Linearizable Counting Networks

Maurice Herlihy   Nir Shavit*   Orli Waarts†

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/12   December 4, 1991

Abstract

The *counting problem* requires n asynchronous processors to assign themselves successive values. A solution is *linearizable* if the order of the values assigned reflects the real-time order in which they were requested. Linearizable counting lies at the heart of concurrent timestamp generation, as well as concurrent implementations of shared counters, FIFO buffers, and similar data structures.

We consider solutions to the linearizable counting problem in a multiprocessor architecture in which processors communicate by applying read-modify-write operations to a shared memory. Linearizable counting algorithms can be judged by three criteria: the memory contention produced, whether processors are required to wait for one another, and how long it takes a processor to choose a value (the latency). A solution is *ideal* if it has low contention, low latency, and it eschews waiting. The conventional software solution, where processes synchronize at a single variable, avoids waiting and has low latency, but has high contention. In this paper we give two new *counting network* constructions, one with low latency and low contention, but that requires processors to wait for one another, and one with low contention and no waiting, but that has high latency. Finally, we prove that these trade-offs are inescapable: an ideal linearizable counting algorithm is impossible. Since ideal non-linearizable counting algorithms exist, these results establish a substantial complexity gap between linearizable and non-linearizable counting.

*MIT Lab. for Computer Science. Supported by 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.

A preliminary version of this work appeared in the *Proceedings of the* 32−*nd Annual Symposium on Foundations of Computer Science,* San Juan, Puerto Rico, October 1991, pp. 526-535. [5]

... replied the businessman. "I count them and recount them. It is difficult but I am a man who is naturally interested in matters of consequence."

— Antoine de Saint-Exupéry, The Little Prince

1 Introduction

In the *counting problem*, asynchronous concurrent processors repeatedly assign themselves successive values, such as integers or locations in memory. The *linearizable* counting problem requires 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 processor P is assigned a value before processor Q requests one, then P's value must be less than Q's. Linearizable counting lies at the heart of a number of basic problems, such as concurrent time-stamp generation, concurrent implementations of shared counters, FIFO buffers, and similar data structures (e.g. [7, 11, 17, 25]).

The requirement that the values chosen reflect the real-time order in which they were requested is called *linearizability* [14]. The use of linearizable data abstractions greatly simplifies both the specification and the proofs of multiple instruction/multiple data (MIMD) shared memory algorithms. As discussed in more detail elsewhere [14], 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 [21].

Linearizable counting algorithms can be judged by three criteria:

- *Contention:* Because of limitations on processor-to-memory bandwidth, performance suffers when too many processors attempt to access the same memory location at the same time. Such "hot-spot" contention is well-documented, and has been the subject of extensive research both in hardware [2, 10, 11, 16, 22] and in software [3, 8, 9, 20, 25].

- *Latency:* The time needed to choose a value is strongly affected by the number of variables a processor must access. We will show that (not surprisingly) there is an inherent (inverse) relationship between the maximum contention at a variable and the number of variables accessed.

- *Waiting:* Algorithms that require later processors to wait for earlier processors are not robust — the failure or delay of a single processor will result in halting or delays in non-faulty processors. All else being equal, it is preferable to choose algorithms that ensure that some processes make progress even when others halt in arbitrary locations.

Informally speaking, a linearizable counting algorithm is *ideal* if it has low contention, low latency, and it eschews waiting. In this paper, we will show that no ideal linearizable counting algorithm exists, but that it is possible to satisfy any two out of the three criteria.

First, consider the naive solution in which all n processors increment a single shared variable using a read-modify-write [1] operation. This algorithm has low latency (a single variable), it eschews waiting (the read-modify-write is assumed to be atomic), but very high contention. (For more complete documentation of the performance problems of the single-variable solution see Anderson et al. [3] and Graunke and Thakkar [12]. Also see the experimental results described below.)

Elsewhere [4], we have proposed low-contention solutions to the (non-linearizable) counting problem based on a new class of data structures called *counting networks*. In this paper, we show how counting networks can be adapted to solve linearizable counting. We first give a construction that employs a counting network of depth $O(\log n)$. This construction has low contention and low latency, but it requires processes to wait for one another. We then give a two alternative counting network constructions that do not require waiting. The first employs a network of depth $O(n)$, and it guarantees that some non-halted processor makes progress. The second employs a network of depth $O(n^2)$, and it guarantees that every non-halted processor makes progress.

Finally, we prove that these trade-offs are a fundamental aspect of linearizable counting: any low-contention network that does not rely on waiting must have depth $\Omega(n)$, where n is the number of processes. Since non-linearizable counting does have ideal solutions [4] with low contention, polylogarithmic depth, and no waiting, this result establishes a substantial complexity gap between linearizable and non-linearizable counting.

[1] A read-modify-write operation [11] atomically reads the value of a memory location, modifies it, writes it back, and returns the location's old value.

1.1 Background

A counting network, like a sorting network [6], 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. Unlike a sorting network, 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 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 wire 2 and leaves on wire 1, the second leaves on wire 2, and so on. (The reader is encouraged to try this for him/herself.) Thus, if on the i-th output wire 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!

Counting networks achieve a high level of throughput by decomposing interactions among processors into pieces that can be performed in parallel, effectively reducing memory contention. In [4], two $O(\log^2 n)$ depth counting network designs were presented. Aharonson and Attiya [1] have recently proved several fan-out and cyclicity properties of such networks, and Klugerman and Plaxton [15] have shown an explicit network construction of depth $O(c^{\log^* n} \log n)$ for some small constant c, and an existential proof of a network of depth $O(\log n)$.

Unfortunately, all known counting network constructions [1, 4, 15] are not linearizable. It is even possible for a processor to shepherd two tokens through a network, one after the other, and by suitable overtaking, have the second token receive the lesser value. Can counting networks solve linearizable counting?

1.2 Overview

In Section 3, we propose a simple data structure, called a WAITING-FILTER, which transforms any low-contention non-linearizable counting protocol into a low-contention linearizable counting protocol. The resulting protocol, however, requires that later processors wait for earlier processors to complete, implying that the failure or delay of a single processor will produce halting or delay in the other, non-faulty processors. Nevertheless, we give experimental evidence that in the absence of such timing anomalies, the transformed protocol performs as well as the original.

In Section 4, we present two linearizable counting protocols that do not require processors to wait for one another. Each of these protocols uses two counting networks: a standard non-linearizable counting network, and a "filter" network. In the first SKEW network each token traverses an *average* of $O(n)$ balancers, but an individual token may be forced along an infinite path if it is infinitely often overtaken. The second REVERSE-SKEW network construction guarantees that every token emerges after traversing $O(n^2)$ balancers, hence starvation is impossible. If implemented directly in terms of balancers, these filter networks would have infinite size, so we give a simple technique for "folding" these infinite networks onto finite data structures.

In Section 5, we prove that the tradoffs among our constructions is inherent. In any low-contention linearizable counting network, a token must traverse an average of $\Omega(n)$ gates before taking a value. In [15] it was shown that there exist width n non-linearizable counting networks in which each token traverses at most $O(\log n)$ balancers. Our results therefore establish a substantial complexity gap between the class of linearizable and non-linearizable counting networks – in other words, linearizability comes at a cost.

2 A Brief Introduction to Counting Networks

In this section we provide an introduction to counting networks. A more complete discussion of the properties of counting networks can be found in [1, 4].

Counting networks belong to a larger class of networks called balancing networks, constructed from wires and computing elements called balancers. A *balancer* is a computing element with two input wires, denoted as the *north* and *south* wires (and indexed by 0 and 1), and two output wires, similarly named. Tokens arrive on the balancer's input wires at arbitrary times and

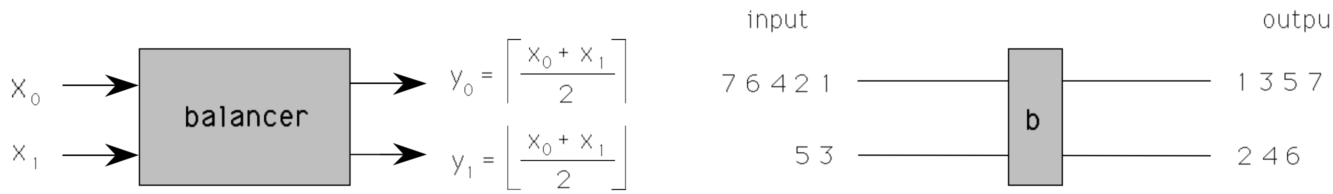

Figure 1: A Balancer.

are output on its output wires. Intuitively, one may think of a balancer as a toggle mechanism, that given a stream of input tokens, repeatedly sends one token to the left output wire and one to the right, effectively balancing the number of tokens that have been output on its output wires. 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 sent on its i-th output wire. Throughout the paper we will abuse this notation and use x_i (y_i) both as the name of the i-th input (output) wire and a count of the number of tokens received on the wire.

Let the state of a balancer at a given time be defined as the collection of tokens on its input and output wires. For the sake of clarity we will assume that tokens are all distinct. We denote by the pair (t, b), the state *transition* in which the token t passes from an input wire to an output wire of the balancer b.

We can now formally state the safety and liveness properties of a balancer:

1. In any state $x_0 + x_1 \geq y_0 + y_1$ (i.e. a balancer never creates output tokens).

2. 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 the sets of input and output tokens are the same. In any quiescent state, $x_0 + x_1 = y_0 + y_1 = m$.

3. In any quiescent state, $y_0 = \lceil m/2 \rceil$ and $y_1 = \lfloor m/2 \rfloor$.

A *balancing network* of width w is a collection of balancers, where output wires are connected to input wires, having w designated input wires $x_0, x_1, .., x_{w-1}$ (which are not connected to output wires of balancers), w designated output wires $y_0, y_1, .., y_{w-1}$ (also unconnected), and containing no cycles. Let the state of a network at a given time be defined as the union of the states of all its component balancers. The safety and liveness of the network follow naturally from the above network definition and the properties of balancers, namely, that it is always the case that $\sum_{i=0}^{w-1} x_i \geq \sum_{i=0}^{w-1} y_i$, and for any finite sequence of m input tokens, within finite time the network reaches a *quiescent* state, i.e. one in which $\sum_{i=0}^{w-1} y_i = m$.

It is important to note that we make no assumptions about the timing of token transitions from balancer to balancer in the network — the network's behavior is completely asynchronous. Although balancer transitions can occur concurrently, it is convenient to model them using an interleaving semantics in the style of Lynch and Tuttle [19]. An *execution* of a network is a finite sequence $s_0, e_1, s_1, \ldots e_n, s_n$ or infinite sequence $s_0, e_1, s_1, \ldots$ of alternating states and balancer transitions such that for each (s_i, e_{i+1}, s_{i+1}), the transition e_{i+1} carries state s_i to s_{i+1}. A *schedule* is the subsequence of transitions occurring in an execution. A schedule is *valid* if it is induced by some execution, and *complete* if it is induced by an execution which results in a quiescent state. A schedule s is *sequential* if for any two transitions $e_i = (t_i, b_i)$ and $e_j = (t_j, b_j)$, where t_i and t_j are the same token, then all transitions between them also involve that token. In other words, tokens traverse the network one completely after the other.

In a MIMD 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 the following Subsection 2.1). The limitation on the number of concurrent processors translates into a limitation on the number of tokens concurrently traversing the network:

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

We define the *depth* of a balancing network to be the maximal depth of any wire, where the depth of a wire is defined as 0 for a network input wire, and $\max_{i \in \{0..1\}}(\mathit{depth}(x_i) + 1)$ for the output wires of a balancer having input wires x_i, $i \in \{0..1\}$.

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 wire i to the k-th token exiting on it, is called the token's *value*. We can now state the following simple yet useful lemma:

Lemma 2.1 *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. If we let the network quiesce, then all values less than v 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.

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.

> 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 $value(a) < value(b)$.

Though outside the scope of this paper, this definition can easily be shown to meet the linearizability definition of [14]. [2]

2.1 Implementing a Counting Network

In this paper, we assume that counting networks are implemented on a multiprocessor in which processors communicate by applying *read-modify-write* operations to a shared memory. The counting network is implemented

[2] Informally, this would amount to showing that the history of all processor's requests (of *values*) and replies is equivalent to a sequential history which is consistent with all non-concurrent pairs of request-reply events.

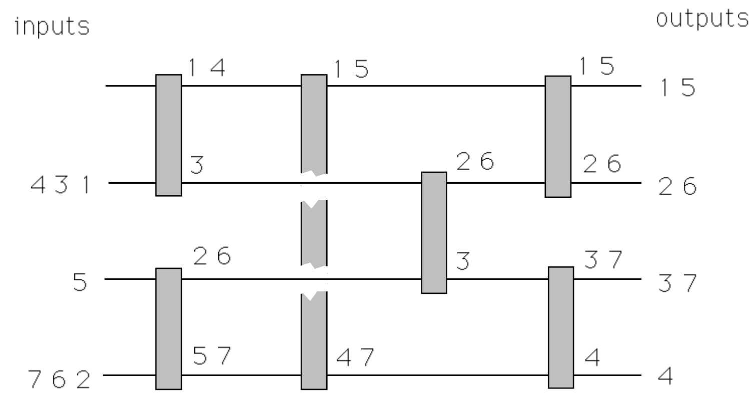

Figure 2: A sequential execution of an input sequence to a BITONIC[4] network.

as a data structure in memory. A balancer is represented as a record with the following fields: *toggle* is a boolean value (initially *True*) and *north* and *south* are pointers which reference either other balancers, or *counter* cells. Processors shepherd tokens through the network by executing the code shown in Figure 3. Each processor 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 to the north or south successor. When it encounters a counter, it atomically increments it by w and returns the old value. Note that balancers use only bounded size memory, but counters, by definition, do not.

3 The Waiting Filter

We begin by proposing a solution with low contention and low latency, but that requires processors to wait for one another. The key idea is simple: each token exiting the network simply waits for a token to take the next lower value. This solution is therefore not robust, since a failure or delay by one processor will force halting or delays in other, non-faulty processors. Nevertheless, contention is low, since every processor waits on a separate location.

We use two component data structures. The first is a non-linearizable counting network of arbitrary width (e.g., the bitonic or periodic networks [4]), and the other is a WAITING-FILTER of width n. Informally, the WAITING-FILTER is a kind of barrier, to be traversed after the non-linearizable network,

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

Figure 3: Code for Traversing a Counting Network of width w

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 *phase bits*, where indexing starts from 0. Define the function $phase(v)$ to be $\lfloor (v/n) \rfloor \bmod 2$. We construct the new network by having tokens first traverse the counting network and then access the WAITING-FILTER. 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 $phase(v-1)$. When this event occurs, it notifies its successor by setting location v to $phase(v)$, and then it returns its value.

Lemma 3.1 *When token p with value v sets its phase bit, every token that takes a lesser value has also set its phase bit.*

Proof: Assume by way of contradiction that p is the token of lowest value v to violate this property. It must have seen location $v-1 \bmod n$ in the filter set to $phase(v-1)$, a value that could only have been written by the token with value $v-2kn-1$, for some $k>0$. In particular, a token with value $v-n-1$ could not have yet written its phase bit, and thus by assumption, neither could any token with one of the n values $v-n \ldots v-1$. By the step property of the non-linearizable counting network, since a token with value v exited the network, there must be at least $n+1$ tokens currently traversing the network or past the network and before the phase change, that will take on the values $v-n-1, v-n, \ldots, v-1$. Since by definition there can be at most n tokens concurrently in the construct, we have a contradiction. ■

Corollary 3.2 *The* WAITING-FILTER *is a linearizable counter.*

3.1 Performance

We now make a brief digression to explore the performance of the WAITING-FILTER. In a cache-consistent architecture, the WAITING-FILTER has very low contention, since each position is concurrently read by only one processor and written by only one processor. Waiting for a location to change value does not produce memory contention, since the waiting processor simply rereads the value in its cache, and does not need to access the shared memory until the cache is invalidated.

To explore how the waiting filter performs in practice, we compare the performance of a bitonic counting network with and without the WAITING-FILTER. As a control, we also compare the performance of a conventional spin lock, implemented by an in-line compiled "test&test&set" [24] loop. These implementations were done in C on an Encore Multimax.

For each network, we measured the elapsed time necessary for a 2^{20} (approximately a million) tokens to traverse the network, controlling the level of concurrency. In Figure 4, the horizontal axis represents the number of processes executing concurrently. When concurrency is 1, each processor runs to completion before the next one starts. The number of concurrent processes increases until all sixteen processes execute concurrently. The vertical axis represents the elapsed time (in seconds) until all 2^{20} tokens have traversed the network. Each point on each curve represents the same amount of work.

At low levels of concurrency, the spin lock outperforms the networks, but as concurrency increases, the spin lock's throughput diminishes dramatically, while the networks' throughputs eventually increase. The throughputs of the linearizable and non-linearizable networks are essentially the same.

4 Linearizable Counting Without Waiting

In this section, we present two linearizable counting protocols with low contention that do not require processes to wait for one another. Just as in the waiting construction given in the previous section, each token traverses a non-linearizable counting network followed by a "filter" network. The first protocol is *non-blocking:* it guarantees that some token always emerges after the system as a whole has taken a bounded number of steps, but it allows individual tokens to run forever without taking a value (starvation). The second construction is *wait-free*: it guarantees that every token emerges after taking a fixed number of steps (no starvation). Both networks have high latency, with depth $\Omega(n)$.

4.1 The Skew Network

Our first construction is based on the SKEW filter, a balancing network illustrated in the right-hand-side of Figure 5 (for now, ignore the empty balancers and the numeric labels). A SKEW-LAYER network is an unbounded size 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 balancing network with a *layer depth* [3] of 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 non-linearizable counting network as follows. Each token first traverses the non-linearizable counting network, and then uses the resulting value as the index of its input wire into the infinite SKEW filter network. For brevity, we refer to these two data structures as the *combined* SKEW network, even though the ensemble is not, strictly speaking, a counting network.

The correctness of our constructions is based on the following technical lemma, easily proven by induction on the number of balancers in a balancing network.

Lemma 4.1 *For any balancing network, if exactly c tokens enter on each input wire, then exactly c tokens will arrive at each input wire of each balancer.*

Corollary 4.2 *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 an execution in which n tokens concurrently traverse a network is defined to be the maximal number of tokens that arrive on any

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

input wire. Let the capacity c of a network be the maximum capacity over all executions. Corollary 4.2 implies that in a network with capacity c, no more than c tokens arrive on any internal or output wire during an execution involving n tokens.

In the SKEW filter, the capacity c is 1, that is, at most one token enters/exits on each of a balancer's input/output wires. We can thus define the *toggle state* of a balancer to be the number of tokens it has output. Let a *northwest barrier* starting in balancer b_k be a sequence of balancers $b_k, \ldots, b_0$, all in toggle state 2, where the north input wire of every b_i is the south output wire of b_{i-1}, and where b_0's north input is wire 0. It immediately follows from Lemma 4.1 that any token that approaches a balancer in a northwest barrier will be diverted below the barrier, effectively protecting all wires behind the barrier from late-arriving tokens.

Lemma 4.3 *If a token p exits a balancer b on its south wire, then there is a northwest barrier starting from b.*

Proof: By induction on i, the number of the wire on which p exited south from a balancer b. For $i = 1$ the result is immediate. Otherwise, assume the claim for $i - 1$. Since p exited on the b's south wire, another token must already have visited b. By Lemma 4.1, one of the two tokens must have come from b's north input wire, the south output wire of a preceding balancer, hence it must have exited south on wire $i - 1$. The result now follows from the induction hypothesis. ■

Lemma 4.4 *Let q be a token that enters the filter after token p has taken a value. If q traverses a higher numbered wire 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 two paths can cross in the SKEW filter is if they traverse a common balancer. By Lemma 4.1, each balancer is visited by only two tokens and since p got there first (i.e. in toggle state 0), p must exit on the north wire, and q on the south. ■

Corollary 4.5 *Let q be a token that enters the filter after token p has taken a value. If p and q pass through a common balancer, then q will take a higher value than p.*

Lemma 4.6 *The protocol ensures that the outputs of the* SKEW *filter have the step property in any quiescent state.*

Proof: In a quiescent state, all $0 \leq k$ tokens entering the combined network must have exited. By definition, the outputs of the non-linearizable counting network part have the step property. This implies that exactly k tokens have arrived on the k lower-numbered input wires of the SKEW filter. By simple induction on the layers of the SKEW filter, if k tokens enter on the k lower input wires, they will exit on the k lower output wires. ■

Lemma 4.7 *If processors use a non-linearizable counting network to choose their input wires, then for a* SKEW *filter of layer depth d, where $d \geq n-1$, 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).*

Proof: We argue inductively that this property is preserved among all tokens that have entered the SKEW filter network on wires less than or equal to k. When $k = 0$, the result is immediate, so assume the result for wires less than $k > 0$.

We prove the result for wires less than or equal to k by way of contradiction. Assume that token p exits the network, and token q then enters the network and exits with a value less than p's. Lemma 4.4 implies that q entered the filter on a lower numbered wire than p. The inductive hypothesis implies therefore that p enters the filter on wire k. There are two cases to consider: (1) p leaves some balancer b on its south wire, and (2) p leaves every balancer on its north wire.

In the first case, Lemma 4.3 implies that there is a northwest barrier extending from b to wire 0, and the token q must be diverted south (below the barrier) to higher numbered lines. Lemma 4.4 implies therefore that q will take a value greater than p's, a contradiction.

In the second case, if $k \leq n - 1 = d$, then p goes north until it reaches wire 0, and the result is immediate. Otherwise, if $k > n - 1$, then p goes north on $n - 1$ balancers, and hence gets value $k - n + 1$. Since $k > n - 1$, Lemma 2.1 applied to the non-linearizable counting network implies that at least $k - n + 1$ tokens must have entered the SKEW filter on lines less than k and left it before p entered. Therefore, since by Lemma 4.1 only one token can exit on a given output wire of the filter, 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 value $\geq k - n + 1$ and greater than r's value of $k - n$, a contradiction. ■

Theorem 4.8 *This protocol solves linearizable counting if the* Skew *filter has layer depth greater than or equal to* $n - 1$.

Proof: The outputs of the combined Skew network satisfy the step property in quiescent states (Lemma 4.6). The proof that the network is linearizable follows from Lemma 4.7 since for any token entering the combined network, its traversal interval through the *Skew* filter is a subinterval of its traversal interval through the whole network. ■

Although the combined Skew network permits starvation, the *average* traversal path length is $O(n)$.

Lemma 4.9 *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 lower numbered 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 combined 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. The protocol is the same as before: each token traverses the non-linearizable counting network, and uses its output value to choose the input wire into the Reverse-skew filter.

Theorem 4.10 *The protocol solves linearizable counting if the non-linearizable counting network has width w and the* Reverse-skew *filter has layer depth greater than or equal to* $\lceil (n-1)/2 \rceil w - 1$.

The proof of this theorem is omitted because it is nearly identical to that of Theorem 4.8. It uses one additional observation, which is: Lemma 2.1 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.

The following lemma shows that the Reverse-skew protocol is wait-free.

Lemma 4.11 *The number of balancers traversed by any token in the* Reverse-skew *filter is at most* $2\lceil (n-1)/2 \rceil w + n - 3$.

Proof: Note that a token can exit on the south end of at most $\lceil (n-1)/2 \rceil w - 1$ balancers. The number of the output wire on which a token exits is at most $n-1$ smaller than the number of the token's input wire in the filter, and therefore, a token can exit on the north end of at most $n - 1 + \lceil (n-1)/2 \rceil w - 1$ balancers, and the claim follows. ■

As in Lemma 4.9, the average number of balancers traversed by any token in the Reverse-skew filter is $2\lceil (n-1)/2 \rceil w - 2$. Note that if $c = 1$ then $n = w$ and the depth of the network is $O(n^2)$.

4.3 Implementing an Infinite Network

We now show how to represent the infinite Skew filter using a finite network. (The construction for the Reverse-skew filter 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 $d-1$ in a network of layer depth d. Balancer $b_{i,0}$ is the first balancer whose 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 filter network is a w width by d depth array of *multibalancers* $c_{i,j}$. Each $c_{0,0}$ has two input wires, $c_{i,0}$, $i > 0$, has one input wire, and each $c_{i,d-1}$ has one output wire. For $0 \leq i \leq w$ and $0 \leq j < d$, there is one wire from $c_{i,j}$ to $c_{i+1,j}$, where index arithmetic is mod w; and for $0 \leq i \leq w$ and $0 \leq j < d-1$, 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 of a Skew network of layer depth $d = 4$ into a folded network with $w = 4$ and $d = 4$ is illustrated in Figure 5.

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 4.12 *Let $s_0, s_1, \ldots$ be the toggle states of $b_{i,j}, b_{i+w,j}, \ldots$ in* SKEW *(the ones represented by a multibalancer $c_{i,j}$). If there are $m \leq n$ tokens traversing the* SKEW *filter, then there are at most $2m + 2$ values of k such that $s_k \neq s_{k+1}$.*

Proof: We argue by induction on m, the number of tokens concurrently traversing the network. Let N be the total number of tokens that are traversing or have completed traversing the network. If $m = 0$, the SKEW network is quiescent, implying that the first $\lfloor N/2 \rfloor$ balancers have been visited by 2 tokens, the next by $N \bmod 2$ tokens, and the rest by no tokens. Assume the result for $m - 1$ tokens concurrently traversing the network, and consider the situation where there are m tokens traversing it. Choose any traversing token, run it to completion, and let s'_k be the new toggle state of balancer $b_{i+kw,j}$. By the induction hypothesis, there are at most $2m$ values of k such that $s'_k \neq s'_{k+1}$. The result follows because with the addition of one more token, there are at most two k values such that $s_k \neq s_{k+1}$ and $s'_k = s'_{k+1}$ ∎

Since the number of concurrently traversing tokens m is always bounded by n, we have that:

Corollary 4.13 *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 [13].

5 Lower Bounds

We now show that it is impossible to construct an ideal linearizable counting algorithm, one with low contention, low latency, and without waiting. We give two results. The first concerns counting networks: first, any non-trivial[4]

[4]The *trivial* counting network consists of a single balancer.

non-waiting linearizable counting network must have an infinite number of balancers, implying that the "folding" structure employed in the previous section's filter constructions is, in a sense, inescapable. The second concerns linearizable counting in general: in *any* non-waiting protocol, whether based on counting networks or not, contention and latency are inversely related.

The lower bound on the number of balancers is not as alarming as it sounds, since we have shown it is possible to "fold" an infinite number of balancers into a simple finite data structure. The time bound is more significant: in a low-contention non-waiting network, any processor must traverse an average of $\Omega(n)$ balancers before choosing a value. There exist non-linearizable counting networks with polylogarithmic depth [1, 4, 15], and therefore non-waiting linearizable counting networks will always have lower throughput than their non-waiting non-linearizable counterparts.

5.1 Lower Bounds on Size

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

Theorem 5.1 *There is no non-blocking finite-width 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 4.1 implies that there is exactly one token on each input wire of b.

One of the halted tokens on b's input wires 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 exactly one token will have emerged on each of the wires $0, \ldots, w-2$, and none on $w-1$. Thus p_{w-1} must exit on wire $w-1$ and therefore is halted on one of b's input wires.

Now let p_{w-1} resume its traversal, taking a value less than $w-1$ (since there is at least one more halted token on the input wires to b), 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. ■

We have shown 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 first-in-first-out.

Corollary 5.2 *Any input wire of a linearizable counting network can be used only a bounded number of times.*

Proof: Suppose otherwise. The step property requires that each output wire of an infinite-width network be traversed no more than once in any finite execution. Consider a sequential execution in which token p enters on input wire i, runs uninterruptedly through the network, and emerges after d steps on output wire j. If we run 2^d additional tokens sequentially from input wire i, then the last token will follow exactly the same path as p, since the state of each balancer along the path will have been reset. Now two tokens have traversed output wire j, violating the step property. ■

5.2 Lower Bounds on Time

In this section, we prove some fundamental lower bounds for *any* linearizable counting protocol that does not use waiting, whether or not it relies on counting networks. A *protocol* is defined as follows: each processor applies *read-modify-write* operations to a sequence of variables and then chooses a value. A processor may choose the next variable based on the values of earlier variables, but some processor must decide after a finite number of steps (no waiting). The protocol's *latency* is the maximum number of variables any processor visits before choosing its value. A protocol is *quiescent* if no processor is in the process of choosing a value. In the protocols given so far, the variables correspond to balancers, and the latency corresponds to the network depth.

A *path* is a sequence of variables. In any protocol state, processor p has *preferred path* u if p would traverse u if it were run in isolation until choosing a value. If p would choose value v, then v is its *preferred value.* Define the *capacity* c of the protocol to be the maximal number of processes that access any particular variable in any execution. If c is high, so is the maximum number of concurrent accesses to a variable, so Capacity is a measure of potential contention.

Consider a linearizable counting protocol for n processors with capacity c.

Lemma 5.3 *In any quiescent state, the preferred path for any token p must traverse at least $\lceil (n-1)/(c-1) \rceil$ variables.*

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

1. q is about to choose value k.

2. q is about to access a variable in p's preferred path.

We claim the first case cannot occur. Since the protocol is in a quiescent state, all values less than i have been taken, and therefore any processor that starts the protocol and runs uninterruptedly must choose i. If p and q can both run to completion without accessing a common variable, they will both choose i, a contradiction. Therefore q's path must eventually intersect p's preferred path.

By hypothesis, no more than $c-1$ processors can access any variable along p's path. Since every process's path must intersect p's path somewhere, the path must include $\lceil (n-1)/(c-1) \rceil$ distinct variables. ∎

Theorem 5.4 *Any linearizable counting protocol for n processes and capacity c has latency $\Omega(n/c)$.*

Proof: It is enough to show that in any sequential execution, every processor traverses at least $\lceil (n-1)/(c-1) \rceil$ variables. Initially, the protocol is quiescent, and Lemma 5.3 implies that the first processor traverses at least $\lceil n/c \rceil - 1$ variables. After each processor chooses a value, the protocol returns to a quiescent state, and the same argument applies. ∎

If we define a low-contention algorithm to be one where c is constant, then any low-contention linearizable counting protocol has linear latency.

This theorem has further implications for counting networks. Elsewhere, [4] we have shown that the set of balancers traversed by a set of tokens in a counting network does not depend on how transitions are interleaved, which implies:

Corollary 5.5 *In any execution of a counting network, the average number of balancers traversed by every token is* $\Omega(n/c)$.

6 Conclusion

The following joke circulated in Italy during the 1920's and 30's.

> Mussolini claims that the ideal citizen is intelligent, honest, and Fascist. Unfortunately, no one is perfect, which explains why everyone one meets is either intelligent and Fascist but not honest, honest and Fascist but not intelligent, or honest and intelligent but not Fascist.

The ideal linearizable counting algorithm has low contention, low latency, and does not require waiting. Unfortunately, Theorem 5.4 shows that no ideal algorithms exist. The best algorithms one can devise either have low latency and no waiting but high contention (like the single shared variable), low contention and low latency but require waiting (like the WAITING-FILTER), or low contention and no waiting but high latency (like the SKEW and REVERSE-SKEW filters).

7 Acknowledgments

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. In *Proceedings of the 3^{rd} Symposium on Discrete Algorithms,* Orlando, Florida, to appear, January 1992. Also: Technical Report 679, The Technion, June 1991.

[2] A. Aggarwal 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] M.P. Herlihy, N. Shavit, and O. Waarts. Linearizable Counting Networks. In *Proceedings of the 32^{nd} Annual Symposium on Foundations of Computer Science,* San Juan, Puerto Rico, October 1991, pp. 526-535.

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

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

[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 *Proceedings of the 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] G. Graunke and S. Thakkar, Synchronization algorithms for shared-memory multiprocessors., *IEEE Computer*, 23(6):60–70, June 1980.

[13] 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.

[14] 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.

[15] M. Klugerman and C. Greg Plaxton, Small-Depth Counting Networks. In preparation, MIT-LCS/UT at Austin, October 1991.

[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] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Sixth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1987, pp. 137–151. Full version available as MIT Technical Report MIT/LCS/TR–387.

[20] 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.

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

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

[23] 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.

[24] L. Rudolph, Decentralized cache scheme for an MIMD parallel processor. In *11th Annual Computing Architecture Conference,* 1983, pp. 340-347.

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

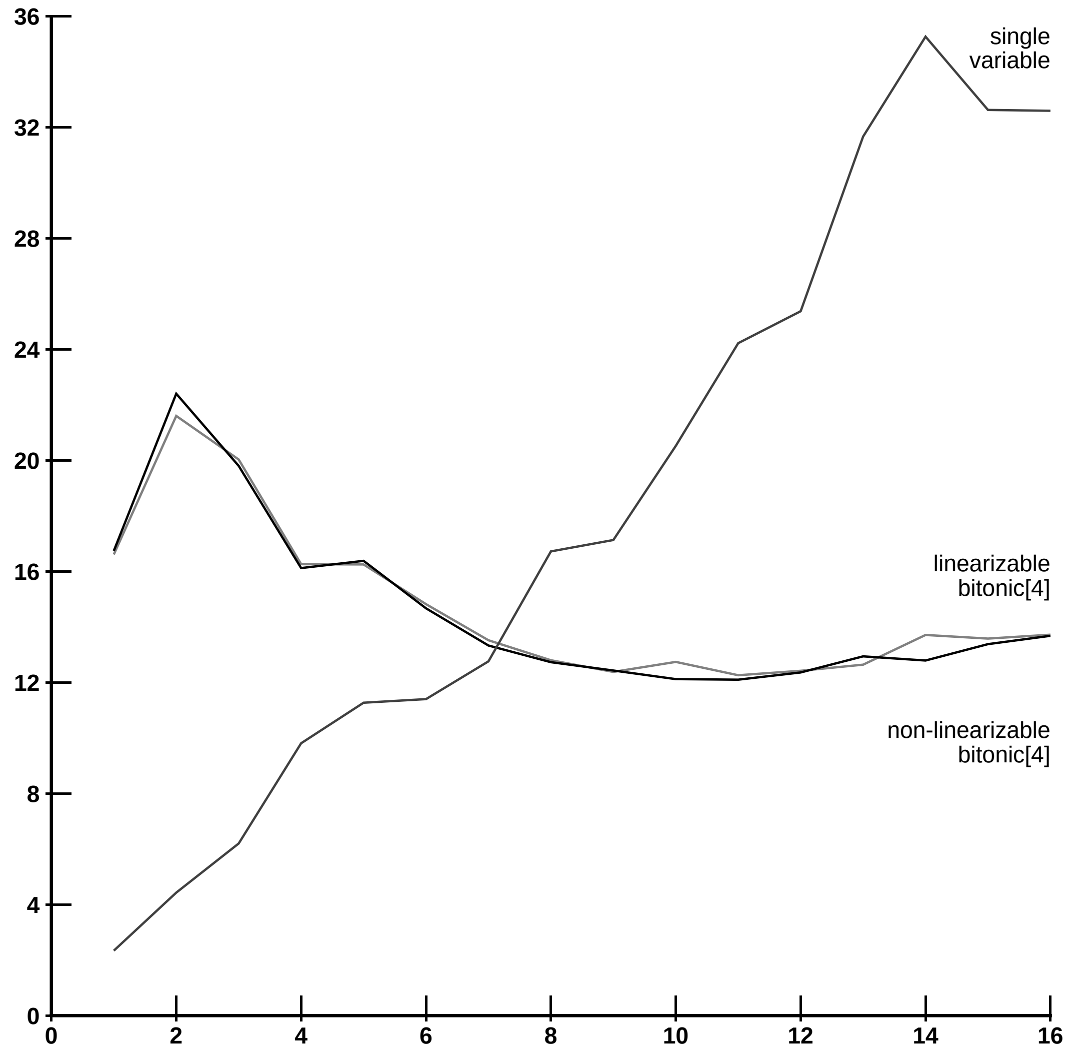

Figure 4: Shared Counter Implementations

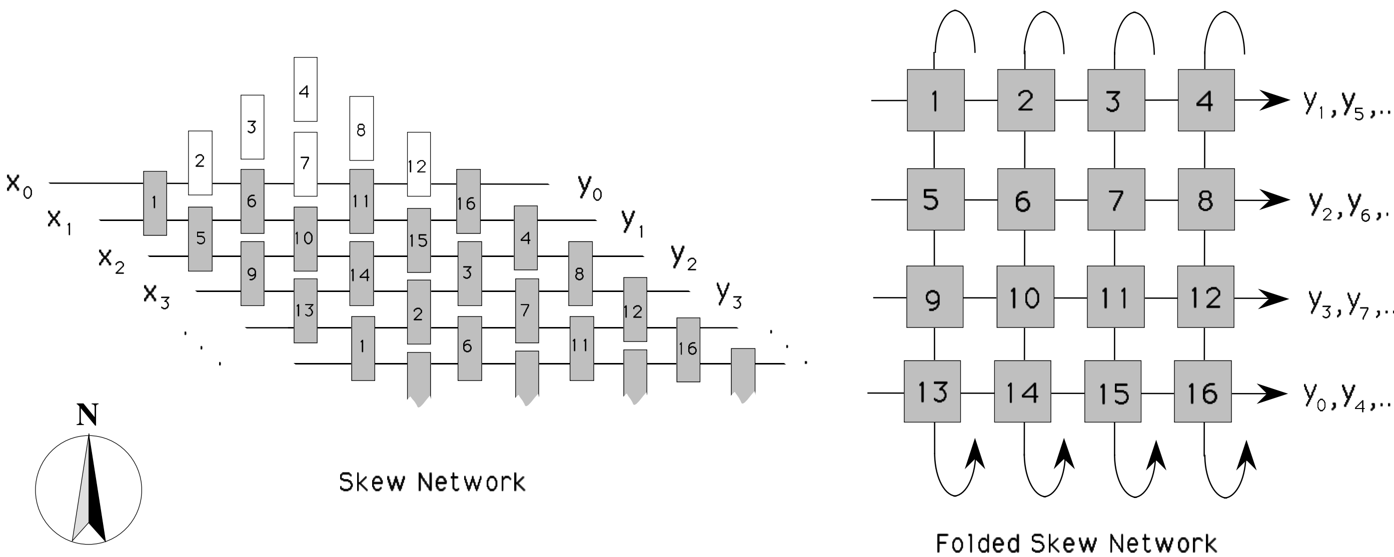

Figure 5: Skew Network and Folding