Low Contention Linearizable Counting

Maurice Herlihy * Nir Shavit † Orli Waarts ‡

Abstract

The *linearizable counting problem* requires asynchronous concurrent processes to assign themselves successive values so that the order of the values assigned reflects the real-time order in which they were requested. This problem lies at the heart of concurrent timestamp generation, as well as concurrent implementations of shared counters, FIFO buffers, and similar data structures. When the processes communicate via shared memory, conventional solutions to this problem require all processes to synchronize at a common memory location, resulting in poor performance from memory contention and a sequential bottleneck.

This paper's main contribution is to show that the linearizable counting problem can be solved without funneling all processes through a common memory location. We give two new constructions for *linearizable counting networks*, data structures that solve the linearizable counting problem. Our first construction is *non-blocking*: some process takes a value after $O(n)$ network gates have been traversed. Our second construction is *wait-free*: it guarantees that each process takes a value after it traverses $O(wn)$ gates, where w is a parameter affecting contention. We show that in any non-blocking or wait-free linearizable counting network, processes must traverse an average of $\Omega(n)$ gates, and so our constructions are close to optimal. Finally, we construct a simpler and more efficient network by giving up the robustness requirements and allowing processes to wait for one another.

1 Introduction

In the *linearizable counting problem*, asynchronous concurrent processes repeatedly assign themselves successive values, such as integers or locations in memory, so that the order of the values assigned reflects the real-time order in which they were requested. For example, if k values are requested, then values $0 \ldots k-1$ should be assigned, and if process P is assigned a value before process Q requests one, then P's value must be less than Q's. This problem lies at the heart of a number of basic problems, such as concurrent time-stamp generation, as well as concurrent implementations of shared counters, FIFO buffers, and similar data structures (e.g. [6, 11, 17, 23]).

The requirement that the values chosen reflect the real-time order in which they were requested is called *linearizability* [13]. The benefits of linearizability are clear: the use of linearizable data abstractions greatly simplifies both the specification and the proofs of shared memory algorithms. As discussed in more detail elsewhere [13], the notion of linearizability generalizes and unifies a number of *ad-hoc* correctness conditions in the literature, and it is related to (but not identical with) correctness criteria such as sequential consistency [18] and strict serializability [20].

In a MIMD shared memory multiprocessor, the

*Digital Equipment Corporation, Cambridge Research Lab.

†MIT Lab. for Computer Science. Supported by ONR contract N00014-91-J-1046, NSF grant CCR-8915206, DARPA contracts N00014-89-J-1988 and N00014-87-K-0825 and by a Rothschild postdoctoral fellowship.

‡Stanford University. Supported by the NSF grant CCR-8814921 and by ONR contract N00014-88-K-0166.

naive solution to this problem has all n processors synchronize at a common memory location, resulting in poor performance both from memory contention and from the absence of concurrency. Reducing such "hot-spot" contention has been the subject of extensive research in hardware architecture design [2, 11, 16, 10, 21] and experimental work in software [3, 8, 9, 19, 23].

1.1 Background

Recently, in [4], low-contention data structures called ***counting networks*** were shown to provide efficient ***non***-linearizable solutions to the counting problem. A counting network [4], like a sorting network [5], is a directed graph whose nodes are simple computing elements called ***balancers***, and whose edges are called ***wires***. Each ***token*** (input item) enters on one of the network's $w \leq n$ input wires, traverses a sequence of balancers, and leaves on an output wire. However, while a w input sorting network sorts a collection of w input values only if they arrive together, on separate wires, and propagate through the network in lockstep, a w input counting network can count any number $N \gg w$ of input tokens even if they arrive at arbitrary times, are distributed unevenly among the input wires, and propagate through the network asynchronously.

Figure 2 shows a four-input four-output counting network. Intuitively, a balancer (see Figure 1) is just a toggle mechanism [1] that repeatedly alternates in sending tokens out on its output wires. Figure 2 shows an example computation in which input tokens traverse the network sequentially, one after the other. For notational convenience, tokens are labeled in arrival order, although these numbers are ***not*** used by the network. In this network, the first input (numbered 1) enters on line 2 and leaves on line 1, the second leaves on line 2, and so on. (The reader is encouraged to try this for him/herself.) Thus, if on the ith output line the network assigns to consecutive output tokens the values $i, i+4, i+2\cdot 4, \ldots$, it is ***counting*** the number of input tokens without ever passing them all through a shared computing element!

Known counting network constructions [1, 4, 14] are not linearizable.

[1] A balancer may be implemented using ***Compare&Swap***, ***Test & Set***, or a randomized consensus primitive.

1.2 Summary of Results

There exist non-linearizable counting networks of $O(w \log^2 w)$ balancers, in which each token traverses $O(\log^2 w)$ balancers, where w is independent of n [4]. By contrast, we show in Section 3 that any non-trivial linearizable network must encompass an infinite number of balancers, and, if the network has low contention, each token must traverse an average of $\Omega(n)$ balancers. In other words, the structure of linearizable counting networks is fundamentally different from that of their non-linearizable counterparts.

Our major result, presented in Section 4, is the construction of two novel networks which we call ***linearizers***. A linearizer network, when connected to any given counting network, transforms it into a linearizable counting network. In the first SKEW network (see Figure 3), each token traverses an ***average*** of $O(n)$ balancers before leaving the network, but an individual token may be forced along an infinite path if it is infinitely often overtaken. The SKEW network is thus ***non-blocking:*** tokens that undergo halting failures or delays cannot prevent all non-faulty tokens from completing a network traversal. In the second REVERSE-SKEW network, each token traverses $O(n \cdot w)$ balancers, but starvation is impossible. This network is ***wait-free:*** tokens that undergo halting failures or delays cannot prevent any non-faulty token from completing a network traversal. If implemented directly in terms of balancers, these networks would have infinite size, so we give a simple technique for "folding" these infinite networks onto finite networks.

Finally, we observe that it is possible to construct a simple and efficient linearizable counting network if we augment non-linearizable counting networks with a simple waiting primitive. This efficiency, however, comes at a cost: the inopportune failure of a single process can prevent all the non-faulty processes from making progress.

2 A Brief Introduction to Counting Networks

For lack of space we have eliminated many of the formal definitions, which can be found in [4].

A ***balancer*** is a computing element with two input wires, denoted as the ***north*** and ***south*** wires (and

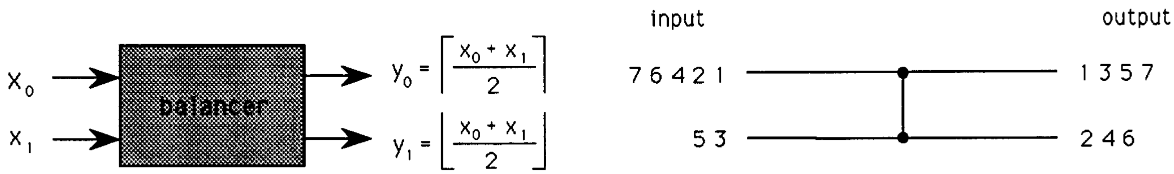

Figure 1: A Balancer.

indexed by 0 and 1), and two output wires, similarly named. We denote by x_i, $i \in \{0, 1\}$ the number of input tokens ever received on the balancer's i-th input wire, and similarly by y_i, $i \in \{0, 1\}$ the number of tokens ever output on its i-th output wire. Let the state of a balancer at a given time be defined as the sets of tokens that have visited its input and output wires. A balancer never swallows nor creates tokens, and given any finite number of input tokens $m = x_0 + x_1$ to the balancer, it is guaranteed that within a finite amount of time, it will reach a *quiescent* state, that is, one in which $x_0 + x_1 = y_0 + y_1 = m$. In any quiescent state, the balancer's outputs y_0 and y_1 have the *step property:* $0 \leq y_0 - y_1 \leq 1$.

A *balancing network* of *width* w is a collection of balancers, where output wires are connected to input wires. It has w designated input wires $x_0, x_1, .., x_{w-1}$ (which are not connected to output wires of balancers), and w designated output wires $y_0, y_1, .., y_{w-1}$ (similarly unconnected). The network has no cycles, and thus its *depth* is defined in the natural way. Let the state of a network at a given time be defined as the union of the states of its balancers.

In a shared memory multiprocessor, a balancing network is implemented as a data structure, where balancers are records and wires are pointers from one record to another. Each of the machine's n asynchronous processors runs a program that repeatedly traverses the data structure, each time shepherding a new token through the network (see Section 5). The limitation on the number of concurrent processes translates into a limitation on the number of tokens concurrently traversing the network:

$$\sum_{i=0}^{w-1} x_i - \sum_{i=0}^{w-1} y_i \leq n.$$

A *counting network* of width w is a balancing network whose outputs $y_0, .., y_{w-1}$ have the *step property* in quiescent states:

$$0 \leq y_i - y_j \leq 1 \text{ for any } i < j.$$

To illustrate this property, consider an execution in which tokens traverse the network sequentially, one completely after another. Figure 2 shows such an execution on the BITONIC[4] network defined in [4]. As can be seen, the network moves input tokens to output wires in increasing order modulo w. A balancing network having this property is called a *counting network*, because it can easily be adapted to count the number of tokens that have entered the network. Counting is done by adding a "local counter" to each output wire i, so that tokens coming out of that wire are consecutively assigned the numbers $i, i+w, i+2w, .., i+(y_i-1)w$. The number $i+w \cdot k$ assigned by the counter at the end of output line i to the k-th token exiting on it, is called the token's *value*.

Define the *traversal interval* of a token through the network to be the time interval $[t_{enter}, t_{exit}]$ from the moment it entered the balancing network and until it exited it.

Definition 2.1. *A counting network is linearizable if for any two tokens a and b with traversal intervals $[t^a_{enter}, t^a_{exit}]$ and $[t^b_{enter}, t^b_{exit}]$, if $t^a_{exit} < t^b_{enter}$ then value$(a) <$ value(b).*

3 Lower Bounds

Linearizable counting networks are fundamentally different from their non-linearizable counterparts. There exist non-linearizable counting networks of $O(w \log^2 w)$ balancers, where each token traverses $O(\log^2 w)$ balancers, where w is independent of n [4]. By contrast, we show here that any non-trivial linearizable network must encompass an infinite number of balancers, and, if the network has low contention, each token must traverse an average of $\Omega(n)$ balancers. Interestingly, these results

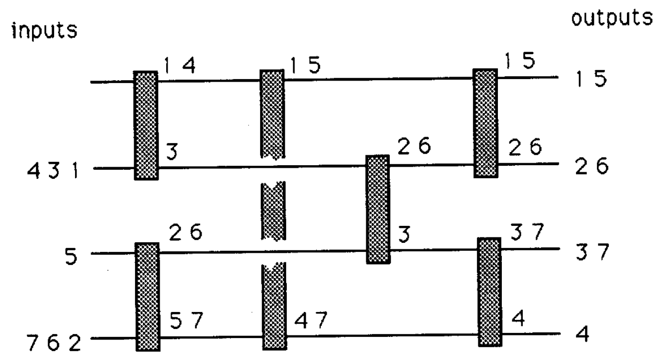

Figure 2: A sequential execution of an input sequence to a BITONIC[4] network.

hold also when the wires are FIFO. In practice, the space bound is not as alarming as it sounds, since we show later that it is possible to "fold" an infinite number of balancers into a finite data structure.

3.1 Lower Bounds on Size

We first show that the only linearizable counting network of finite width is the trivial one consisting of a single balancer. Given a nontrivial finite counting network, we construct an execution in which a later token overtakes an earlier token, resulting in non-linearizable behavior.

The following technical lemma is easily proved by induction on the number of balancers in the network.

Lemma 3.1. *Consider a balancing network of width w. If exactly p tokens enter on each input wire, then exactly p tokens will arrive at each input wire of each balancer.*

Theorem 3.2. *There is no linearizable counting network of width greater than two.*

Proof: We assume such a network and derive a contradiction. Let b be the last balancer on wire $w-1$. Send w tokens $p_0, \ldots, p_{w-1}$ sequentially through the network, where each p_i enters on input wire i. If a token arrives at balancer b, halt it on b's input wire, otherwise let it proceed until it takes a value. Lemma 3.1 implies that there is exactly one token on each input wire of b.

We claim that (1) the first token to visit b will not exit on wire $w-1$, and (2) one of these halted tokens is p_{w-1}. To see why, consider the state of the network before p_{w-1} enters. At least one token is halted before b. If all halted tokens resume their traversals, then the step property implies that one token will have emerged on each of wires $0, \ldots, w-2$, and none on $w-1$. A contradiction arises if either (1) the first token to visit b exits on $w-1$, or (2) there are two tokens halted at b, since one must exit on $w-1$.

Now let p_{w-1} resume its traversal, taking a value less than $w-1$, and send w more tokens $q_0, \ldots, q_{w-1}$ sequentially through the network, where each q_i enters on input wire i. As before, if a token arrives at balancer b, halt it on b's input wire, otherwise let it proceed until it takes a value. Each q_i follows the same path as p_i, and by similar reasoning, two q_i are halted before b, one being q_{w-1}. The remaining $w-2>0$ tokens will each take values greater than $w-1$. If q_{w-1} resumes its traversal, it will be the second token to visit b, hence it will take $w-1$, violating linearizability. ■

This proof actually proves a slightly stronger result. In the execution we constructed, no token overtakes another on a single wire, and therefore there is no non-trivial finite linearizable counting network even under the additional constraint that the wires between balancers are FIFO.

Corollary 3.3. *Any input wire of a linearizable counting network can be used only a bounded number of times.*

3.2 Lower Bounds on Time

In any network state, token p has *preferred path* u if p would traverse u if it were run in isolation until exiting the network. The *capacity* c of an execution in which n tokens concurrently traverse a network is defined to be the maximal number of tokens that arrive on any input wire.

The following lemma is implied by Lemma 3.1.

Lemma 3.4. *In any execution where no more than c tokens enter on any input wire, there are never more than c tokens on any wire.*

The capacity c of a network is the maximum capacity of any of its executions that involve only n concurrent tokens. Lemma 3.4 implies that in a network with capacity c, no more than c tokens arrive on any wire during an execution involving n tokens.

Theorem 3.5. *Consider a linearizable counting network for n processes with capacity c. In any quiescent state, the preferred path for any token p must traverse at least $\lceil n/c \rceil - 1$ balancers.*

Proof: Consider the following execution. Suppose the network is in a quiescent state, and $i-1$ is the last value taken by a token. For each token q distinct from p, run q in isolation until either

1. q is about to leave the filter with value k.
2. q is about to join p's preferred path.

We claim the first case cannot occur. Since the filter is in a quiescent state, all values less than i have been taken, and therefore q's preferred path must exit the filter on wire i. If q exits with a value k, then k must be greater than i, since q's path did not intersect p's preferred path. If q exits with $k > i$, however, then p can traverse the entire network and emerge with value i after q left the network with value $k > i$, violating linearizability. Therefore q's path must eventually intersect p's preferred path.

No more than $c-1$ tokens can share p's input wire. The remaining $n-c$ tokens will halt on an input wire to a balancer connected to p's preferred path. Lemma 3.4 implies that not more than c additional tokens can arrive at each such balancer. Since there are $n-c$ such tokens, there are at least $\lceil (n-c)/c \rceil = \lceil n/c \rceil - 1$ distinct balancers along p's preferred path. ∎

Theorem 3.6. *In any sequential execution, every token traverses at least $\lceil n/c \rceil - 1$ balancers.*

Proof: Initially, the network is quiescent, and Theorem 3.5 implies that the first token traverses at least $\lceil n/c \rceil - 1$ balancers. After each token leaves the network, the network returns to a quiescent state, and the same argument applies. ∎

Elsewhere, [4] it has been shown that the set of balancers traversed by a set of tokens does not depend on how transitions are interleaved, which implies:

Corollary 3.7. *In any execution, the average number of balancers traversed by each token is at least $\lceil n/c \rceil - 1$.*

Notice that the network capacity c is a measure of potential contention. If c is high, so is the maximum number of concurrent accesses to a balancer. These lower bounds therefore imply a reciprocal relation between contention, the number of tokens that can arrive concurrently at a balancer, and latency, the number of balancers a token must traverse.

4 Linearizable Counting Network Constructions

In this section we describe two linearizable counting network constructions. In the previous section, we proved that in any linearizable counting network, each input wire could be used only a finite number of times. In this section we give two constructions for networks in which each input wire is used only once. Each token chooses an input wire by traversing an auxiliary (non-linearizable) counting network (e.g., [4]), and uses the resulting value as the index of its input wire. Another way to visualize this two-part construction is to view the linearizable counting network as a "filter" that linearizes the outputs of a standard counting network. Our lower bounds from the previous section apply to the time and space complexity of the filter network. In particular, if each input wire is used only once, then $c = 1$, and each token must traverse an average of $\Omega(n)$ balancers. Although these constructions require an infinite number of balancers, we show in Section 5 how they can be "folded" into data structures of bounded size. Our first construction is *non-blocking:* it guarantees that some token always emerges after the network as a whole has taken a bounded number of steps, but it allows individual tokens to run forever without taking a value. The second construction is *wait-free*: it guarantees that every token emerges after taking a fixed number of steps.

4.1 The Skew Network

Our first filter is the SKEW network, illustrated in Figure 3. A SKEW-LAYER network is an unbounded balancing network consisting of a sequence of balancers b_i, for $0 \leq i$. For b_0, both input wires are network input wires. For all b_i, the north output wire is a network output wire, and the south output wire is the north input wire for b_{i+1}. A SKEW network of *layer depth* [2] d is constructed by layering d SKEW-LAYER networks so that the i^{th} output wire of one is the i^{th} input wire to the next. We say that a balancer b has *layer* i if it belongs to the i^{th} SKEW-LAYER component.

This filter is combined with a counting network as follows. Each token first traverses the counting network, and then uses the resulting value as the index of its input wire into the SKEW network. It is easy to show that the result is still a counting network.

Lemma 4.1. *Consider a counting network where n is a bound on the number of concurrent tokens. When a token takes a value v, then there are at most $n-1$ values less than v that are yet untaken.*

Proof: Suppose otherwise. A value is *missing* if no token has taken it. Let i be the least value to violate the claim. If we let the network quiesce, then all values less than i will be taken. Therefore every missing value corresponds to a token traversing the network, and the claim follows because there are at most n tokens in the network. ∎

Note that when a token takes v, it may not yet be determined which token will take which of the lower values.

A *northwest barrier* in a SKEW network is a sequence of balancers $b_0, \ldots, b_k$ such that the south output wire of b_i is joined to the north input wire of b_{i+1}, and each balancer's state is 1. Any token that approaches a northwest barrier will be diverted south below the barrier, effectively protecting all wires behind the barrier from late-arriving tokens.

Lemma 4.2. *If a token p exits a balancer on its south wire, then there is a northwest barrier starting from the token and continuing as far as it can go.*

Proof: By induction on i, the number of the wire on which p exited south. For $i = 0$ the result is immediate. Otherwise, assume the claim for $i-1$. Since p exited on the balancer's south wire, another token must already have visited this balancer. One of the two tokens must have traversed the south output wire of the preceding balancer, and hence must have exited south on wire $i-1$. The result now follows from the induction hypothesis. ∎

Lemma 4.3. *Let q be a token that enters the filter after token p has taken a value. If q traverses a lower wire (higher numbered) than p at layer k, then it does so at all layers greater than k.*

Proof: Assume otherwise. Then, p's path and q's must cross. The only way their paths can cross in this network is if they traverse a common balancer. Since each balancer is visited by only two tokens and since p got there first, p must exit on the north wire, and q on the south. ∎

Theorem 4.4. *If processes use a non-linearizable counting network to choose their input wires, then the* SKEW *filter is linearizable if $d \geq n-1$.*

Proof: The network is clearly a counter. We argue inductively that linearizability is preserved among all tokens that have entered the network on wires less than or equal to k. When $k = 0$, the result is immediate, so assume the result for wires less than $k > 0$.

Suppose that token p exits the network, and token q then enters the network and exits with a value less than p's. Lemma 4.3 implies that p entered the filter on a lower wire than q. The inductive hypothesis implies therefore that p enters the filter on wire k. There are two cases to consider: (1) p leaves some balancer on its south wire, and (2) p leaves every balancer on its north wire.

[2] Layer depth should not be confused with depth, which is infinite for SKEW.

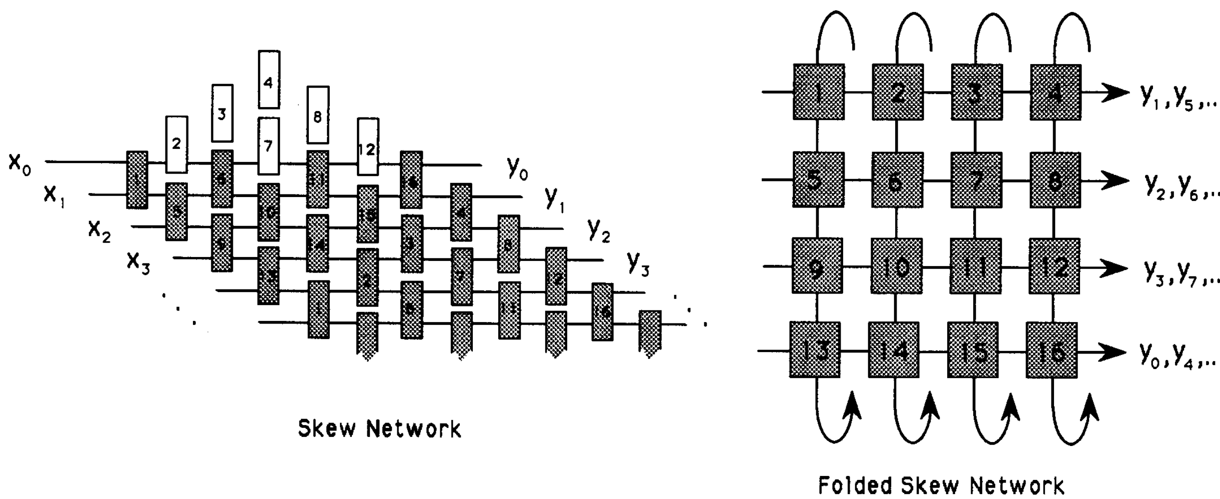

Figure 3: Skew Network and Folding

In the first case, Lemma 4.2 implies that there is a northwest barrier extending up to the top line. Lemma 3.4 implies that token q must be diverted south below the barrier. Lemma 4.3 implies therefore that q will take a value greater than p's, a contradiction.

In the second case, if $k \leq n-1$, then p goes north until it reaches the top line, and the result is immediate. Otherwise, if $k > n-1$, then p goes north on all $n-1$ balancers, and hence gets value $k-n+1$. Since $k > n-1$, Lemma 4.1 implies that at least $k-n+1$ tokens must have entered the filter on lines less than k and left it before p entered the skew. Therefore there exists a token r that exited the network before p entered the filter, and took a value $\geq k-n$. It follows that r exits the network before q entered it, and by the induction hypothesis, it took a lesser value than q, since otherwise we would have a linearizability violation among the first $k-1$ lines. But in this case, q's value must be smaller than p's and greater than r's, a contradiction. ■

Although the SKEW network permits starvation, the *average* path length is $O(n)$, so this filter is optimal up to a constant factor.

Lemma 4.5. *The average number of balancers traversed by any token in the* SKEW *filter is* $2n-2$.

Proof: In any quiescent state, k tokens have entered and exited the network on the top k wires. There are k wires of $2n-2$ balancers each, yielding an average path length of $2n-2$. ■

4.2 The Reverse-skew Network

Our second filter is the REVERSE-SKEW network. A REVERSE-LAYER network is the mirror image of the SKEW-LAYER. It consists of a sequence of balancers b_i, for $0 \leq i$. For b_0, both output wires are network output wires. For all b_i, $i > 0$, the south output wire is a network output wire, and the north output wire is the south input wire for b_{i-1}. A REVERSE-SKEW network of layer depth d is constructed by layering d REVERSE-LAYER networks so that the i^{th} output wire of one is the i^{th} input wire to the next.

Theorem 4.6. *The result of joining a counting network of width w to the* REVERSE-SKEW *filter is a linearizable counter if* $d \geq \lceil (n-1)/2 \rceil w - 1$.

The proof of this theorem is omitted for lack of space, but it is nearly identical to that of Theorem

4.4. It uses one additional observation, which is: Lemma 4.1 implies that there is no violation of linearizability between any two tokens that enter the filter on input wires that are of distance greater than $\lceil (n-1)/2 \rceil w - 1$. Therefore, the northwest barrier created when some token exits the network, need only protect against tokens that entered on input wires that are less than $\lceil (n-1)/2 \rceil w$ apart from its filter input wire.

As in Lemma 4.5, the average number of balancers traversed by any token in the REVERSE-SKEW filter is $2\lceil (n-1)/2 \rceil w - 2$.

Finally, the following lemma shows that the REVERSE-SKEW network is wait-free.

Lemma 4.7. *The number of balancers traversed by any token in the* REVERSE-SKEW *filter is at most* $2\lceil (n-1)/2 \rceil w + n - 3$.

5 Implementation

5.1 Implementing an Infinite Network

A balancer can be represented as a record with the following fields: *toggle* is a boolean value, and *north* and *south* are pointers which reference either other balancers, or *counter* cells. A process shepherds a token through the network by executing the procedure shown in Figure 4. It toggles the balancer's state by calling *fetch&complement*, which atomically complements the toggle field and returns the old value. Based on the toggle state, it goes either north or south. When it encounters a counter, it atomically increments it by w and returns the old value. Both complementing and incrementing can be accomplished either by a short critical section, or by a *read-modify-write* operation if the hardware supports it.

Note that balancers are bounded, but counters, by definition, are not.

5.2 Folding the Network

We now show how to represent the infinite SKEW network using a finite network. (The construction for the REVERSE network is omitted, since it is nearly identical.) We first define a coordinate system for identifying balancers. Each balancer is denoted $b_{i,j}$, where i ranges from 0 to infinity, and j ranges from 0 to $d-1$, where d is the network's layer depth. Balancer $b_{i,0}$ is the first balancer whose

```
balancer = [toggle: integer, north, south: pointer]
visit(b: pointer) returns(integer)
   loop until counter(b)
         i := fetch&complement(b.toggle)
         if i = 0
            then b := b.north
            else  b := b.south
            end if
         end loop
   v := fetch&add(b.state, w)
   return v
   end visit
```

Figure 4: Code for Traversing an Infinite Network

north output wire is on row i, $b_{i,d-1}$ is the last balancer on row i (equivalently, whose north output wire is on row i), and $b_{i,j}$ is balancer on layer j and on row i.

A *folded* SKEW network is a w by d array of *multibalancers* $c_{i,j}$. $c_{0,0}$ has two input wires, each $c_{i,0}$, $i > 0$, has one input wire, and each $c_{i,d-1}$ has one output wire. For $0 \le i \le w$ and $0 \le j < d$, there is one wire from $c_{i,j}$ to $c_{i+1,j}$, where index arithmetic is mod w; and for $0 \le i \le w$ and $0 \le j < d-1$, there is also one wire from $c_{i,j}$ to $c_{i,j+1}$. The multibalancer $c_{i,j}$ simulates each of the balancers $b_{i,j}, b_{i+w,j}, b_{i+2w,j}, \ldots$. The folding for a network with $w = 3$, and $d = 3$ is illustrated in Figure 3.

Like a balancer, a multibalancer can also be represented as a record with *toggle*, *north*, and *south* fields. The *north* and *south* fields are still pointers to the neighboring multibalancers or counters, but the *toggle* component is more complex, since it encodes the toggle states of an infinite number of balancers. The following theorem shows that this infinite sequence has a simple structure.

Theorem 5.1. *Let* $s_0, s_1, \ldots$ *be the toggle states of* $b_{i,j}, b_{i+w,j}, \ldots$. *If there are* m *active processes, then there are at most* $2m+2$ *values of* k *such that* $s_k \ne s_{k+1}$.

Proof: We argue by induction on m. Let N be the total number of tokens in the network, including those whose traversals have completed. If $m = 0$, the network is quiescent, the first $\lfloor N/2 \rfloor$ balancers have been visited by 2 tokens, the next by $N \pmod 2$ tokens, and the rest by no tokens. Assume the result for $m-1$ active tokens, and consider the situation where there are m active tokens. Choose any active token, run it to completion, and let s'_k be the new toggle state of balancer $(i+kw, j)$. By the induction hypothesis, there are at most $2m$ values of k such that $s'_k \neq s'_{k+1}$. The result follows because there are at most two k such that $s'_k \neq s'_{k+1}$ and $s_k = s_{k+1}$. ■

Corollary 5.2. *There are at most $2n+2$ values of k such that $s_k \neq s_{k+1}$.*

The *toggle* component of the multibalancer $c_{i,j}$ can therefore be treated as a set containing (at most) $2n+2$ pairs (k, s_k) such that $b_{i+kw,j} \neq b_{i+(k-1)w,j}$, and an additional pair of $(0, s_0)$. This set could be implemented with a short critical section (which introduces a small likelihood of blocking) or it could be implemented without blocking using *read-modify-write* operations as discussed elsewhere [12].

6 Other Solutions

So far, linearizability seems like an expensive property: linearizable counting networks must encompass an infinite number of balancers and each traversal takes linear average time. Nevertheless, we now show that it is still possible to construct efficient linearizable counters by augmenting the counting network structure with a simple *waiting* primitive.

Just as before, we construct a network from two component networks. One is an arbitrary non-linearizable counting network (e.g., [4]), and the other is a *waiting filter*. Informally, the waiting filter is a kind of barrier, where each token waits for the tokens with lower values to "catch up." A token leaves the filter only when all lower values have been assigned, guaranteeing that every token that enters the network later will receive a higher value. More precisely, a waiting filter is an n-element array of boolean values, called toggle bits, where indexing starts from 0. Define the function *toggle*(v) to be $\lfloor (v/n) \rfloor \bmod 2$. When a token exits the non-linearizable counting network with value v, it awaits its predecessor by going to location $(v-1) \bmod n$ in the filter, and waiting for that location to be set to *toggle*$(v-1)$. When this event occurs, it notifies its successor by setting location v to *toggle*(v), and then it returns its value.

Lemma 6.1. *When token p with value v sets its toggle bit, every token that takes a lesser value has also set its toggle bit.*

Proof: Let p be the first token to violate this property. It must have seen location $v-1 \bmod n$ in the filter set to *toggle*$(v-1)$, but that value was written by the token with value $v-2kn-1$, for some $k>0$. In particular, the token with value $v-n-1$ has not yet written its toggle bit, and by the induction hypothesis, neither have the n tokens with values $v-n \ldots v-1$. Since there can be at most n tokens in the network, we have a contradiction. ■

Corollary 6.2. *The waiting filter is a linearizable counter.*

Note that the waiting filter has very low contention, since each position is concurrently read by only one process and written by only one process. As an aside, we remark that waiting for a location to change value is very efficient on a cache-consistent architecture, since the waiting process simply rereads the value in its cache, and does not need to access the shared memory until the cache is invalidated.

The principal disadvantage of this scheme is that it is blocking. A failure or delay by any single token will result in the failure or delay of all the other tokens. Preliminary investigation of the waiting filter shows that it performs well in experiments, where failures are non-existent and timing anomalies are rare.

7 Acknowledgements

We thank Cynthia Dwork, Serge Plotkin, and Vaughan Pratt for their many constructive comments.

References

[1] E. Aharonson and H. Attiya. Counting networks with arbitrary fan out. Technical Report 679, The Technion, June 1991.

[2] A. Agarwal and M. Cherian Adaptive backoff synchronization techniques *16th Symposium on Computer Architecture*, June 1989.

[3] T.E. Anderson. The performance implications of spin-waiting alternatives for shared-memory multiprocessors. Technical Report 89-04-03, University of Washington, Seattle, WA 98195, April 1989. To appear, IEEE Transactions on Parallel and Distributed Systems.

[4] J. Aspnes, M.P. Herlihy, and N. Shavit. Counting networks and multi-processor coordination In *Proceedings of the 23rd Annual Symposium on Theory of Computing*, May 1991, New Orleans, Louisiana.

[5] T.H. Cormen, C.E. Leiserson, and R. L. Rivest. Introduction to Algorithms MIT Press, Cambridge MA, 1990.

[6] C.S. Ellis and T.J. Olson. Algorithms for parallel memory allocation. *Journal of Parallel Programming*, 17(4):303–345, August 1988.

[7] M. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM* 32(2):374-382, April 1985.

[8] D. Gawlick. Processing 'hot spots' in high performance systems. In *Proceedings COMPCON'85*, 1985.

[9] J. Goodman, M. Vernon, and P. Woest. A set of efficient synchronization primitives for a large-scale shared-memory multiprocessor. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989.

[10] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer – designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1984.

[11] A. Gottlieb, B.D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[12] M.P. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, Seattle, WA, March 14-16 1990.

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

[14] M. Klugerman, An $O(\log n \log\log n)$ counting network. Unpublished Manuscript, MIT-LCS, June 1991.

[15] D. Kranz, R. Halstead, and E. Mohr, "Mul-T, A High-Performance Parallel Lisp", *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989, pp. 81–90.

[16] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization on multiprocessors with shared memory. In *Fifth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1986.

[17] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[18] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979

[19] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. Technical Report Technical Report 342, University of Rochester, Rochester, NY 14627, April 1990.

[20] C.H. Papadimitriou. The serializability of concurrent database updates *Journal of the ACM*, 26(4):631-653, October 1979.

[21] G.H. Pfister et al. The IBM research parallel processor prototype (RP3): introduction and architecture. In *International Conference on Parallel Processing*, 1985.

[22] G.H. Pfister and A. Norton. 'hot spot' contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(11):933–938, November 1985.

[23] H.S. Stone. Database applications of the fetch-and-add instruction. *IEEE Transactions on Computers*, C-33(7):604–612, July 1984.