *Let $E = E_1 s_f E_2 s_l E_3$ be an execution, where E_1, E_2 and E_3 are execution fragments, and let Φ be an operation instance that completes in E, whose first and last steps are s_f and s_l, respectively. We define the* latency *of Φ in E to be $time(E, s_l) - time(E, s_f) + 1$.*

3 An Overview of the BWC Algorithm

In this section, we provide a high-level description of the *bounded-wait combining* (henceforth, BWC) algorithm. The algorithm can be used to provide a linearizable, nonblocking and high-throughput implementation of any combinable operation [8, 17]. For the sake of presentation simplicity, however, we describe it in the context of implementing a shared counter. A shared counter is an object that holds an integer and supports the *fetch&increment* operation for incrementing the counter and returning its previous value atomically.

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. When the process wishes to request a value from the counter, it first accesses its leaf and then starts climbing the tree towards the root. If the process happens to run by itself, then it reaches the root, increments the value of the counter, and, eventually, descends down the tree to its leaf with a response - the previous value of the counter. The algorithm attempts to combine such requests.

A process that accesses a node in the tree attempts to lock that node. To prevent blocking, such a lock is respected by processes only as long as they "think" that the execution is k-synchronous. We call these locks *slocks* (for Synchronous locks). If a process calculates that it waited "too much" (we explain later how processes do that), it shifts to an asynchronous mode of operation, in which it no longer waits for other processes. This mode is described later. We continue now with the description of the synchronous modes of operation.

Consider a process q who managed to reach and capture the root's slock and assume the execution is k-synchronous. The slocks in the nodes along the branch from q's leaf to the root were all captured by q. Some other processes may have climbed until they reached this branch and have been waiting with their requests. The algorithm will combine the requests of all these processes with q's request. We say that all these requests are combined in the same *phase*. To help construct the phase, q does not yet increase the counter. Instead, q waits some additional time in the root to let even more processes climb from their leaves and arrive at a node along the branch locked by q. Process q then descends down its branch towards its leaf. Whenever it reaches a node along the branch where another process p is waiting, it leaves a sign for p that p

has been added to the phase. Let us call p an *offspring process* of q for that phase. Process q then continues descending towards its leaf. The offspring process p now acts similarly- it starts descending along its own branch and, on its way, adds waiting processes as its own offspring processes for that phase. Those offspring processes also act the same, recursively.

A descending process that reached its leaf starts climbing up again along the same branch. On the way, it waits for its offspring processes to deliver to it their requests (and the requests of their offsprings processes, recursively). If p is not the process that captured the root's slock, then it only climbs until it reaches the last node v whose slock it managed to capture on its way up. There it delivers its requests (its own, and those of its offspring processes), to be collected by the process that did manage to capture v's parent.

The single process q that captured the root's slock reaches the root again in this climb. It has collected all the requests of the processes that "got in time" to join this phase. It increases the counter by the number of collected requests and starts descending again towards its leaf. On its way down, it distributes request responses to its waiting offsprings and unlocks all the slocks it locked.

If some process p calculates that it waited "too much", then it shifts to an asynchronous mode. It no longer respects slocks. However, it cannot just go directly to the counter at the root and access it for obtaining a number, since this would violate linearizability. Moreover, this can even violate the semantics of a counter, since p's original request may be handled later by some other process who may still be in a synchronous mode. Thus, the counter is incremented *twice* per a single request of p. Instead, starting from its leaf, p climbs up the tree (ignoring slocks) while combining any requests it finds in nodes on the way up. It then descends again towards its leaf and distributes responses to waiting processes along its way. It repeats this operation until it finds a response to its original request in its leaf node. Note that, in an asynchronous mode, multiple processes may try to combine requests to (or distribute responses to) the same node simultaneously. This does not violate correctness, as our algorithm applies these operations as *pseudo transactions*. In fact, pseudo transactions are used also in the synchronous modes.

Pseudo transactions are discussed in detail in Section 3.1. For now, it suffices to say that they allow higher parallelism than "regular" transactions do since they allow inter-transaction reads. This is necessary for achieving high throughput in synchronous modes, as well as for guaranteeing progress in asynchronous modes.

We now give a detailed description of the algorithm. Let us start with describing the data structures it uses. 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 context of the procedures that use them.)

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* (*slock*). *Slock*s are stored in a structure-field by the same name. 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. Recall, that *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 4), 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 of $\mathcal{T}$, $\mathcal{T}_i$, 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 all the nodes on the path 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 perform the task of forwarding (and combining) these requests up $\mathcal{T}_i$ cooperatively. 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, at the same time, 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 4.

After waiting for a while in vain for some event to occur, a process q concludes 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 is no longer guaranteed. Progress *is* guaranteed, however, as long as some processes continue to take steps.

The asynchronous part of the BWC algorithm is implemented by the *AsynchFAI* procedure (see Section

---

constant LEFT=0, RIGHT=1, FREE=$\perp$
structure Range {**int** from, **int** till}, typedef RRQ **queue** of Ranges
structure Reqs {**int** direction, **int** num}, typdef REQQ **queue** of Reqs
structure Node {

Node$\star$ parent	Node$\star$ children[2]	**int** requests initially 0
int reqsTaken[2] initially {0,0}	REQQ pending initially EMPTY	RRQ responses initially EMPTY
boolean inPhase initially **false**	**boolean** collected initially **false**	**int** slock initially FREE
int phaseTop initially null		

}
Node$\star$ nodes[2n-1]

Figure 1: The structure of BWC combining-tree nodes.

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

Figure 2: The main procedure of the BWC algorithm.

6). 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 all 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 to each node from which requests have been forwarded. Process *q does not respect node* slock*s while traversing this path in both directions*. However, it releases the *slock* of every node it descends from. (This guarantees that if the system reaches quiescence and then becomes k-synchronous, 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. [1]

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 *SendResponses* combining procedures (see Section 5). The BWC algorithm allows multiple processes (in both synchronous and asynchronous modes) to apply these procedures to multiple 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 ensuring that each such procedure application either has no effect, or procedure statements are applied to n in order. Moreover, the data-sets accessed by combining procedures that are applied concurrently to different nodes are, in general, not disjoint. That is, a process owning a node u is responsible for combining (and "bringing" to u) the requests found at the children (say, v and w) of u. Note that the requests in one of the children, say w, may be updated by the process handling w. Hence, that process needs to access w, as well as accessing w's children, which can conflict with another process, and so forth. We could have solved those conflicts by using transactions. However, this might have prevented parallelism, since the process performing a transaction accessing u might have to wait for a process performing a transaction accessing w, who might have to wait also, and so on. Reduced parallelism will clearly reduce algorithm throughput.

We solved this problem by introducing the notion of pseudo transactions instead of using "regular" transactions. As described in Section 3.1, the implementation of pseudo transactsions required us to design the node data-structure so that procedures applied to different nodes never *write* to the same field. However, a process executing a combining procedure applied to a node v must still be able to *read* fields stored at the node v's parent or children. To permit high throughput, these reads must be allowed *even if other processes*

[1] Observe that it is possible that, while operating in the asynchronous mode, a process attempts to forward requests (propagate responses) from a node that has none.

apply combining procedures to these nodes concurrently. This is the main difference between transactions and pseudo transactions. We now describe the pseudo-transactions mechanism in more detail.

3.1 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 4 *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 that is applied as a pseudo-transaction.*

The following requirements must be met to ensure the correctness, liveness, and high-parallelism of the BWC algorithm.

1. **Combining correctness**: the combining procedures *FwdReqs* and *SendResponses* 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.

- We have designed node structure carefully so that applications of the *FwdReqs* or *SendResponses* combining procedures to different nodes never write to the same field (see Section 5 for more details). Let us explain why this is required.

  Consider the *FwdReqs* procedure that is applied to a node so as to forward requests to it from its children. A simple (but problematic) way of implementing it would be the following. Define a *requests* node field that stores the number of requests that have yet to be forwarded from the node up the tree. When a process applies *FwdReqs* to node n, it reads the *requests* fields of n's children. If their values are positive, the requests they represent are forwarded and combined by adding their values to n's *requests* field and by substracting them from the *request* fields of the children. Had the algorithm acted that way, concurrent applications of *FwdReqs* to n and n's parent might have resulted in simultaneous updates that attempt to simultaneously increase and decrease node n's *requests* field and algorithm correctness would have been compromised.

  To avoid concurrent writes, we interpret the number of the requests (not forwarded yet) in n to be the *difference* between the value of variables in n and in n's parent (rather than representing this value by a single field). Our implementation uses the size-2 *reqsTaken* array in addition to the *requests* field. Field *n.requests* stores the number of requests forwarded to n so far (or injected to it, if n is a leaf). Field *n.reqsTaken* stores the number of requests forwarded so far to n from each of its children (if n is not a leaf). To forward requests to n from a child m, *FwdReqs* reads *m.requests* and compares it with the value of *n.reqsTaken* corresponding to m. If these values differ, then the difference Δ is

computed and requests are forwarded to n by increasing both $n.requests$ and the entry of *n.reqsTaken* corresponding to m by Δ. Though concurrent procedures applied to different nodes never write to the same field, a procedure applied to one node may read from another node that is updated concurrently.

- We treat each node (in conjunction with the *FwdReqs* and *SendResponses* combining procedures) as a separate object. *We apply Greenwald's two-handed emulation to each of these objects separately* [9]. A detailed description of two-handed emulation is beyond the scope of this paper, and the reader is referred to [9]. 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.

  It is known that Greenwald's emulation is sequential and may result in contention linear in the number of participating processes. The BWC algorithm does not 'inherit' these problems, however. This is because each time our algorithm applies Greenwald's emulation in a k-synchronous execution, the application is to a node accessed by a small constant number of processes (as we prove for such executions, see Lemmas 3 and 16). Indeed, we use many applications of the emulation concurrently, but each is applied to a different node. In asynchronous executions, however, processes may apply operations to the same node concurrently and so high contention and low throughput may result.

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 can be applied as pseudo-transactions and requirement 1. above is satisfied.

Even if some applications of the procedures fail, this happens because the required instructions are performed by some other applications. That is, 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.

4 The Synchronous Modus Operandi

The synchronous part of the algorithm is implemented by the *SynchPhase* procedure. See pseudo-code in Figure 4 (some of the helper functions it uses appear in the appendix). This procedure executes iterations of the **do forever** block of line 3. Formally, we say that an *iteration* is a sequence of steps taken by a process that starts with the execution of line 3 and ends either with a **return** instruction (if this is the last iteration made by the operation) or just before line 3 is performed by the process again. In each iteration, the algorithm acts according to the value of the local variable *mode*. Variable *mode* stores a code representing the current synchronous mode. Figure 3 shows the transitions-diagram of the synchronous modes.

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 at q's (statically assigned) leaf. In some of the modes (*UP*, *FORWARD_REQUESTS* and *AWAIT_RESPONSES*), 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 waiting times, M denotes the maximum number of primitive operations (that is, read, write or CAS operations) applied by a process to shared variable while executing a single iteration of the *SynchPhase* procedure. Clearly, M is a constant number. We prove that this selection of waiting periods guarantees that no process shifts to an asynchronous mode in k-synchronous executions (see Lemma 16). 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 path to the root in order to join a phase. Initially, q decrements *timer* and, if it timeouts, concludes that the execution is asynchronous and returns the *ASYNCH* code (statements 11, 12). Otherwise, if q acquired the root node's *slock* in the previous iteration, 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 14 - 16). To climb from a non-root node to its parent m, q first verifies that m is free (statement 17), in which case it performs a *CAS* operation to try and acquire m's *slock* (statement 18). 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 20) which indicates whether or not m is a part of the current phase's subtree. If *m.inPhase* is set then q checks whether n was added to the phase subtree by m's owner (statement 21). If both conditions hold, then q managed to join the current phase and all the nodes along the path from its leaf to n will become part of that phase's subtree. In this case, q shifts to the *FREEZE_NODES* mode and stores a pointer to its highest node along this path (statements 22, 24). Before shifting to the *FREEZE_NODES* mode, q initializes the timer to the maximum number of iterations that should be performed by it in both the *FREEZE_NODES* mode and the following *FORWARD_REQUESTS* mode.

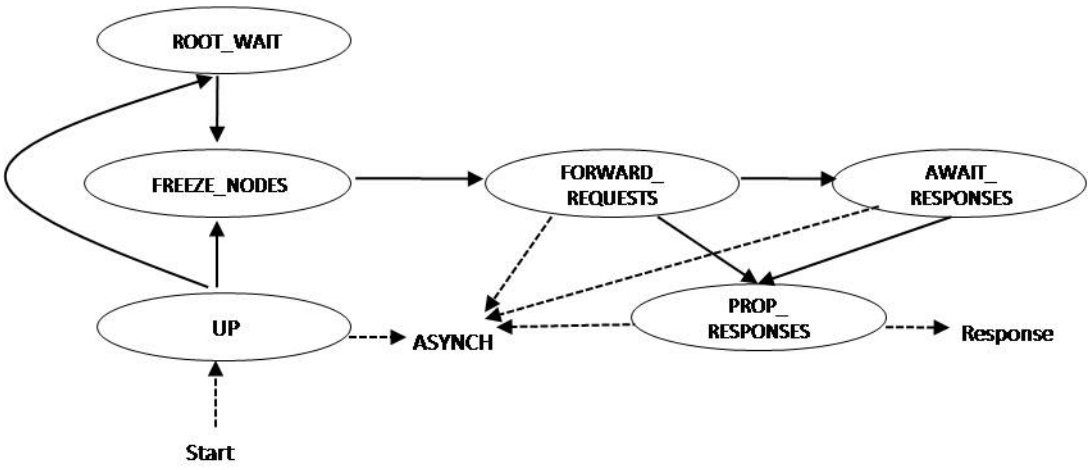

Figure 3: Mode transitions in synchronous phases. Solid arrows point to synchronous phases, dashed arrows exit synchronous mode.

```
SynchPhase()
1  Node⋆ n=Leaf(myID), int phaseTop, int mode=UP, int timer=14M(log N + 1)(k + 3)^2
2  boolean chCollected[2]={false,false}
3  do forever
4    switch(mode)
5      case UP: UpMode()
6      case ROOT_WAIT: RootWait()
7      case FREEZE_NODES: FreezeNodes()
8      case FORWARD_REQUESTS: ForwardRequests()
9      case AWAIT_RESPONSES: AwaitResponses()
10     case PROP_RESPONSES: PropResponses() od

UpMode()
11 timer=timer-1
12 if (timer=0) return ASYNCH
   // If captured the root node
13 if (ROOT(n))
     // This is the highest node whose slock I captured
14   phaseTop=n
15   timer=2M(log N + 1)(k + 3)
16   mode=ROOT_WAIT
   // Else if parent node is free
17 else if (n.parent.slock=FREE)
     // Try to capture parent's slock
18   if (CAS(n.parent.slock, FREE, myID)=SUCCESS)
19     n=n.parent
   // Else if parent node joined phase
20 else if (n.parent.inPhase)
     // If this node joined phase
21   if (n.inPhase)
       // This is the highest node whose slock I captured
22     phaseTop=n
23     timer=(3M + 2)(log N + 1)(k + 2)
24     mode=FREEZE_NODES

RootWait()
25 if (timer > 0)
     // Wait at the root
26   read n.slock
27   timer=timer-1
28 else
29   timer=(3M + 2)(log N + 1)(k + 2)
30   mode=FREEZE_NODES

FreezeNodes()
31 timer=timer-1
32 if (n=Leaf(myID))
     // Inject request at leaf node
33   n.requests=n.requests+1
34   injected=true
35   mode=FORWARD_REQUESTS
36 else
     // Find which child to descend to
37   int whichChild=LeafDir(n, myID)
38   Node ⋆ch=n.children[whichChild]
     // If child node is captured, freeze it
39   if (n.children[1-whichChild].slock ≠ FREE)
40     n.children[1-whichChild].inPhase=true
     // Descend to child node and freeze it
41   n=ch
42   n.inPhase=true
```

Figure 4: Pseudo-code for the synchronous part of the algorithm, part 1.

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 primitive operation. Finally, q initializes the timer to the maximum number of iterations that should be performed by it in both the *FREEZE_NODES* mode and the following *FORWARD_REQUESTS* mode and then shifts to the *FREEZE_NODES* mode (statements 25-30).

FREEZE_NODES: in this mode, q freezes the nodes along the path from *phaseTop* to its leaf and some children of these nodes. Freezing a node adds it to the phase's subtree. When q gets to its leaf, it injects

```
ForwardRequests()
43  if (n ≠ Leaf(myID))
      // Forward request from both subtrees
44    (for i=LEFT; i≤RIGHT; i++)
        // If child node in phase and its requests
        // not yet forwarded to it
45      if (n.children[i].inPhase ∧¬ chCollected[i])
          // If all requests from the child node's sub-tree
          // were forwarded to it
46        if (n.children[i].collected)
            // Forward requests from child node
47          FwdReqs(n, i)
            // Mark that this child node was handled
48          chCollected[i]=true
49        else
50          timer = timer -1
51          if (timer = 0) return ASYNCH
52          else continue do-forever
    // Mark that requests from this node's
    // sub-tree were forwarded to it
53  n.collected=true
54  if (n ≠ phaseTop)
55    n=n.parent
56  else if (ROOT(n))
57    mode=PROP_RESPONSES
58  else
59    timer=5M(log N + 1)(k + 3)
60    mode=AWAIT_RESPONSES

AwaitResponses()
    // If responses were propagated to this node
61  if (¬ Empty(n.responses))
      // Start the process of propagating them to children
62    mode=PROP_RESPONSES
63  else
64    timer=timer-1
65    if (timer=0) return ASYNCH

PropResponses()
66  if (n = Leaf(myID))
67    if (Empty(n.responses))
        // This execution is not k-synchronous,
        // shift to the asynchronous mode
68      return ASYNCH
      // There is a response at the child node
      // return it as this operation's response
69    Range r=DeqRange(n.responses,1)
70    return r.from
    // Propagate this node's responses to its children
71  SendResponses(n)
    // This node now leaves phase and is freed
72  n.inPhase=false, n.collected=false, n.slock=FREE
    // Descend towards this process' leaf node
73  n=n.children[LeafDir(n,myID)]
```

Figure 5: Pseudo-code for the synchronous part of the algorithm, part 2.

a single request, sets the iterations counter, sets a flag indicating that a request was injected, and shifts to the *FORWARD_REQUESTS* mode (statements 33-35). For each internal node n along the path, q uses the *LeafDir* macro to determined which of n's children is owned by q. Given an internal node n and an id of a process whose leaf m is in the subtree rooted at n, this macro returns *LEFT* or *RIGHT* (statement 37) according to whether m is in the left or right subtree of n. If the child of n not owned by q is not free, q sets that child's *inPhase* flag. Process q then descends to n's child on the path back to q's leaf (statements 39-41). Regardless of whether n is a leaf or an internal node, its *inPhase* flag is set (statement 42).

FORWARD_REQUESTS: in this mode, q forwards and combines requests along the path starting with its leaf and ending with $q.topPhase$. For each internal node n along this path (if any), q checks for each child ch of n whether ch is in the current phase's subtree and whether requests need to be forwarded from it (statements 44-45). 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 flag *chCollected* corresponding to ch in order to not repeat this work (statements

46-48). When n is the root node, the *FwdReqs* procedure increases the value of the counter by the number of forwarded requests. It also adds the corresponding range of responses to the root's responses-queue (see Section 5). 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 50 - 52). 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_RESPONSES* mode or the *AWAIT_RESPONSES* mode, depending on whether or not it is the current phase's initiator (statements 53-60).

AWAIT_RESPONSES: In this mode, q, who is not the initiator of the current phase, waits for the owner of n's parent to propagate responses to n. Responses are propagated to n when the process that owns n's parent enqueues them into the $n.responses$ queue. This is a producer/consumer queue storing response ranges that were received at n and not yet propagated to its children. See Section 5 for more details. If and when q finds that the responses queue is not empty, q shifts to the *PROP_RESPONSES* mode (statements 61, 62). If *timer* expires, q returns the *ASYNCH* code (statement 65).

PROP_RESPONSES: 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 71-73). Eventually, q descends down to its leaf. If a response awaits it there (possibly propagated by q itself), it is returned as the response of *SynchPhase* procedure. As we prove in Lemma 15, this is guaranteed to happen in a k-synchronous execution. Otherwise, it might be that there is no response in q's leaf. In this case, q returns the *ASYNCH* code (statements 66-68).

5 The Combining Process

The combining process is implemented by the *FwdReqs* and *SendResponses* procedures. The pseudo-code of these procedures appears in Figure 6. As discussed in Section 3.1, the code actually performed is a transformation of the code shown in Figure 6 according to Greenwald's two-handed emulation technique, as indicated by the *two-handed-emulate* attribute. Recall that this, together with the fact that processes serving different nodes never write to the same variables, guarantee that these procedure execute as pseudo-transactions (see Section 3.1).

The *FwdReqs* procedure forwards requests from a child node to its parent. The *SendResponses* procedure dispatches responses from a parent node to its children. The two procedures use the following node fields.

- *requests*: 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 already 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. The *pending* queue allows a process serving node n to send responses in the order in which the corresponding requests were received. As we prove, 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. Please refer to Appendix A for the procedures that access and manipulate the *pending* queue.

- *responses*: 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. The *responses* queue contains at most a single range in k-synchronous executions. In asynchronous executions it contains at most N ranges. Please refer to Appendix A for the procedures that access and manipulate the *responses* queue.

We now describe the pseudo-code of the two combining procedures, which are performed as pseudo-transactions. In the following description, q is the process executing the code.

- *FwdReqs*: this procedure receives two parameters. A node pointer n to the node to which requests need to be forwarded, and an integer, *direction*, 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 *direction* 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.

  Forwarding the requests is implemented as follows. Process q first increases n's *requests* field by *delta* (statement 5) to indicate that *delta* additional requests were forwarded to n. It then increases n's *reqsTaken* entry corresponding to m by *delta* (statement 4). 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* requests were now taken from m.

  Note, that the fact that requests were taken from m is not recorded by decreasing the number of requests at m. In fact, no field of m is written at all. Rather, this fact is recorded by increasing the value of the *reqsTaken* field at n which corresponds to m. This design prevents a situation in which combining procedures applied to different nodes write to the same field. As explained in Section 3.1, this allows us to maintain the semantics of pseudo-transactions. There may be multiple processes performing *FwdReqs* in parallel for the same node (in the asynchronous mode). In this case, the pseudo transactions ensure that it is performed at least once, and that its statements are executed sequentially.

  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.requests* is increased by statement 5. In that case, q adds the corresponding range of counter values to n's queue of response ranges immediately (statements 7-8).

- *SendResponses*: this 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. This is enough to propagate the responses for all the requests forwarded to a node in a synchronous phase. If the responses queue is not empty, the length of the first range in the queue 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 the *reqsLen* local variable (statements 13, 14). The minimum of these two values is stored to the *numToSend* local variable (statement 15). This number of responses is now sent in the direction specified by the *direction* field of the first requests entry (statements 16, 18). Finally, *numToSend* responses and requests are removed from the *n.responses* and *n.pending* queues, respectively.

```
two-handed-emulate FwdReqs(NODE* n, int direction)
1   Node* child = n.children[direction]
    // Compute the number of requests not yet forwarded from child
2   int delta=child.requests - n.reqsTaken[direction]
3   if (delta > 0)
    // Forward requests
4     n.reqsTaken[direction]=n.reqsTaken[direction]+delta
5     n.requests=n.requests+delta
    // Make sure responses are returned in order
6     ENQ_REQS(n.pending, <direction,delta>)
7     if (ROOT(n))
      // Requests generate responses at the root
8       EnqRange(n.responses, n.requests-delta+1, delta)

two-handed-emulate SendResponses(NODE* n)
9   do twice
10    if (¬ Empty(n.responses))
        // Extract first responses range
11      Range firstResp=FirstRange(n.responses)
12      int respsLen=RangeLen(firstResp)
        // Send responses to oldest un-services requests
13      Reqs firstReqs=FIRST_REQS(n.pending)
14      int reqsNum=firstReqs.num
15      int numToSend=min(respLen, reqsNum)
16      int dirToSend=firstReqs.direction
17      DeqRange(n.responses, numToSend)
        // Enqueue responses in child's queue
18      EnqRange(n.children[dirToSend].responses, firstResp.start, numToSend)
19      DEQ_REQS(n.pending, numToSend)
20  od
two-handed-emulate FwdReqsAndSendResp(NODE* n)
21  FwdReqs(n,LEFT)
22  FwdReqs(n,RIGHT)
23  SendResponses(n)
```

Figure 6: Pseudo-code for the combining procedures

For presentation simplicity, the *requests* and *reqsTaken* fields used in the pseudo-code of Figure 6 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 (i.e., the *requests* field of a child node and the corresponding *reqsTaken* entry of its parent) be maintained. This is the number of requests that have arrived at the child and were not yet forwarded to the parent. Clearly, this is bounded by the maximum number of concurrent operations on the counter, which is N. Hence the code can be modified to use bounded *requests* and *reqsTaken* fields that count modulo $2N + 1$.

```
AsynchFAI()
1   Node *n=Leaf(myID)
2   int mode=ASYNCH_UP
    // Inject a request, if haven't done so in synchronous mode
3   if (¬ injected)
4     n.requests=n.requests+1
5   do
      // In asynchronous mode, free slocks but disregard them otherwise
6     n.slock=FREE
7     switch(mode)
8       case ASYNCH_UP:
9         if ¬LEAF(n)
            // Both forward requests and send responses
10          FwdReqsAndSendResp(n)
11        if (ROOT(n))
12          mode=ASYNCH_DOWN
13        else
14          n=n.parent
15      case ASYNCH_DOWN:
16        if (n = Leaf(myID))
17          Range r=DeqRange(responses)
18          if (RangeLen(r) > 0)
19            return r.from
20          else
21            mode=ASYNCH_UP
22        else
            // Both forward requests and send responses
23          FwdReqsAndSendResp(n)
24          n=n.children[LeafDir(n,myID)]
25        break
26  forever
```

Figure 7: Pseudo-code for the asynchronous part of the BWC algorithm.

When a process operates in an asynchronous modus operandi, it calls the *FwdReqsAndSendResp* procedure, which executes as a two-handed-emulation procedure. This procedure, called only for non-leaf nodes, both forwards requests to a node from both its children and propagates responses from it to its children.

6 The Asynchronous Modus Operandi

After waiting for a while in vain for some event to occur, a process concludes that the execution is not k-synchronous. If and when that occurs, the process falls back on an asynchronous mode. The algorithm performed by processes while operating in asynchronous modes is implemented by the *AsynchFAI* procedure (see pseudo-code in Figure 7).

A process q, executing *AsynchFAI*, iterates in the loop of statements 5 - 26. Process q starts at its leaf

in the *ASYNCH_UP* mode (statements 1 - 2). If a request has not been injected by q while operating in a synchronous mode, it is injected now (statements 3 - 4). Process q then climbs up the tree. For each node n it climbs to, q calls the *FwdReqsAndSendResp* procedure (statement 10), which *both* forwards requests from n'th children and propagates responses from n to its children. When it reaches the root, q shifts to the *ASYNCH_DOWN* mode (statements 11 - 12). While in the *ASYNCH_DOWN* mode, q descends down the path from the root to its leaf. Whenever it descends to a non-leaf node, q forwards requests to that node and propagates responses from it (statement 23). When q descends to its leaf, it checks whether a response awaits it there, in which case it returns this response (statements 16 - 19); otherwise, q shifts to the *ASYNCH_UP* mode (statement 21) and starts a new iteration of the loop.

7 Correctness and Complexity

This section is organized as follows. Definitions required for the proofs are provided in Section 7.1. Section 7.2 establishes that the BWC algorithm provides high throughput in k-synchronous executions. In Section 7.3 we prove that the algorithm is nonblocking and linearizable.

7.1 Definitions

We say that process p is in an *asynchronous mode* in configuration C if p has an active invocation of *Fetch&Inc* in C that received an *ASYNCH* response from the call of *SynchPhase*; otherwise we say that p is in a *synchronous mode* in C. We say that a configuration C is an *asynchronous configuration* if at least one of the processes is in an asynchronous mode in C, otherwise we say that C is a *synchronous configuration*. We say that an execution E is *BWC-synchronous* if all the configurations reached by prefixes of E are synchronous. Otherwise we say that E is *BWC-asynchronous*. We say that an execution is *k-BWC-synchronous* if it is both k-synchronous and *BWC-synchronous*. We prove that every k-synchronous execution is also *k-BWC-synchronous*. In other words, we prove that no process is ever in an asynchronous mode in k-synchronous executions of the algorithm.

Let p be a process and n a node. We say that *p owns n in configuration C* if *n.slock=p* holds in C or if n is the leaf node of p. If a node is not owned by any process in C (i.e. *n.slock=FREE* holds in C), we say that *n is free in C*. We say that an application ξ of a primitive operation *frees* a node n if n is owned by some process in the configuration in which ξ is applied and free in the configuration immediately following the corresponding response step.

Let C be a configuration in which process p executes the *SynchPhase* procedure or a procedure called by it (either directly or indirectly) and let n be a node. We say that *p serves n in C* if variable *SynchPhase.n* stores a pointer to n in C.

In the following description, statement numbers refer to the *SynchPhase* procedure unless stated otherwise. We say that *phase i begins* when statement 30 is performed for the i'th time in the execution (phase numbering starts from 0). The process that performs statement 30 for the i'th time is called the *initiator of phase i*. We say that process *p joins a phase at time t* if p reads the value *true* from *n.parent.inPhase* at time t (in statement 20). The phase p joins is the phase of the process who wrote the value *true* that p reads at time t in *n.parent.inPhase*. We also say that process p joins a phase at time t, if p initiates that phase at time t. We say that *p leaves phase i at time t* if, after joining phase i, p returns at time t. (A process may leave a phase by returning the **ASYNCH** code in statements 51, 65 or 68, or by returning a response number in

statement 70). We say that a process p *participates in phase j at time t* if p joins phase j at time t_0 and leaves it at time t_1 for $t_0 \leq t \leq t_1$. We define the *duration of a phase* to be the time that elapses from the time it is initiated until the time when the last process leaves it. The *active phase* in configuration C is defined to be i, if the initiator of phase i still participates in phase i and has not yet freed the root.

We let M denote the maximum number of primitive operations applied by a process as it performs a single iteration of the do-forever loop of *SynchPhase*. We note that M is a constant independent of N and k

Let E be an execution. An *E-interval* is a period of time during the execution of E. Formally, for non-negative integers t_0 and $t_1 \geq t_0$, $[t_0, t_1]$ is an E-interval if and only if $t_1 \leq time(E)$. A process p is *active throughout an E-interval* $[t_0, t_1]$ if p is in midst of performing the *Fetch&Inc* procedure after every prefix E' of E such that $t_0 \leq time(E') \leq t_1$. The *duration of an E-interval* $[t_0, t_1]$ is defined to be $time(E, t_1) + 1 - time(E, t_0)$.

7.2 Correctness and Throughput in k-Synchronous Executions

In this section we prove the correctness of the BWC algorithm. We also prove that the algorithm possesses high-throughput in k-synchronous executions. *In the proofs that follow, E designates a k-BWC-synchronous execution*. For simplicity of presentation, and without loss of generality, we assume in the following that N is an integral power of 2. In the following proofs, statement numbers refer to the *SynchPhase* procedure, unless stated otherwise.

The high-level structure of our correctness proofs is as follows. Lemma 3 (which uses Lemmas 1 and 2) proves that the contention level of any k-BWC-synchronous execution is at most 2. Lemma 16 is a key lemma where we prove that a k-synchronous execution is also a k-BWC-synchronous execution. In other words, as long as the execution is k-synchronous, processes always operate in synchronous modes. Towards proving that, Lemmas 5 - 12 prove that the waiting times with which the *timer* is set in the code are such that the timer never expires in k-synchronous executions (with the single exception of the intentional waiting period at the root). These proofs use technical Lemmas 4 and 5. Based on Lemmas 3 and 16, it is easily shown in Lemma 17 that the number of primitive operations executed during a single instance of the *SynchPhase* procedure is $O(\log n)$. Our bounds on latency and throughput (Theorems 1 and 2) then follow easily.

Lemma 1 *A node owned by process p can only be freed by an application of a primitive operation by p.*

Proof: Since E is k-BWC-synchronous, processes do not execute the *ASynchFAI* procedure. Therefore, we should only consider the *SynchPhase* procedure. Clearly from the code, a process can become an owner of a node only by successfully performing the *CAS* operation of statement 18. After statement 18 is performed and until n is freed, *n.slock=p* holds and thus other processes will fail the *CAS* of statement 18. The *slock* field can be set to *FREE* only by statement 72. From the code, this statement can only be performed by process p (when it is in the *PROP_RESPONSES* mode). ■

Lemma 2 *In E, a process p only serves nodes owned by it. Moreover, when p serves node n, all the nodes on the path from p's leaf to n are owned by p.*

Proof: When a process first calls *SynchPhase*, it starts by serving its leaf node (which it owns by definition). A process p, serving node n in E, can start serving a different node only in the following cases.

- p climbs to n's parent (statement 19) when in the *UP* mode. The success of the *CAS* of statement 18 guarantees that p owns n's parent. From Lemma 1, all the nodes in the path from p's leaf up to n's parent are still owned by p, as, from Lemma 1 and the code, they can only be freed later when p enters the *PROP_RESPONSES* mode.

- p descends from n to its child down the path to its leaf in the *FREEZE_NODES* mode (statement 41). From Lemma 1, all the nodes in the path from p's leaf up to n are still owned by p.

- p climbs up to n's parent when in the *FORWARD_REQUESTS* mode (statement 55). From the test of statement 54, p can only climb up to nodes it served in the *UP* mode of the current phase. Thus, from Lemma 1, and as these nodes can only be freed by p when it enters the *PROP_RESPONSES* mode later, n's parent is still owned by p.

- p descends from n to its child down the path to its leaf in the *PROP_RESPONSES* mode (statement 73). All the nodes in p'th path from *phaseTop* down to p's leaf are owned by it as it shifts to the *PROP_RESPONSES* mode. Thus, from Lemma 1, p descends to a node it owns.

■

Lemma 3 *The contention level of E is at most* 2.

Proof: Let n be a node. As E is BWC-synchronous, no process ever calls the *ASynchFAI* procedure in E. Also, from Lemmas 1, 2, at any given time there is at most a single process that serves a node. The following possibilities exist.

- n is leaf. Let p be the process to which n is statically assigned, and let q be the process to which n's sibling, denoted m, is statically assigned. Nodes n and m are always owned by p and q, respectively. Thus, from the code, n can be accessed only by p and by q. It follows that the maximum number of stalls incurred by any invocation step applied to n is 1.

- n is an internal node. Assume that an invocation step s is applied to n when it is free. Then s can be applied either by the process that owns n's parent, or by a process that owns one of n's children. Moreover, none of these processes can free its node or capture n before it receives a response for its invocation step. It follows that the maximum number of stalls incurred by s is 2.

- Otherwise, s is applied when n is an internal node owned by some process p. From Lemma 2, p also owns one of n's children, denoted m. Thus, n can be accessed only by p, by the process that owns m's sibling (if any) and by the process that owns n's parent (if any). Since none of these processes can free its node before it receives a response for its invocation step, it follows that the maximum number of stalls incurred by s is 2.

■

Lemma 4 *Let T be an E-interval, p be a process that is active all throughout T, and $q \neq p$ be a process. Then p applies during T at least one invocation step every $k + 3$ invocation steps of q.*

Proof: From Lemma 3, each invocation step by p incurs at most 2 stalls. From the k-synchrony of E, an enabled step of p is scheduled before $k + 1$ invocations of q are scheduled. The lemma follows. ■

Lemma 5 *Let T be an E-interval, p be a process that is active all throughout T, and $q \neq p$ be a process. Then, if p applies l primitive operations during T, q applies at most $(l+1)(k+2)$ primitive operations during T.*

Proof: From Lemma 4, q applies at most $k+2$ primitive operations between any two consecutive invocation steps of p. In addition, q may apply $k+2$ primitives before p's first invocation and after its last invocation. ■

Lemma 6 *Let T be an E-interval that starts when some process q shifts to the* ROOT_WAIT *mode (in statement 16) and ends when q shifts to the* FREEZE_NODES *mode (in statement 30). Also, let p be a process that is active all throughout T. Then, the total number of primitive operations applied by p during T is at least $2M \log N$ and at most $(2M(k+3)\log N + 1)(k+2)$.*

Proof: From Lemma 1, q owns the root as long as it stays in the *ROOT_WAIT* mode. In statement 15, q sets the number of iterations for waiting at the root to $2M(k+3)\log N$ and this number is decremented in each iteration by statement 27. Thus, q remains in the *ROOT_WAIT* mode for $2M(k+3)\log N$ iterations. In each such iteration, q applies a single primitive operation in statement 26. Hence, q applies exactly $2M(k+3)\log N$ primitive operations during T. The lower and upper bound on the number of primitive operations applied by p now follow from Lemmas 4 and 5, respectively. ■

Definition 5 *Let q be a process that executes some operation instance Φ in a k-synchronous execution. We partition the time interval during which Φ is performed to time intervals that correspond to the synchronous phases as follows.*

- *Assume q enters the* UP *mode (in statement 1) at time t_s and exits the* UP *mode (in statement 16 or 24) at time t_e. We call $T = [t_s, t_e]$ an* UP interval *of q.*
- *Assume q enters the* ROOT_WAIT *mode (in statement 16) at time t_s and exits the* ROOT_WAIT *mode (in statement 30) at time t_e. We call $T = [t_s, t_e]$ a* ROOT_WAIT interval *of q.*
- *Assume q enters the* FREEZE_NODES *mode (in statement 24 or 30) at time t_s and exits the* FREEZE_NODES *mode (in statement 35) at time t_e. We call $T = [t_s, t_e]$ a* FREEZE_NODES interval *of q.*
- *Assume q enters the* FORWARD_REQUESTS *mode (in statement 35) at time t_s and exits the* FORWARD_REQUESTS *mode (in statement 57 or 60) at time t_e. We call $T = [t_s, t_e]$ a* FORWARD_REQUESTS interval *of q.*
- *Assume q enters the* AWAIT_RESPONSES *mode (in statement 60) at time t_s and exits the* AWAIT_RESPONSES *mode (in statement 62) at time t_e. We call $T = [t_s, t_e]$ an* AWAIT_RESPONSES interval *of q.*
- *Assume q enters the* PROP_RESPONSES *mode (in statement 57 or 62) at time t_s and exits the* PROP_RESPONSES *mode (in statement 70) at time t_e. We call $T = [t_s, t_e]$ a* PROP_RESPONSES interval *of q.*

Lemma 7 *Let q be a process with a* FREEZE_NODES *interval $[t_0, t_1]$ and an (immediately following)* FORWARD_REQUESTS *interval $[t_1, t_2]$ that are contained in E, let n denote q's* phaseTop *node during T and let h denote n's height. Then no process applies more than $(3M+2)(h+1)(k+2)$ primitive operations during $T = [t_0, t_2]$.*

Proof: We prove the claim by induction on h. The base case is when $h = 0$, implying that n is q's leaf node. In this case, q shifts to the *FORWARD_REQUESTS* mode (in statement 35), in the first iteration after it shifts to the *FREEZE_NODES* mode in statement 24. Then, in the immediately following iteration, q shifts to the *AWAIT_RESPONSES* mode. It follows that q applies at most $3M$ steps in T. Hence, from Lemma 5, no process applies more than $(3M + 1)(k + 2)$ steps during T.

For $h > 0$, Assume the claim holds for $h - 1$ and prove for h. We say that a node m along the path from n to q's leaf is *potentially delaying*, if m has a child whose *inPhase* flag is set by q in statement 40. If there are no potentially delaying nodes along the path, then the claim follows easily, since then q never has to wait in statements 45-52 while in the *FORWARD_REQUESTS* mode. Assume otherwise, and let $m_1, \ldots, m_i$ denote the potentially delaying nodes along q's path, ordered in increasing height. Also, let h_j denote m_j's height and let l_j denote m_j's child whose *inPhase* flag is set by q in statement 40, for $j = 1, \ldots, i$. For the induction step proof, we need the following claim.

Claim 1 *for $1 \leq j \leq i$, Let t_j be the time when q descends to node m_j in the* PHASE_FREEZE *mode and let t'_j be the time when q subsequently sets m_j's collected flag in statement 53. Then no process applies more than $(3M + 2)(h_j + 1)(k + 2)$ steps during $[t_j, t'_j]$.*

Proof: We prove the claim by induction. The base case is when $j = 1$. Recall that m_1 is the lowest potentially delaying node along q's path, hence q need not wait when it ascends from its leaf node to m_1 in the *FORWARD_REQUESTS* mode. It follows that, after t_1, q performs less than $2h_1+1$ full iterations before it reaches m_1 again, in the *FORWARD_REQUESTS* mode. Let t_1^* denote the time when this occurs. From Lemma 5, no process applies more than $(2M(h_1 + 1) + 1)(k + 2)$ steps during $[t_1, t_1^*]$. Let t_1^{**} denote the time when the process owning l_1 in the current phase shifts to the *AWAIT_RESPONSES* mode (while serving l_1). From induction hypothesis applied to Lemma 7, no process applies more than $(3M+2)(h_1+1)(k+2)$ steps during $[t_1, t_1^{**}]$. Since $t'_1 = max(t_1^*, t_1^{**})$, the claim follows.

Assume the claim holds for $j < i$ and prove for $j + 1$. Process q applies at most $M(h_{j+1} - h_j)$ steps when it descends from m_{j+1} to m_j in the *NODES_FREEZE* mode. The same claim applies when q ascends from m_j to m_{j+1} in the *FORWARD_REQUESTS* mode. From Lemma 5, no process applies more than $(2M(h_{j+1} - h_j) + 1)(k + 2)$ steps during these two intervals. From induction hypothesis applied to Claim 1, no process applies more than $(3M + 2)(h_j + 1)(k + 2)$ steps starting from when q descends to m_j in the *NODES_FREEZE* mode and until it ascends from m_j in the *FORWARD_REQUESTS* mode. The claim follows.

■

We can now complete the induction proof of Lemma 7. Process q applies at most $h - h_i$ steps as it descends from n to m_i in the *PHASE_FREEZE* mode. The same claim applies when q ascends from m_i to n in the *FORWARD_REQUESTS* mode. From Lemma 5, no process applies more than $(2M(h - h_i) + 1)(k + 2)$ steps during these two intervals. Using the induction hypothesis of Lemma 7, we get from Claim 1 (instantiated with $j = i$) that no process applies more than $(3M + 2)(h_i + 1)(k + 2)$ steps starting when q descends to m_i in the *PHASE_FREEZE* mode and until it sets m_i's *collected* flag. The lemma follows.

■

Lemma 8 *Let $T = [t_s, t_e]$ be an* AWAIT_RESPONSES *interval of q. Then no process applies more than $5M(\log N + 1)(k + 2)$ events during T.*

Proof: Let h be the height of *q.phaseTop*, let m denote the parent node of *q.phaseTop*, and let p be the process that owns m. (Note, that the phase initiator does not enter the *AWAIT_RESPONSES* mode and so *m* and *p* do exist.) Denote by t_0 the time when p shifts to the *FREEZE_NODES* mode of the current phase. Clearly, from the code of the *FreezeNodes* procedure, p shifts to the *FREEZE_NODES* mode of the current phase before q does. It follows, that $t_0 < t_s$. We denote $S = [t_0, t_e]$ and we let r denote the process that applies the maximum number of primitive operations, l, during S. We prove the following inequality by reverse induction on h.

$$l \leq (3M + 2)(\log N + 1)(k + 2) + (M + 1)h(k + 2). \tag{1}$$

The base case is when $h = \log N - 1$, i.e. when *q.phaseTop* is a child of the root. In this case, p is the phase initiator. Let l_1 denote the number of primitive operations applied by r starting from t_0 and until p shifts from the *FORWARD_REQUESTS* mode to the *PROP_RESPONSES* mode in statement 57. From Lemma 7 we get:

$$l_1 \leq (3M + 2)(\log N + 1)(k + 2). \tag{2}$$

Let l_2 denote the number of primitive operations performed by r starting when p shifts to the *PROP_RESPONSES* mode and before q exits the *AWAIT_RESPONSES* mode. Process p sends responses to the root's children in its first iteration in the *PROP_RESPONSES* mode (statement 71). During that iteration, p performs at most M primitive operations. Then q performs at most one iteration in the *AWAIT_RESPONSES* mode before it shifts to the *PROP_RESPONSES* mode. During this iteration, q performs at most M primitive operations. From Lemma 5, we get:

$$l_2 \leq (M + 1)(k + 2). \tag{3}$$

Combining Inequalities 2 and 3 proves Equation 1 for the base case.

For the induction step, assume Inequality 1 holds for $\log N - 1 \geq h \geq j > 0$ and prove for $h = j - 1$. Let m' denote the grandparent node of *q.phaseTop* and let p' denote the process that owns m'. Also let S' denote the time interval that starts when p' enters the *FREEZE_NODES* mode and ends when p exits the *AWAIT_RESPONSES* mode. From the induction hypothesis, r performs at most $(3M + 2)(\log N + 1)(k + 2) + (M + 1)(h - 1)(k + 2)$ primitive operations during S'. Let l_2 denote the number of primitive operations performed by r starting when p shifts to the *PROP_RESPONSES* mode and ending when q exits the *AWAIT_RESPONSES* mode. Inequality 3 can be shown to hold also here by using exactly the same proof used for the base case. Since $S \subset S'$, this establishes the proof of Inequality 1. To conclude the proof of the lemma, note that $T \subseteq S$ and that $h \leq \log N - 1$. ■

Lemma 9 *Let $T = [t_s, t_e]$ be a* PROP_RESPONSES *interval of q. Then, (1) q applies at most $M(logN+1)$ events during T, and (2) no process applies more than $(M(\log N + 1) + 1)(k + 2)$ events during T.*

Proof: From the condition of statement 54, *q.n=q.phaseTop* holds at time t_s. Let h be the height of the node pointed at by $q.phaseTop$ at that time. Then, q performs exactly $h+1$ iterations of the do-forever loop of *SynchPhase*, as it descends down from *q.phaseTop* to its leaf node. In each of these iterations q applies at most M events. Thus, it applies at most $M(h+1) \leq M(\log N + 1)$ events while in this mode. The second claim follows from Lemma 5. ■

Lemma 10 *Let q be a process and let T be an E-interval during which q participates in some phase i. Then, the maximum number of events executed by any process during T is $10M(\log N + 1)(k+3)$.*

Proof: Immediate from Lemmas 7-9. ■

Lemma 11 *Let q be a process and let $[t_0, t_1]$ be a* ROOT_WAIT *interval of q. Also, let $p \neq q$ be a process that is in the* UP *mode in t_0. Then, at time t_1, p serves a node n such that the path from n to the root consists of nodes owned by processes in the* UP *mode.*

Proof: Before shifting to the *ROOT_WAIT* mode, q sets its timer to $2M(\log N + 1)(k+3)$ (statement 15). From Lemma 6, any process $p \neq q$ that is active in t_0 either exits the *SynchPhase* procedure during $[t_0, t_1]$ or applies at least $2M \log N$ primitive operations during this time-interval. Let j be the number of the phase that begins in t_1 when q executes statement 30. Then any process that is active in t_0 and not in the *UP* mode is in the *PROP_RESPONSES* mode of phase j' for some $j' < j$. Let t' denote the time when q finishes its first $M(\log N + 1)(k+3)$ iterations in the *ROOT_WAIT* mode. From Lemma 9, all these processes free any nodes they own and return from their current instance of *SynchPhase* by time t'.

Let n be the node served by p at time t'. From Lemma 6, during the time-period $[t', t_1]$, p performs at least $M \log N$ primitive operations. It follows that during this time period p performs at least $\log N$ iterations in the *SynchPhase* procedure and is continuously in the *UP* mode. Thus, at time t', n is either a child of the root node or n's parent is owned by another process in the *UP* mode. ■

Lemma 12 *Let $T = [t_s, t_e]$ be an* UP *interval of q. Then q applies at most $14M(\log N + 1)(k+3)^2$ primitive operations during T.*

Proof: Let $\mathcal{P}$ denote the group of processes that participate in a phase at time t_s and let j be the maximum number of these phases or -1 if $\mathcal{P}$ is empty. Let t_0 be the time when the last of these processes exits its current phase and finishes its *Fetch&Inc* operation. Note that $t_0 = t_s$ if $\mathcal{P}$ is empty. Let l_1 denote the number of primitive operations applied by q during time interval $[t_s, t_0]$. From Lemma 10, we get:

$$l_1 \leq 10M(logN + 1)(k+3). \tag{4}$$

Let t_1 denote the first time after t_s in which a new phase starts. Also let l_2 denote the number of primitive operations applied by q during time interval $[t_s, t_1]$. We prove the following inequality:

$$l_2 \leq l_0 + (M \log N + 1)(k+2). \tag{5}$$

By time t_0, either phase $j+1$ has already started or all active processes are in the *UP* mode. In the first case, Inequality 5 clearly holds. In the later case, some process succeeds in capturing the root's slock and initiates phase $j+1$ after applying at most $M \log n$ steps. From Lemma 4, this implies that phase $j+1$ starts before q performs $(M \log N + 1)(k+2)$ primitive operations after t_0. This proves Inequality 5.

At time t_e q joins phase $j+1$. We let l_3 denote the number of primitive operations applied by q in the interval $[t_1, t_e]$. We prove the following inequality.

$$l_3 \leq (3M(k+3)\log N + 1)(k+2). \tag{6}$$

If q initiates phase $j+1$ then $t_1 = t_e$ and the claim is obvious. Otherwise, we consider the following two subintervals of $[t_1, t_e]$:

- Let t'_1 denote the time when the initiator of phase $j+1$ shifts to the *FREEZE_NODES* mode. From Lemma 6, q performs at most $(2M(k+3)\log N + 1)(k+2)$ primitive operations during subinterval $[t_1, t'_1]$.
- We now consider the subinterval $[t'_1, t_e]$. Let m denote the node served by q at time t_1 and let h denote m's height. From Lemma 11, the path from m to the root at time t_1 consists of nodes served by processes in the *UP* mode. For $i \in \{h, \cdots, \log N\}$, let p_i denote the process that serves the node in height i along this path, with $q = p_h$ and $p_{\log N}$ being the initiator of phase $j+1$ (note that the same process may serve multiple nodes along this path). Starting from time t'_1, p_{logN} performs a single iteration in the *FREEZE_NODES* mode. During this iteration it sets the *inPhase* flag of the node served by $p_{\log N-1}$ (statement 39). From Lemma 5, q performs at most $(M+1)(k+2)$ primitive operation during this iteration. Following that, $p_{\log N-1}$ performs at most a single full iteration before it sets the *inPhase* flag of the node served by $p_{\log N-2}$. Continuing in this manner, we get that the total number of primitive operations performed by q during $[t'_1, t_e]$ is at most $(M+1)\log N(k+2)$.

Summing up over these two sub-intervals proves Inequality 6. The lemma now follows from inequalities 4 - 6.

■

Lemma 13 *An application of the* FwdReqs *procedure to node n either has no effect or correctly forwards new requests from n's child* m *designated by* direction *to n.*

Proof: Assume the procedure is applied by process q to node n. The *FwdReqs* procedure is applied to n as a pseudo-transaction. Hence either it has no effect, or its statements are applied to n sequentially with no interleaved writes applied to fields it writes to. In the later case, clearly from the code, the difference between *m.requests* and the entry of *n.requestsTaken* corresponding to m is computed and added to *n.requests*. ■

The proof of the following lemma is very similar and is therefore omitted.

Lemma 14 *An application of the* SendResponses *procedure to node n either has no effect or correctly sends new responses from n's to its children*

Lemma 15 *When a process descends to its leaf node in the* PROP_RESPONSES *mode in E, there is a single response at that node.*

Proof: Let q be a process performing statement 69 and let j be the phase in which it participates when it performs it. Let $\mathcal{N}$ be the path of nodes starting with q's leaf node and ending with the root. Also, let t_0 be the time when q enters the *FORWARD_REQUESTS* mode after performing statement 35 in phase j.

Since E is BWC-synchronous then, from the code, starting from time t_0 the *FwdReqs* function is called once (in statement 46) for each of the nodes in $\mathcal{N}$, with the exception of q's leaf, in increasing order of their height. From Lemma 13, calling *FwdReqs* guarantees that requests have been forwarded from child to parent.

Let t_1 be the time when the process serving the root enters the *PROP_RESPONSES* mode of phase j in statement 57. Let Δ denote the increase in the value of the $requests$ field of the root$_i$ from time t_0 to time t_1. It is easily shown by induction that Δ equals the number of the processes that participate in phase j.

Similarly, starting from t_1, the *SendResponses* function is called once (in statement 71) for each of the nodes in $\mathcal{N}$, with the exception of q's leaf, in decreasing order of their height. It is easily shown by induction that each node along $\mathcal{P}$ receives a number of responses that is equal to the number of requests taken from it in phase j. Thus, from Lemma 14, this implies that when q performs statement 69 there is a single response in its leaf node. ■

We now prove that any k-synchronous execution is also BWC-synchronous (hence also k-BWC-synchronous).

Lemma 16 *Any k-synchronous execution is also BWC-synchronous.*

Proof: The proof goes by induction on the number of statements performed. The initial configuration is vacuously BWC-synchronous. Assume now that any k-synchronous execution in which i statements are performed is BWC-synchronous and let $E = E's$ be a k-synchronous execution of length $i + 1$. We only have to consider statements s where the *SynchPhase* procedure returns the *ASYNCH* response. We now consider each of these statements. In what follows, q denote the process that performs s.

- Statement 12: from the initialization of *timer* in statement 1, this implies that q performed in E' more than $2Mk(k + 13) \log N$ iterations while in the *UP* mode. By the induction hypothesis, E' is k-BWC-synchronous. Hence, this is a contradiction to Lemma 12.

- Statement 51: from statements 23 and 29, this implies that q performed in E' more than $(3M + 2)(\log N + 1)(k + 2)$ iterations while in the *PHASE_FREEZE* and *FORWARD_REQUESTS* modes. By the induction hypothesis, E' is k-BWC-synchronous. Hence, this is a contradiction to Lemma 7.

- Statement 65: from statement 59, this implies that q performed in E' more than $5M(\log N + 1)(k + 3)$ iterations while in the *AWAIT_RESPONSES* mode. By the induction hypothesis, E' is k-BWC-synchronous. Hence, this is a contradiction to Lemma 8.

- Statement 68: in this case, q has reached its leaf while in the *PROP_RESPONSES* mode and failed to find a response there. By the induction hypothesis, E' is k-BWC-synchronous. Hence, this is a contradiction to Lemma 15 .

■

Lemma 17 *Let E be a k-synchronous execution and let T be an E-interval during which some process q executes a single instance of the* SynchPhase *procedure. Then the maximum number of primitive operations executed by any process p during T is $24M(\log N + 1)(k + 3)^2$.*

Proof: From Lemma 16, E is k-BWC synchronous. The claim now follows from Lemmas 10 and 12. ■

Theorem 1 *The latency of operation instances in k-synchronous executions is* $\Theta(\log n)$.

Proof: Let E be a k-synchronous execution for some constant k and let T be an E-interval during which some process q executes a single instance of the *SynchPhase* procedure. From Lemmas 3 and 17, the duration of T is $O(k^2 \log N)$. As for the other direction, since the height of the combining tree used by the algorithm is $\Theta(\log N)$, operation latency is $\Omega(\log N)$. ■

Theorem 2 *The throughput of the BWC algorithm with asynchrony tolerance parameter k in k-synchronous executions in which all processes start their operations concurrently (i.e., apply their first invocation at time 0) is* $\Omega(N/(k^2 \log N))$.

Proof: Let E be a k-synchronous execution in which all processes start their *Fetch&Inc* operations in the initial configuration and perform them until they are completed. Clearly, *completed(E)=n* holds. From Lemma 16 and the k-synchrony of E, E is k-BWC synchronous. Hence, from Theorem 1, $time(E) = O(k^2 \log N)$ time. The claim now follows from Definition 2 ■

7.3 The BWC Algorithm is Nonblocking and Linearizable

In this section we prove that the BWC algorithm is nonblocking and linearizable. In the following we let k denote the asynchrony tolerance parameter with which the BWC algorithm is performed. We need some new definitions for our proofs. First, we define how we quantify the number of requests and responses that exist at a node in a configuration.

Definition 6 *Let n be a node. We define* the number of requests at node n in configuration C*, denoted $reqs(C, n)$ as follows:*

1. *If node n is a non-root node with parent m, statement 4 of* FwdReqs *increased* n.reqsTaken[direction] *by some positive δ and the following statement 5 was not yet performed, then $reqs(C, n) = \delta + n.requests - m.reqsTaken[direction]$, where* direction *is the entry of* reqsTaken *in m corresponding to n. Otherwise,*

2. *if node n is a non-root node with parent m, then $reqs(C, n) = n.requests - m.reqsTaken[direction]$, where* direction *is the entry of* reqsTaken *in m corresponding to n. Otherwise,*

3. *node n is the root node: if statement 4 of* FwdReqs *increased* n.reqsTaken[direction] *by some positive δ, and the following statement 8 was not yet performed, then $reqs(C, n) = \delta$. Otherwise $reqs(C, n) = 0$*

Definition 7 *Let n be a node. We define* the number of responses in node n in configuration C*, denoted* resp(C,n)*, as follows:*

1. *If statement 17 of* SendResponses *dequeued a range of k responses, the following statement 18 was not yet performed and there are x responses altogether in n's* responses *queue in C, then $resp(C, n) = x + k$. Otherwise,*

2. *$resp(C, n)$ is defined as the total number of responses that are in n's* responses *queue in C.*

In the following proofs, we view a request injected at a leaf by the execution of an operation instance Φ as *tagged by* Φ. We assume all operation instances are distinct, and so each request receives a unique tag. When a response is forwarded from a node to its parent (in statements 4-5 of *FwdReqs*), it maintains its tag. Also, when δ responses are generated at the root (in statement 8 of *FwdReqs*) we assume that they are tagged with the tags of the corresponding requests moved to the root in the same invocation of *FwdReqs*; that is, for $1 \leq i \leq \delta$, the i'th response is tagged with the tag of the i'th request. When a response is propagated from a node to its child (in statement 18 of *SendResponses*), it maintains its tag.

The following definitions define an order between all requests and responses that arrive at a node in the course of an execution.

Definition 8 *Let E be an execution and let Φ and Φ_1 be two instances of* Fetch&Inc*, performed in E by processes p and p_1, respectively. Let n be a node in the intersection of the paths from the leaf nodes of p and p_1 to the root. We say that* the Φ-tagged request precedes the Φ_1-tagged request in n*, if it holds that*

1. *the call to* FwdReqs *in which the Φ-tagged request is forwarded to n precedes the call to* FwdReqs *in which the Φ_1-tagged request is forwarded to n (or the later call does not occur), or*
2. *both requests are forwarded to n from a child* ch *in the same call to* FwdReqs *and the Φ-tagged request precedes the Φ_1-tagged requests in* ch*.*

Definition 9 *Let E be an execution and let Φ and Φ_1 be two instances of* Fetch&Inc *that are performed in E by processes p and p_1, respectively. Let n be a node in the intersection of paths from the leaf nodes of p and p_1 to the root. We say that* the Φ-tagged response precedes the Φ_1-tagged response in n*, if it holds that*

1. *the call to* EnqRange *(in either statement 8 of* FwdReqs*, if n is the root node, or in statement 18 of* SendResponses*, otherwise) that sends the Φ-tagged response to n precedes the one that sends the Φ_1-tagged response to n (or the later call does not exist), or*
2. *both responses are sent to n in the same* EnqRange *call and the Φ-tagged request preceded the Φ_1-tagged request at the root.*

Lemma 18 *The following holds for all executions E.*

1. *For any instance Φ of* Fetch&Inc *in E, performed by some process p, a Φ-tagged response can only descend to nodes n along the path from the root to p's leaf.*
2. *Let Φ and Φ_1 be two instances of* Fetch&Inc *in E, performed by processes p and p_1 respectively, and let n be a node on the intersection of the paths between the leaves of these processes and the root. If the Φ-tagged request precedes the Φ_1-tagged request at node n and both responses arrive at n then the Φ-tagged response precedes the Φ_1-tagged response in n.*

Proof: We prove the lemma by induction on the distance of n from the root. For the base case, node n is the root and the distance is 0. Requests forwarded to the root are immediately transformed to responses in statement 8 of *FwdReqs*. Hence both the Φ-tagged and the Φ'-tagged responses arrive at the root and, from Definitions 8 and 9, the Φ-tagged response precedes the Φ_1-response at the root.

Assume the claim holds for nodes of distance k or less from the root and let n be a node in distance $k+1$ from the root. Let m denote n'th parent. From the *FwdReqs* code, all requests are forwarded from n to m,

hence, specifically, the Φ-tagged and the Φ_1-tagged requests are forwarded to m. From induction hypothesis applied to claim 1., both corresponding responses descend to m. Since the Φ-tagged request precedes the Φ_1-tagged request at n, we have from Definition 8 and the *FwdReqs* code that the Φ-tagged request precedes the Φ_1-tagged request also at m. From induction hypothesis claims 1. and 2., tagged-responses arrive at m in exactly the same order in which the respectively-tagged requests arrived at m. Since this order is maintained in m'th *pending* queue in statement 6 of *FwdReqs*, and responses are sent from m according to this order in statement 13 of *SendResponses*, it follows that both the Φ-tagged and the Φ_1-tagged must descend from m to n and that the former precedes the latter at n. ■

Lemma 19 *Let p be a process and let E be an execution in which p has already injected a request (either in statement 33 of* SynchPhase *or in statement 4 of* AsynchFAI*) in the course of executing an operation instance Φ. Let C denote the configuration after E. Assume also that p did not dequeue a response in Φ (in statement 69 of* SynchPhase *or in statement 17 of* AsynchFAI*). Then exactly one of the following conditions holds in C:*

1. *there is a single node on the path from p'th leaf to the root with a Φ-tagged request, or*
2. *there is a single node on the path from p'th leaf to the root with a Φ-tagged response.*

Proof: From Definition 6 and the code, the claim's invariant can be violated only by the injection statements or by statements of the *FwdReqs* or *SendResponses* procedures. We prove the lemma by induction on the number of such statements executed in E. For the base case, observe that condition 1. holds right after a request in injected by p to its leaf, since it creates a p-tagged request at the leaf. Assume the invariant holds for the first j such statements and consider the execution of the $j+1$ statement. The following cases exist.

- Statement 4 of *FwdReqs*: this statement increases n's *reqsTaken* entry, corresponding to some child *ch*, by the *delta* computed in the preceding statement 2. From Definition 6 (clause 1.), right after this statement executes, these *delta* requests are still at ch. Hence the invariant is maintained
- Statement 5 of *FwdReqs*: this statement increases n's *requests* field by *delta*. If n is not the root node then, from Definition 6 (clause 2.), this atomically moves *delta* requests from one of n's children to n and the invariant is maintained. Otherwise, n is the root node and, right after the statement executes, we are under clause 3. of Definition 6. Hence the effect of the statement is to atomically move *delta* requests to the root node from one of its children and the invariant is maintained.
- Statement 8 of *FwdReqs*: This statement enqueues the responses produced on behalf of the δ requests moved on the preceding statement 5 to the root. From Definition 6 (clauses 2. and 3.), this has the effect of transforming these requests to corresponding responses (at the root) with the same tags. Thus, the invariant is maintained.
- Statement 17 of *SendResponses*: this statement dequeues *numToSend* responses from n's responses queue. From Definition 7 (clause 1.), this does not change the responses that are at n and the invariant is maintained.
- Statement 19 of *SendResponses*: This statement enqueues a range of responses, dequeued in the preceding statement 17. From Lemma 18, these responses are enqueued to the responses queue of

the child *ch* of n from which requests with the same tags were forwarded to n . From Definition 7 (clauses 1. and 2.), this has the effect of moving these responses from n to its child and the invariant is maintained.

■

Theorem 3 *The BWC algorithm is nonblocking.*

Proof: From Theorem 1, in k-synchronous executions of the algorithm, each operation terminates in $O(\log N)$ time. It follows that, in such executions, the algorithm is wait-free hence also nonblocking. We still need to prove that the BWC algorithm is nonblocking in executions that are not k-synchronous. Assume otherwise in contradiction. Then there is a non k-synchronous execution $E = E'E''$ such that the following conditions hold:

1. Starting from the configuration reached after E', all processes in synchronous modes (if any) have fail-stopped,
2. no process starts a new operation after E',
3. E'' is infinite and no operation terminates in E''.

As all the processes that take steps in E'' are in asynchronous modes, each such process iteratively traverses the path from its leaf to the root and back applying the *FwdReqsAndSendResp* procedure to each of the nodes on its way.

Let q be a process that is in the midst of executing instance Φ in E'' and does not fail-stop in E'' (if there is no such process then the nonblocking requirement is satisfied vacuously). Let C be the configuration right after E'. From Lemma 19, there is a single node n in C that has either a Φ-tagged request or a Φ-tagged response. Assume the first case first and consider the first N full path traversals (namely, traversals that go up from the leaf node to the root and back again) that are performed by q in E''. Since all steps taken in E'' are by processes in asynchronous modes, the only combining operation applied in E'' is *FwdReqsAndSendResp*. Thus, from the node progress requirement of pseudo-transactions, each application of *FwdReqsAndSendResp* to a node completes an instance (not necessarily initiated by q) of *FwdReqsAndSendResp* on the node. Observe that the maximum size of a requests queue or a responses queue is N since this is the maximum number of concurrent operations. It follows that after at most N path traversals by q, the Φ-tag response is either moved up the tree (if n is not the root) or transformed to a q-tag response (if n is the root). Continuing in this manner we get that, after at most $N \log N$ full iterations performed by q in E'', a Φ-tagged response is at the root. Let C' be the configuration when this first happens.

By applying similar arguments, and by using Lemma 18 to show that the Φ-tagged response must descend down towards q's leaf, we get that after q performs at most $N \log N$ additional full path traversals starting from C', the Φ-tagged response arrives at q's leaf. Hence q receives a response and terminates its operation in E''. This contradicts our assumption that the algorithm does not satisfy the nonblocking condition. ■

Theorem 4 *The BWC algorithm is a linearizable implementation of a counter.*

Proof: Let Φ_1 and Φ_2 be two instances of the *Fetch&Inc* procedure that complete in some execution E. Let p and q (which may be equal) denote the processes that perform Φ_1 and Φ_2, respectively. Assume also that Φ_1 terminates before Φ_2 starts and let v_1 and v_2 be the counter values returned by Φ_1 and Φ_2, respectively.

Let r_{Φ_1} and r_{Φ_2} respectively denote the Φ_1- and Φ_2-tagged responses. Clearly from assumptions, the Φ_1-tagged request precedes the Φ_2-tagged request at the root. It follows from clause 2. of Lemma 18 that r_{Φ_1} precedes r_{Φ_2} at the root and is hence is smaller. Finally, from Clause 1. of Lemma 18, r_{Φ_1} (respectively, r_{Φ_2}) is the response returned by Φ_1 (respectively, by Φ_2's). It follows that $v_1 < v_2$. ■

7.4 Performance in Non k-Synchronous Executions

In this section we analyze the throughput of the BWC algorithm with asynchrony tolerance parameter k when the system is not k-synchronous but maintains k'-synchrony for some constant $k' > k$. In these circumstances, processes may execute in asynchronous modes and throughput may be greatly reduced. We prove that the BWC algorithm guarantees throughput of $\Omega(1/N)$ in such executions and show that this bound is tight by describing an execution with such throughput.

Lemma 20 *The latency of the* SynchPhase *procedure in k'-synchronous executions, for $k' > k$, is $O(N \log N)$.*

Proof: Clearly from the code, the maximum number of primitive operations performed by a process as it performs a single instance of the *SynchPhase* procedure is $O(\log N)$. The lemma follows from the fact that no primitive operation can incur more than $N - 1$ stalls. ■

Next, we consider the latency of the *AsynchPhase* procedure in executions in which each process performs a single instance of *Fetch&Inc*. In the following we let t^* denote the time after which all processes have either terminated their *Fetch&Inc* operations or have shifted to an asynchronous mode. We denote by $\mathcal{P}_q$ the set of nodes on the path from q's leaf to the root. The following definition is required for the proofs that follow.

Definition 10 *Let q be a process that shifts in E to an asynchronous mode. We say that q performs an* asynchronous round *when it climbs from its leaf to the root while performing statements 5-14 in the* ASYNCH_UP *mode and then descends back down to its leaf while performing statements 16-24 in the* ASYCNH_DOWN *mode.*

Lemma 21 *Let q be a process that starts an asynchronous round in time $t' > t^*$ and let t'' denote the time when q descends from the root in that round. If there are one or more requests in the nodes of $\mathcal{P}_q$ at time t', then at least one new response has been generated at the root during time interval $(t', t'']$.*

Proof: Since all active processes are in asynchronous modes after time t^*, the only procedure performed as a pseudo-transaction starting from that time is *FwdReqsAndSendResp*. It follows that, whenever q performs *FwdReqsAndSendResp* on node n, an instance of *FwdReqsAndSendResp* on node n terminates. Let n denote the highest node in $\mathcal{P}_q$ in which there are requests at time t'. When q starts performing *FwdReqsAndSendResp* on n's parent, either n's requests queue is empty or, otherwise, that instance of *FwdReqsAndSendResp* will forward at least a single request of n to n's parent (and generate the corresponding response, if n's parent is the root). In either case, some request has been forwarded from n during $(t', t'']$. The claim follows by applying this argument iteratively as q ascends the nodes of $\mathcal{P}_q$. ■

Lemma 22 *Let q be a process that starts an asynchronous round in time $t' > t^*$ and let t'' denote the time when q terminates that round. Then at least one response has been propagated out of $\mathcal{P}_q$ during time interval $(t', t'']$.*

Proof: Since q starts a new asynchronous round in time t', either q's response or q's request is at some node in $\mathcal{P}_q$ at time t'. In the latter case, Lemma 21 guarantees that, at some point during time interval $(t', t'']$, a new response is generated at the root. Let t_0 denote the earliest time during interval $(t', t'']$ in which there is a response in some node of $\mathcal{P}_q$ and let n denote the lowest node in $\mathcal{P}_q$ in which there are responses at t_0.

Since all active processes are in asynchronous modes after time t^*, the only procedure performed as a pseudo-transaction starting from that time is *FwdReqsAndSendResp*. It follows that, whenever q performs *FwdReqsAndSendResp* on node n, an instance of *FwdReqsAndSendResp* on node n terminates. When q starts performing *FwdReqsAndSendResp* on node n, either n's responses queue is empty or, otherwise, that instance of *FwdReqsAndSendResp* will propagate at least a single response r to a child of n or, if n is q's leaf, that response will be returned. If r has been propagated to a node not in $\mathcal{P}_q$ or has been returned the claim clearly holds. Otherwise, the claim follows by applying this argument iteratively as q descends down the nodes of $\mathcal{P}_q$. ■

Theorem 5 *The throughput of the BWC algorithm with asynchrony tolerance parameter k in k'-synchronous executions, for constant $k' > k$, in which all processes start their operations concurrently (i.e., apply their first invocation at time 0) is $\Omega(1/N)$.*

Proof: Let E be a k'-synchronous execution in which all processes start their operations concurrently. Let t^{**} denote the time after t^* when all processes in asynchronous modes (if any) have already started their first asynchronous round. From Lemma 20 and from the code of *AsynchFAI*, $t^{**} = O(N \log N)$ holds. We now consider the time spent by a process in *AsynchFAI* after t^{**}.

Consider first the latency incurred by q in each asynchronous round after t^{**}. Clearly from the code, q performs $\Theta(\log N)$ primitive operations in each asynchronous round. When q applies a primitive operation to a node $n \in \mathcal{P}_q$ of height h, it incurs at most 2^{h+1} stalls. This follows from the fact that only the processes whose leaves belong to the sub-tree rooted by n's parent (or n itself, it n is the root node) ever access n. Since q applies a constant number of primitive operations to each node in $\mathcal{P}_q$ in any single asynchronous round, the total number of stalls incurred by q in each such round is $O(N)$. It follows that the latency of each round is $O(N)$.

Assume q is in an asynchronous mode at time t^{**} We now bound the number of asynchronous rounds performed by q after t^{**}. At the beginning of a round, either q's request is stored in the requests queue of some node in $\mathcal{P}_q$ or its respective response is stored in the responses queue of some node in $\mathcal{P}_q$. Since exactly N instances of *Fetch&Inc* are performed in E, we have from Lemma 21 that, by the time q completes its N'th asynchronous round after t^{**}, a response corresponding to q's request has already been generated at the root. By a similar argument based on Lemma 22, q returns its response at the end of the $2N$'th asynchronous round performed after t^{**} at the latest. Since q performs at most $2N$ asynchronous rounds after t^{**}, the latency of each of which is $O(N)$, q completes its *Fetch&Inc* operation in $O(N^2)$ time. The claim follows. ■

We now sketch a scenario showing that the lower bound of Theorem 5 is tight. Consider an execution E of the BWC algorithm with asynchrony tolerance parameter k, in which processes $p_0, \ldots, p_{N-1}$ simultaneously start executing their *Fetch&Inc* operations in time 0. We construct E as follows.

1. We first schedule processes in lock-step, until process p_N captures the root node and all other processes wait in the *UP* mode to be joined to a phase.

2. Next, the scheduler slows p_N down and lets it perform a single step every k' steps performed by the other processes, for some $k' >> k$. After $O(k \log N)$ time, processes $p_1, \ldots p_{N-1}$ all shift to asynchronous mode (in statement 12 of *SynchFAI*).

3. When operating in asynchronous mode, each of processes $p_1, \cdots, p_{N-1}$ injects a request at its leaf node (statement 4 of *AsynchFAI*) and proceeds to forward its request.

4. We schedule the steps of $p_1, \cdots, p_{N-1}$ so that each non-empty entry in the *pending* queue contains exactly a single request. That is, whenever the condition of statement 3 of *FwdReqs* holds, the value of local variable *delta* is 1. Moreover, we schedule operations such that, at each node n, requests of processes are enqueued into *n.pending* in increasing order of process ID.

   Finally we reach a configuration in which there are $N-1$ responses in the root node, each occupying a separate entry of the root's *responses* queue, in increasing order of process ID.

5. Finally, we let all processes perform the *FwdReqsAndSendResponses* procedures in lock-step as they execute their asynchronous rounds: Initially, they all perform *FwdReqsAndSendResponses* on the root node. Then, they all perform *FwdReqsAndSendResponses* on the two children of the root node, and so on. Note that, since the *FwdReqsAndSendResponses* procedure is performed as a pseudo-transaction, whenever a few processes execute it in lock-step on the same node, only a single instance of the procedure completes on that node.

   Since the response of process p_{N-1} is in the $(N-1)$'th position in the *responses* queue of the root node, p_{N-1} will perform $N-1$ asynchronous rounds. Moreover, since exactly a single process terminates in asynchronous round i, p_{N-1} incurs $\Theta(N-i-1)$ stalls in round i, for $i = 1, \ldots, N-1$. It follows that p_{N-1} terminates its operation after $\Theta(N^2)$ time.

8 The Throughput of Prior Art Counter Algorithms

In this section we consider the throughput of a few prior art shared-memory counter algorithms. We confine our attention to executions in which each process performs a single *Fetch&Inc* operation.

- **Centralized counter**: in a centralized counter implementation, all processes apply their operations by applying a primitive (hardware-supported) *Fetch&Inc* operation to the same variable. The throughput of the centralized counter is 1, since if several processes apply their *Fetch&Inc* operations concurrently, whenever one of them receives its response all pending operations incur a single stall.

- **Counting networks**: Aspnes et al. introduce *counting networks* [1], constructed from two-input two-output computing elements called *balancers*. The counting networks they present are non-linearizable and wait-free. The number of primitive operations applied by a process as it performs its *Fetch&Inc* operation to the counting network is proportional to the network's *depth* - the maximum length of a path between an input balancer and an output balancer. The counting networks presented in [1] have width $\Theta(\log^2 N)$. In executions in which each process performs at most a single *Fetch&Inc* operation, these networks ensure that every primitive operation incurs a constant number of stalls. Thus, all

processes terminate their operations after $\Theta(\log^2 N)$ time, implying throughput of $\Theta(N/\log^2 N)$ in k-synchronous executions (for any constant k) in which all processes simultaneously start their *Fetch&Inc* operations.

- **Combining**: Yew et al. [22] propose a locking software-combining counter algorithm that allows each process to perform a single decrement operation. If all processes start their operations simultaneously, then their algorithm achieves $\Theta(N/\log N)$ throughput in k-synchronous executions, for all constant k. The algorithm is limited, since each process is allowed to perform only a single *Fetch&Inc* operation. Moreover, it assumes that all processes participate. If even a single process does not "show up", then an execution can be constructed in which the rest of the processes deadlock, implying 0 throughput.

  Goodman et al. [7] propose a locking software-combining algorithm that does not possess the limitations of [22] and provides the same throughput.

- **Diffracting trees**: Shavit and Zemach introduce *diffracting trees* [19] to replace the static tree used in software combining with a collection of randomly created dynamic trees. Diffracting trees are wait-free but non-linearizable. They are composed of *diffracting balancers*. A diffracting balancer is based on adding a collision array called *prism* in front of a single-input two-output balancer. To perform its *Fetch&Inc* operation, a process first tries to combine its operation with another process by picking a random location in the *prism* array of the root node and trying to collide on that location. Whenever two processes collide, they both descend down the tree in different directions without having to access the root balancer. A process that fails to collide has to access the root balancer to determine the direction it which it will descend. Processes continue operating this way as they descend down the tree, until they reach a leaf node, obtain a counter value and return.

  The worst-case throughput of the diffracting-tree-based counter can be as low as $O(1/N)$ even in synchronous executions. This can happen if all processes fail to collide in the prism array of the root balancer. If this occurs, then all processes repeatedly try to "capture" the root balancer by performing test-and-set operations. If a few processes concurrently apply test-and-set operations to the root balancer, only one succeeds and all others incur stalls. Thus, in this scenario, the last process to succeed incurs a total of $\Sigma_{i=1}^{N-1} i = \Theta(N^2)$ stalls, implying a throughput of $O(1/N)$. On the other hand, throughput can be as high as $\Theta(N/\log N)$, if all collision attempts are successful.

- **Combining funnels**: Shavit and Zemach introduce *combining funnels* [20], a software-combining-based scheme in which processes randomly try to collide with other processes and combine their operations. Specifically, they present a counter implementation based on their scheme ([20, Figure 14]). To try and collide, a process p randomly picks a location in a *collision layer* and tries to swap out an ID of another process q from this location. If it succeeds, then p and q can combine their operations. Otherwise, p busy-waits for *spin* iterations (*spin* is an algorithm parameter), to allow other processes to collide with it.

  Whenever two processes manage to collide, one of them waits indefinitely for the other to return with a response. It is this indefinite waiting period that makes the combining-funnels counter algorithm locking; it is, however, linearizable. If a process fails to collide, then it attempts to complete its operation by performing *CAS* on a centralized variable called *counter*, storing the counter value.

If N processes start their *Fetc&Inc* operations concurrently, if the algorithm uses $\Theta(\log N)$ collision layers and if each process manages to collide in each collision layer it accesses, then combining funnels may provide high throughput of $O(N/\log N)$. On the other hand, even when the execution is synchronous, throughput may be as low as $O(1)$ if all processes randomly pick the same location in the collision layer.

Consider now a k-synchronous execution in which $k >>$ *spin* holds. In this case, regardless of the width of the collision layers used and of the random collision-layer position selected, all processes may fail to collide and end up applying *CAS* to the central *counter* variable. An adversarial scheduler can then cause all but a single *CAS* operation to succeed whenever multiple *CAS* operations are applied to *counter*. Thus, the last operation to succeed incurs a total of $\Sigma_{i=1}^{N-1} i = \Theta(N^2)$ stalls, implying a throughput of $O(1/N)$ in this scenario also.

9 Discussion

In this paper, we have presented the bounded-wait combining (BWC) algorithm. This is the first implementation of a shared counter that is both linearizable, nonblocking and can provably achieve high throughput in k-synchronous executions. Our algorithm can be easily adapted for implementing additional shared objects that support combinable operations, such as queues and stacks, possessing the same properties.

We have also introduced a formal contention-aware metric for the throughput of concurrent objects and used it for a rigorous analysis of the BWC algorithm's throughput. The BWC algorithm uses two techniques - synchronous locks and pseudo-transactions, a weakening of regular transactions - that we believe may be useful for the design of additional algorithms with similar properties.

We have shown that, when the level of asynchrony is bounded by some constant k, the BWC algorithm can obtain throughput of $\Theta(N/\log N)$. We leave the question of whether or not this is optimal to future work. If one considers k that may be a function of N, then the maximal throughput obtained by the BWC algorithm is $\Omega(N/k^2 \log N)$. Whether this is optimal or not is another interesting open question.

Our algorithm uses the asynchrony tolerance parameter k for achieving its throughput. If the system behaves in a k'-synchronous manner, for $k' > k$, then the BWC algorithm can no longer guarantee high throughput. Our analysis shows that in this case, similarly to diffracting trees [19] and combining funnels [20], worst-case throughput can be as low as $O(1/N)$. A natural question is whether an *adaptive* version of the BWC algorithm, one that has no a-priori knowledge of k, can be devised. Such an adaptive algorithm would have to adjust processes' waiting times dynamically according to the changing synchrony level of the system.

Full-binary combining trees have space complexity of $\Omega(N)$. Each node of the combining-tree used by the BWC algorithm stores the *pending* and *responses* queues, which help guarantee its linearizability and wait-freedom properties. In asynchronous executions, these queues may have to accommodate up to N entries each, since N is the maximum number of simultaneous operations. Thus the BWC algorithm has space complexity of $\Theta(N^2)$. We do now know whether this space complexity is required for achieving the combination of properties provided by our algorithm.

The BWC algorithm employs pseudo-transactions by applying Greenwald's two-handed emulation [9] to all the nodes of the combining tree in parallel and making sure that pseudo-transactions applied to different nodes do not write to the same locations. It would be interesting to see whether the semantics of pseudo-transactions can be satisfied also by applying local algorithms such as that of Attiya and Dagan [2] instead.

Acknowledgments: We would like to thank Erez Michalak and the anonymous referees for many helpful comments on an earlier draft of this paper.

References

[1] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.

[2] H. Attiya and A. Dagan. Improved implementations of binary universal operations. *Journal of the ACM*, 48(5):1013–1037, 2001.

[3] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, 2005.

[4] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, 1997.

[5] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: scalable nonzero indicators. In *Proceedings of the 27th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[6] F. E. Fich, D. Hendler, and N. Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *Proceedings of the 46th Annual Symposium on Foundations of Computer Science (FOCS)*, 2005.

[7] 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.

[8] 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.

[9] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 260–269, New York, NY, USA, 2002. ACM Press.

[10] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Self-tuning reactive distributed trees for counting and balancing. In *OPODIS*, pages 213–228, 2004.

[11] D. Hendler and S. Kutten. Bounded-wait combining: Constructing robust and high-throughput shared objects. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, pages xx–xx, 2006.

[12] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.

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

[14] M. Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[15] M. Herlihy, N. Shavit, and O. Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.

[16] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

[17] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988.

[18] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262, New York, NY, USA, 2005. ACM Press.

[19] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[20] N. Shavit and A. Zemach. Combining funnels: A dynamic approach to software combining. *J. Parallel Distrib. Comput.*, 60(11):1355–1387, 2000.

[21] R. Wattenhofer and P. Widmayer. The counting pyramid: an adaptive distributed counting scheme. *J. Parallel Distrib. Comput.*, 64(4):449–460, 2004.

[22] 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.

A Code for Manipulating Requests and Responses Queues

```
structure Reqs {int direction, int num}, structure REQQ {Reqs requests[N+1], int in=0, int out=0}
structure Range {int from, int till}, structure RRQ {Range resposeRanges[N+1], int in=0, int out=0}

int RangeLen(Range r)
1   return r.till-r.from+1

ENQ_REQS(REQQ pending, int direction, int num)
2   pending.requests[pending.in].direction=direction
3   pending.requests[pending.in].num=num
4   pending.in=(pending.in+1) % (N+1)

DEQ_REQS(REQQ pending, numToSend)
5   if (pending.requests[pending.out].num > numToSend)
6      pending.requests[pending.out].num -= numToSend
7   else
8      pending.out=(pending.out+1) % (N+1)

REQS FIRST_REQS(REQQ pending)
9   return pending.requests[pending.in]

EnqRange(RRQ responses, int from, int num)
10  int localIn
11  responses.responseRanges[responses.in].from=from
12  responses.responseRanges[responses.in].till=from+till-1
13  localIn=(responses.in+1) % (N+1)
14  responses.in=localIn

DeqRange(RRQ responses, int numSent)
15  int localOut
16  Range r=responses.responseRanges[responses.out]
17  if (r.till-r.from+1 > numSent)
18     responses.responseRanges[responses.out].till -= numSent
19  else
20     localOut = (localOut+1) % (N+1)
21     responses.out = localOut

boolean Empty(RRQ responses)
22  if (responses.in=responses.out)
23     return true
24  else
25     return false

REQS FirstRange(RRQ responses)
26  return responses.responseRanges[responses.in]
```

Figure 8: Pseudo-code for the combining procedures