CHALMERS | GÖTEBORG UNIVERSITY

Technical Report No. 2009:12

Functional and dynamic programming in the design of parallel prefix networks

MARY SHEERAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY/
UNIVERSITY OF GOTHENBURG

Functional and dynamic programming in the design of parallel prefix networks

Mary Sheeran
ms@chalmers.se

December 9, 2009

Abstract

A parallel prefix network computes each $x_0 \circ x_1 \circ \ldots x_k$ for $0 \leq k < n$, for an associative operator $\circ$. This is one of the fundamental problems in computer science, because it gives insight into how parallel computation can be used to solve an apparently sequential problem. As parallel programming becomes the dominant programming paradigm, parallel prefix or scan is proving to be a very important building block of parallel algorithms and applications. There are a great many different parallel prefix networks, with different properties such as number of operators, depth and allowed fanout from the operators. In this paper, ideas from functional programming are combined with search to enable a deep exploration of parallel prefix network design. Networks that improve on the best known previous results are generated. It is argued that precise modelling in Haskell, together with simple visualization of the networks, gives a new, more experimental, approach to parallel prefix network design, improving on the manual techniques typically employed in the literature. The programming idiom that marries search with higher order functions may well have wider application than the network generation described here.

1 Introduction

The *all-prefix-sums* operation calculates the sums of the prefixes of a sequence, where the *sum* operation can be any associative (but not necessarily commutative) operator [Blelloch, 1990]. Parallel implementations of *all-prefix-sums* are usually called *parallel prefix* or scan, emphasizing that the operator can be varied. Parallel prefix is one of the fundamental algorithms of computer science, and it has been much studied. Blelloch [1990] lists string comparison, polynomial evaluation, the solution of tri-diagonal linear systems, lexical analysis and many other uses of parallel prefix. Parallel prefix is interesting because it permits parallel implementation of what initially appears to be a sequential problem. Parallel prefix *networks* comprise an arrangement of operators that is independent of the values of the inputs, and so can be directly implemented as a circuit. Such networks are one of the most important building blocks in modern microprocessors, for example implementing priority encoders and computing the carries in fast adders. With recent renewed interest in data-parallel programming, parallel prefix or scan is again an important building block, for example as a key library function used in programming graphics and other algorithms on graphics processors (GPUs).

Yet many questions remain unanswered. As we shall see later, there are two main classes of parallel prefix networks: the so-called Depth Size Optimal (DSO) networks, where we have good results about optimality, and the shallower but larger networks that we must resort to when DSO networks do not exist. These shallow networks are much less well understood. Even designing DSO networks is still typically a painstaking hand-craft, tackled for particular situations such as allowing the output of an operator to be fanned out exactly two or four times. For small shallow networks, there has been little real progress since the 1980s.

This paper shows how simple ideas from functional programming can help to make the process of designing parallel prefix networks more systematic. For DSO networks, we are able to

make the allowed fanout of the operators a parameter, and to systematically generate DSO networks for a given width and depth. The key idea is to search for networks with a particular recursive decomposition within a given context. The same idea, slightly generalized, allows us to construct small shallow networks even where DSO networks don't exist, and still maintaining control over fanout. The resulting networks improve on the best known networks. Finally, our experiments with using search led us to a new construction of minimum depth networks that is again hard-wired and no longer uses search, further advancing the state of the art. One of our aims is to encourage readers to try to make further improvements, or indeed to fill in the gaps in the theory that would tell us how much further we can push the limits. Because the prefix networks discussed are precisely described in Haskell, the paper also functions as a tutorial on prefix networks. Files containing the Haskell code discussed in the paper are available at URL **http://www.cse.chalmers.se/∼ms/TR0912/**.

We regard the paper as an interesting application of functional programming. For some readers, it might even serve as an introduction to functional programming and its use in problem solving. In particular, we want to emphasize the benefits that modelling ones problem in a functional language can bring, especially when combined with some simple visualization. We do, however, assume some basic knowledge of the functional programming language Haskell [haskell.org, 2009]. The combination of higher order functions used to capture network structure and search is, we believe, a powerful programming idiom that may have application beyond the kind of network generation described here.

2 The prefix problem

The *prefix problem* is to compute each $x_0 \circ x_1 \circ \ldots x_k$ for $0 \leq k < n$, for an associative operator $\circ$. The most obvious solution is to connect a series of $n-1$ operators in a linear array, as shown schematically in Figure 1. Input nodes are on the top of the network, with the least significant input (x_0) being on the left. Data flows from top to bottom, and we also count the stages or levels of the network in this direction, starting with level zero on the top. An operation node, represented by a small circle, performs the $\circ$ operation on its two inputs. It may also produce an output along a diagonal line below and to the right of the node. The depth of a network is the maximum number of operators on a path from input to output. In real circuits implementing these networks, this is related to delay, where the time unit is the the time taken to perform one operation. So one can think of the levels in the diagrams as time steps. The operator node at the top left of the serial network takes two inputs at time zero, and produces an output at time one. It is placed in the diagram at level one, the same level at which it produces its output.

The fanout of a node is its out-degree, and of a network is the maximum of the fanouts of its nodes. In this example, all but the rightmost node have fanout 2, so the whole network is said to have fanout 2. The *size* of a network is the number of binary operators that it includes. We will typically use w or n for width, s for size, d for depth and f for fanout. To ease presentation, we write $j : k$ for $x_j \circ x_{j+1} \circ \ldots \circ x_k$.

Examining the serial prefix structure in Figure 1, we see that at each non-zero level only one of the vertical lines contains an operator. *Parallel* prefix networks can have more than one operator at a given level, so that there is some parallelism in the resulting computation. Parallel prefix structures have been much studied because they shed light on the theory of parallelism, and on the complexity of computation by networks [Pippenger, 1987]. For an introduction to parallel prefix (or scan) that will appeal to functional programmers, the reader is referred to [Hinze, 2004]. Blelloch's paper is also an excellent tutorial introduction to the topic [Blelloch, 1990].

Because the last output of a prefix network of width n must combine each of the n inputs using a binary operator, the minimum possible depth is $\lceil log_2 n \rceil$. This bound is achieved by a standard divide and conquer approach, usually attributed to Sklansky [1960], and illustrated in Figure 2.

Another very well known network, shown in Figure 3, is due to Brent and Kung [1982], who were among the first to advocate the use of prefix networks to calculate the carries in fast adders. This network is deeper than the Sklansky construction, but it has a fanout of only 2, and considerably

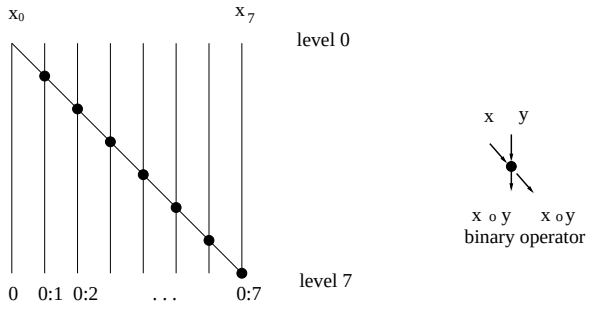


Figure 1: The serial prefix structure for 8 inputs. Here, at level zero, there is a diagonal line leaving a vertical wire in the absence of a node. This is a fork, and the outputs of the following nodes are also forked. We say that this network has *width* 8. The network shown contains 7 nodes, and so is said to be of *size* 7. Its lowest level in the picture (and that with the highest number) is level 7, so the network has *depth* 7.

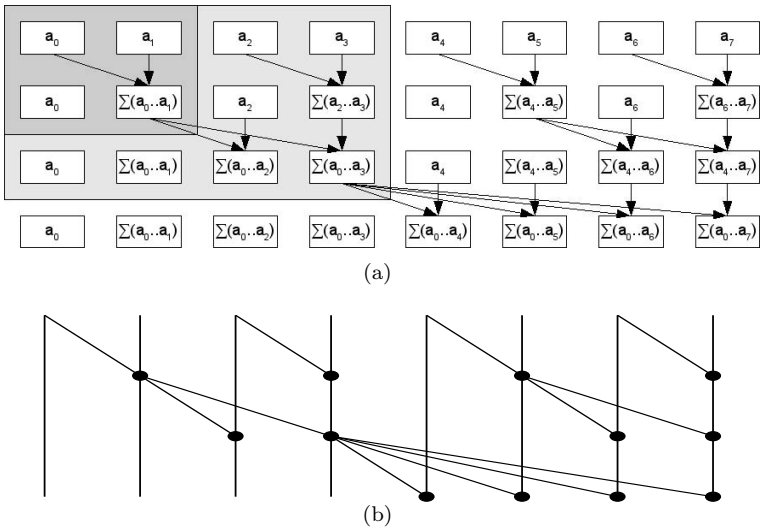


Figure 2: The Sklansky construction for 8 inputs. (a) shows how the data flows through the network. A box with two incoming arrows is an operator. (b) shows how the same network is drawn in the style that is standard for prefix networks. The construction recursively computes the parallel prefix for each half of the inputs and then combines the last output of the lower (left) half with each of the outputs of the upper (right) half. The result, for 8 inputs, is that there are 4 operators on each of four levels.

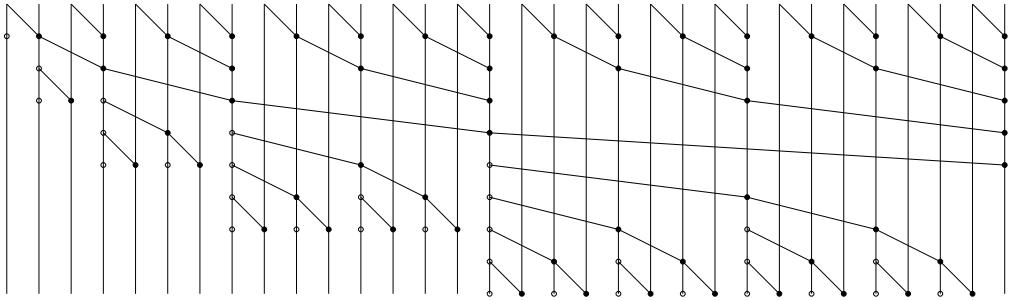


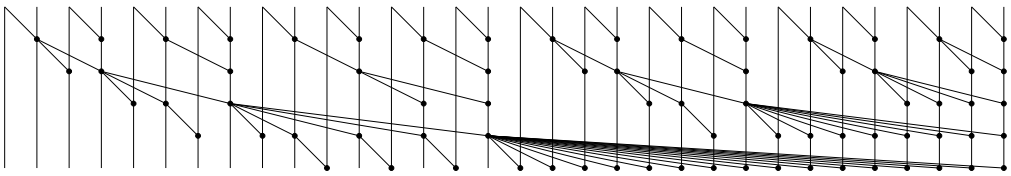
Figure 3: The Brent Kung construction, fanout 2, 32 inputs, depth 9, 57 operators. Note that the network has a wire in the middle that first (reading from the top) has 4 2-input operators (black circles) followed by 5 buffers (white circles). Without the first four of those buffers, the last of the 2-input operators would have fanout 6, rather than the intended 2.

fewer operators. In order to maintain a fanout of two at each level, this construction uses so-called buffers, which are shown in the diagram as white circles, and are one-input, one-output prefix networks that simply pass the input through unchanged. Think of a buffer as indicating that one should step down one level (one time unit) in the diagram, without adding any binary operators. In VLSI circuits, a large fanout increases the load on the component driving those wires, and is generally to be avoided. So controlling fanout is important, and buffer circuits, whose logical function is the identity are actually used. Aiming for a small number of operators is also important, as power consumption is closely related to the number of components in a circuit.

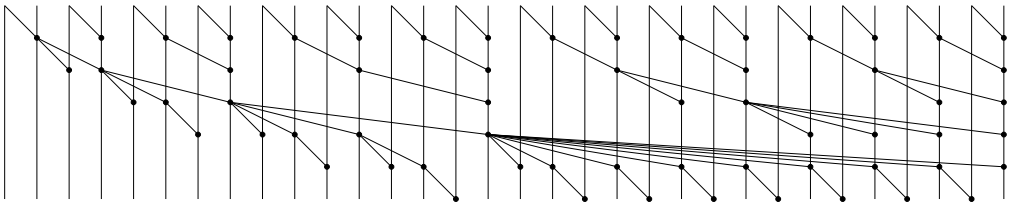
Finally, we mention the Ladner Fischer construction, which is a subtle (and often misunderstood) combination of Brent Kung- and Sklansky-like patterns [Ladner & Fischer, 1980], see Figure 4.

We will return to all of these networks to consider them in more detail when we develop Haskell descriptions of them in the following sections. Although we have shown some of the most well-known prefix networks, we have not covered the entire design space. The Kogge Stone network is perhaps the most prominent of those that we do not cover in detail [Kogge & Stone, 1973]. It is shown in Figure 5. In practice, the Kogge Stone construction tends to give very fast circuits that are large and consume a lot of power. The high power consumption is related to the large number of operators in the network. An important research direction has been the exploration of ways to modify the construction to reduce the area and power consumption without sacrificing too much speed, see for example [Han & Carlson, 1987]. Knowles [1999] explores a family of networks that have Sklansky (with high fanout) at one extreme, and Kogge Stone (with fanout 2) at the other.

Our motivations are similar. We want to achieve low depth, low fanout and small number of operators simultaneously. This is a far from trivial task. It involves exploring a fresh area of the design space, and reveals some gaps in existing theoretical knowledge about optimality in prefix networks. We reject the Kogge Stone construction as a basis because it is so expensive in number of operators. Instead, we concentrate on ways to generalize the Brent Kung and Ladner Fischer constructions. The following section provides the necessary background by showing how to describe the Brent Kung and Sklansky networks in Haskell. Next, we generalize Brent Kung construction, and show how dynamic programming can be used to find good networks. A lazy functional programming language proves to be the ideal setting for this approach. To generate small shallow networks, we must generalize further; we introduce the Ladner Fischer construction and show how it too can effectively be combined with search. The insights gained in searching for networks in this way led finally to a new generalisation of Ladner Fischer that improves markedly upon the original algorithm, and is even an improvement on the best known solution found in the literature. We would argue that the functional style of algorithm description presented here



(a)



(b)

Figure 4: (a) LF_0, the minimum depth Ladner Fischer network, for 32 inputs and (b) LF_1, which is one level deeper (depth 6) but contains fewer operators (62 vs. 74).

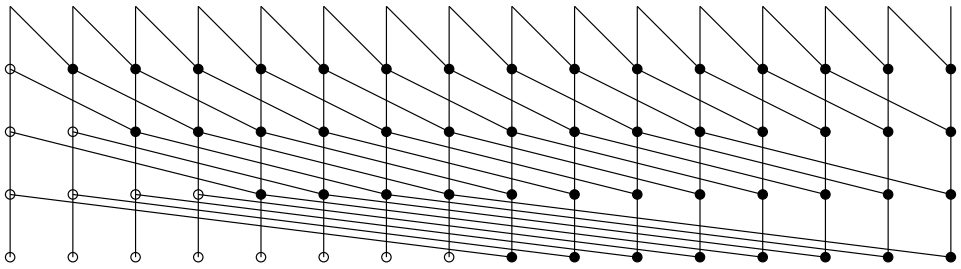


Figure 5: The basic Kogge Stone construction, with 16 inputs, depth 4 and fanout 2. For 16 inputs, it has 49 operators, compared to 31 for the minimum depth Ladner Fischer network.

provides a new tool for algorithm design and experimentation. We hope, therefore, that this paper will be read not only by functional programmers, but also by researchers in algorithms who might be willing to explore the use of functional programming in their research. For this reason, we have tried to keep to a simple style of functional programming.

3 Describing standard prefix networks in Haskell

3.1 The serial network and analyzing descriptions by non-standard interpretation

A prefix network takes a list of n inputs, and returns a list of the same length. We will concentrate on the patterns (or higher order functions) used to construct such networks from smaller prefix networks. Our first basic building block will be a two-input, two-output prefix network, as shown on the right of figure 1. Thus, the kind of pattern that we wish to study has type

```
type NW a = ([a] -> [a]) -> [a] -> [a]
```

It takes a function from list of `a` to list of `a`, the building block, and returns a network of the same type – the prefix network constructed from that building block. The function `ser` captures the serial connection pattern from Figure 1. Here, the `comp` parameter is a two-input two-output building block. The definition of `ser` consists of two obvious base cases and a step. The step case includes one use of `comp` and a recursive call of `ser`.

```
ser :: NW a
ser _    []       = []
ser _    [a]      = [a]
ser comp (a:b:bs) = a1:cs
  where
    [a1,a2] = comp [a,b]
    cs      = ser comp (a2:bs)
```

An example of a suitable building block would be the two input prefix sum, the function `pplus`. Then, `ser pplus` computes the prefix sum of its inputs:

```
pplus[a,b] = [a,a+b]

*Main> ser pplus [1..8]
[1,3,6,10,15,21,28,36]
```

To analyse the generated networks, we typically use non-standard interpretation (NSI), see for instance [Singh, 1992]. In NSI, we simply replace the building block by one designed to give the required information when the resulting network is run. For instance, to calculate the delay through the network, we would model the two-input two-output component as

```
delF :: [Int] -> [Int]
delF [a,b] = [m,m+1]
  where m = max a b
```

giving the means, for each output, to count how many components are on the longest path from the inputs to that output. Then, running the resulting network on a list of zeros (indicating delay-in) gives us delay information for each of the outputs:

```
*Main> ser delF (replicate 8 0)
[0,1,2,3,4,5,6,7]
```

This corresponds to what we would expect by examining Figure 1. This simple analysis works because we restrict our attention to networks that can be represented in the kinds of diagrams that we have already used to depict prefix networks. Our interest is in these circuit-like *data independent* networks, and we do not consider more general data-dependent algorithms.

As well as estimating delays using NSI, we also use it to gather the information about the networks that is used to generate the prefix network diagrams themselves, see Appendix A.

3.2 The Sklansky network

To describe the Sklansky divide and conquer algorithm, we first define the fanout structure that is used to combine the last output of the left-hand network with each of the outputs of the right-hand one:

```
fan :: NW a
fan _ [a]  = [a]
fan comp (a:as) = bs++[c]
  where
    [b1,c] = comp [a, last as]
    bs     = fan comp (b1:init as)

*Main> fan pplus (4:[5..8])
[4,9,10,11,12]
```

Note that `fan comp` is not itself a prefix network (although it has the same type as our network patterns). Using `fan`, the definition of the Sklansky network is straight-forward:

```
skl :: NW a
skl _ [a]   = [a]
skl comp as = init los ++ ros'
  where
    (los,ros) = (skl comp las, skl comp ras)
    ros'      = fan comp (last los : ros)
    (las,ras) = splitAt ((length as + 1) `div` 2) as
```

The two recursive calls appear explicitly, each working on roughly half of the inputs. Again, we can run the resulting network to convince ourselves that we have got it right.

```
*Main> skl pplus [1..4]
[1,3,6,10]

*Main> skl pplus [5..8]
[5,11,18,26]

*Main> fan pplus (10:[5,11,18,26])
[10,15,21,28,36]

*Main> skl pplus [1..8]
[1,3,6,10,15,21,28,36]

*Main> skl delF (replicate 8 0)
[0,1,2,2,3,3,3,3]
```

As expected, the final 4 outputs are produced after 3 units of time.

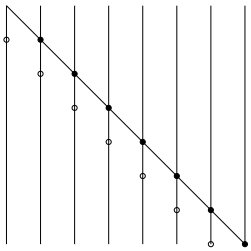


Figure 6: Adding buffers to the serial network using the function `bser`. If we now compose some fanouts in series with this network, those fans will only start at the buffers, and will not increase the fanouts at the nodes.

3.3 Describing the buffers that are used to control fanout

So far, in the above Haskell descriptions, we have assumed that the building block, `comp`, acts on a 2-list of inputs, to produce a 2-list of outputs. To describe buffers, we simply extend it to also act on singleton lists (and it is this need to deal with buffers that led to the choice of lists rather than tuples in the first place). This approach also opens the possibility of using larger prefix networks as building blocks, which makes sense, for instance, when designing parallel prefix networks for implementation on silicon. However, we will not consider such higher radix networks here.

Typical patterns of buffer use are to place a buffer only on the first element of a list, or on all the elements of a list:

```
buf1 :: NW a
buf1 comp (a:as) = comp [a] ++ as

bufall :: NW a
bufall _ [] = []
bufall comp (a:as) = buf1 comp (a:bufall comp as)
```

We will later make repeated use of "buffered fan" and "buffered serial" patterns. The latter composes a serial network with the application of buffers to all but its last output, see Figure 6. The function `bser` could also be defined directly as a serial network of buffered components. However, the combinator for serial composition, written `->-`, and the combinator-based style of programming that its use encourages will both prove useful later.

```
bfan :: NW a
bfan _ [a]   = [a]
bfan comp as = buf1 comp (fan comp as)

toInit :: NW a
toInit f as = f (init as) ++ [last as]

compose :: (a -> b) -> (b -> c) -> a -> c
compose circ1 circ2 = circ2 . circ1
circ1 ->- circ2     = compose circ1 circ2

bser :: NW a
bser comp = ser comp ->- toInit (bufall comp)
```

This approach to describing buffers demands that any building block, such as `pplus` used earlier, must be extended with a case that makes it behave as the identity on singleton lists. The timing behaviour of the buffer is modelled by adding a new case to `delF`:

```
delF :: [Int] -> [Int]
delF [a]   = [a+1]
delF [a,b] = [m,m+1]
  where m = max a b

*Main> fan delF (replicate 8 0)
[0,1,1,1,1,1,1,1]
*Main> bfan delF (replicate 8 0)
[1,1,1,1,1,1,1,1]
```

3.4 Checking correctness

To check functional correctness, we make use of parametricity, feeding singleton lists `[0]`, `[1]`, `[2]` and so on into a network in which the operators (both unary and binary) have been replaced by the function that appends two lists. The result should then be `[0]`, `[0,1]`, `[0,1,2]`, and so on. This idea is encoded in the following function:

```
check func m = func app [[a]| a <- l] == tail (inits l)
  where l = [0..m-1]
        app [a]   = [a]
        app [a,b] = [a,a++b]

*Main> tail (inits [0..7])
[[0],[0,1],[0,1,2],[0,1,2,3],[0,1,2,3,4],[0,1,2,3,4,5],
 [0,1,2,3,4,5,6],[0,1,2,3,4,5,6,7]]

*Main> check bser 12
True

*Main> check skl 33
True
```

For further exploration of parametricity in the context of prefix networks, see [Voigtländer, 2008], which was partly inspired by an earlier unpublished version of this paper.

3.5 The Brent Kung network and variations upon it

Consider again the network shown in Figure 3. Brent and Kung [1982] describe their network as a binary tree producing the last output, and an "inverted tree" that produces all the remaining outputs. We prefer the recursive decomposition that is also frequently used in the literature, and which we illustrate in Figure 7. The network can be viewed as having three phases: a first in which adjacent inputs are combined using small 2-input prefix networks; a second in which the last outputs of those small prefix networks are passed to a smaller Brent Kung network; a third in which the final result is slightly adjusted, again by combining adjacent elements. In designing a higher order function to capture this pattern, it is important to be sufficiently general to enable later reuse. Such a pattern is shown in Figure 8, and is captured in the function `build`. The widths of the small networks across the top and bottom are allowed to vary, and are captured by a list of integers specifying the partition – the `ss` parameter to `build`. In addition, the networks across the top and bottom are allowed to be possibly different from each other, and are specified by the parameters `ts` and `bs` to `build`.

```
parL :: [a -> b] -> [a] -> [b]
parL _      []     = []
parL (f:fs) (a:as) = f a : parL fs as
```

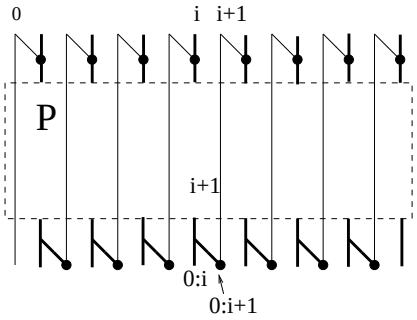


Figure 7: The recursive decomposition used in the Brent Kung network. The inputs to the recursive call (the dotted box marked P) are $(0:1),(2:3),\ldots,(i-1:i),(i+1:i+2),\ldots$, and the corresponding outputs are $(0:1),(0:3),\ldots(0:i),(0:i+2),\ldots$. The even numbered inputs, are interleaved with these values, to give $0,(0:1),2,(0:3),\ldots,(0:i),i+1,(0:i+2),\ldots$. This allows the final (bottom) row of operators to combine adjacent values (such as $(0:i)$ and $i+1$), to produce the correct result at each output. The Brent Kung paper is not explicit about how the outputs should be computed for input width not a power of two. In our formulation, for an odd number of inputs, the rightmost wire and all attached operators would be dropped.

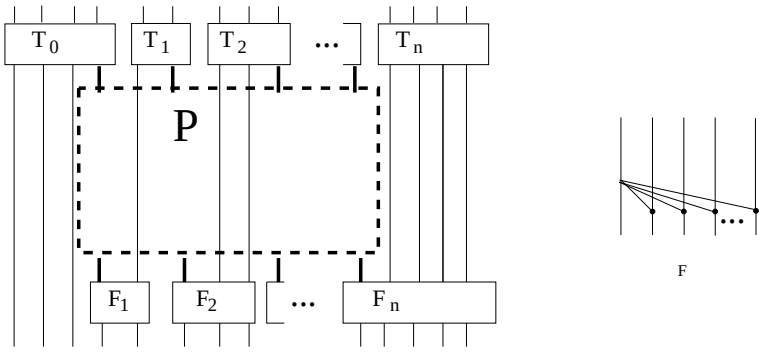


Figure 8: Generalizing the Brent Kung pattern. The widths of the small prefix networks marked T across the top can vary, and are no longer restricted to be 2. All but the leftmost prefix network (T_0) are matched by corresponding *fan* networks across the bottom (marked F). If the partition on the top is $[a_0, a_1, \ldots, a_n]$, then that across the bottom is $[a_0 - 1, a_1, \ldots, a_n + 1]$. Each T_i has width a_i. Each F_i has width a_i for $0 < i < n$. F_n has width $a_n + 1$.

```
split :: [Int] -> [a] -> [[a]]
split []     [] = []
split (d:ds) as = let (las,ras) = splitAt d as in las : split ds ras

build :: [Int] -> [NW a] -> [NW a] -> NW a -> NW a
build ss ts bs p comp = split ss ->- parL [t comp | t <- ts]  ->-
                        toInit (toLasts (p comp)) ->- concat ->-
                        split (shift ss) ->- parL [b comp | b <- bs] ->- concat
  where shift (a:as) = a-1 : init as ++ [last as + 1]
```

The three distinct phases correspond to the three lines in the definition of `build`, and are composed using serial composition. In the first phase, using the function `split`, the input is split according to the parameter `ss`, and the resulting sub-lists are operated on by the elements in the list of patterns `ts`. If `ts` is longer than the length of `split ss`, the superfluous elements will be ignored (see the base case of `parL`). In the second phase, `(p comp)` is applied to the last elements of the outputs of all but the last of those `ts` (corresponding to the n last outputs of T_0 to T_{n-1} in the diagram). All other "wires" are passed through unchanged, and the result is then concatenated back to a single list. (The function `toLasts` is defined in Appendix B.) In the third phase, `split` is once again used to divide up the inputs to that phase, but this time slightly differently. If the partition in the first phase is $[a_0, a_1, \ldots, a_n]$, then that in the third is $[a_0 - 1, a_1, \ldots, a_n + 1]$. This is what is captured by the function `shift`. When the division is made, the resulting list is operated on, pointwise, by the list of patterns `bs`. In all cases, `bs` will have the identity or buffers as its first element and a list of fans as the rest of the list, matching Figure 8. Think of the recursive call as being the filling of a (British) sandwich with small prefix networks on one side and fans on the other. This recursive decomposition is inspired by (but not exactly the same as) one by Snir [1986].

The use of the partitions parameter in `build` differs from the style of combinators that we have used earlier in Lava [Bjesse *et al.* , 1998], where we tend to avoid explicit size parameters. However, the inclusion of this parameter will permit our later use of search to find good prefix networks. Hinze also used partitions in his study of combinators for building scans [Hinze, 2004]. The function `build` is similar to, but slightly more general than Hinze's generalization of Brent Kung (p. 16 of [Hinze, 2004]). We will return to this topic in section 6. If the `ts` parameter consists of a list of prefix networks, `bs` is a list of fans and `p`, the sandwiched network, is also a prefix network, then `build ss ts bs p` produces a prefix network. We have chosen `build` as our preferred combinator because it is sufficiently general to cover *all* of the prefix networks described in the remainder of the paper. One could say that the rest of the paper is about investigating possible parameters to `build`, and thus exploring the design space of prefix networks.

Now, the following choice of parameters to `build` gives the standard "steps of two" Brent Kung construction (and `bK0` was the definition used to produce both Figures 3 and 9):

```
bfans = repeat bfan
sers  = repeat ser

twos n = replicate (n `div` 2) 2 ++ [n `mod` 2]

bK0 _ [a] = [a]
bK0 op as = build (twos (length as)) sers (bufall:bfans) bK0 op as
```

We place serial networks across the top, and buffered fans across the bottom, apart from the leftmost block, which is simply buffered. Thus, the list from which the top network patterns are drawn is `sers` and that for the bottom is `bufall:bfans`. The prefix network sandwiched between these two layers is a recursive call of the function `bK0` itself. For an even number of inputs, the arrangement of small serial networks along the top is given by the list `[2,2, .. ,2,0]`, which means that the corresponding list for the pattern along the bottom is `[1,2,2,..,2,1]`. When the last element of the top partition is zero, the rightmost fan at the bottom of the network will

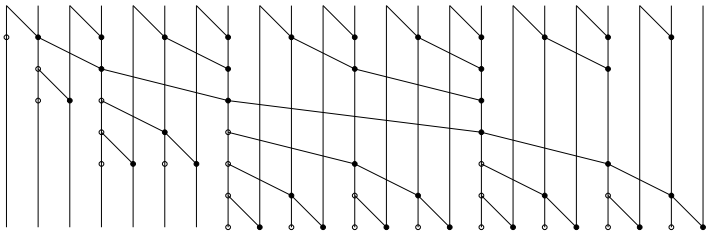


Figure 9: The Brent Kung-like pattern for 23 inputs, produced from the function `bK0`.

have width one, see Figure 8. This typically means that that output will be produced earlier than the outputs just to its left. (Note that we now allow some elements of the top partition to be zero. This means that those functions that can be mapped across partitions, and in particular `ser`, must have a cases for when the input is an empty list.)

For an odd number of inputs, there are a number of possible choices. We have chosen that the two sequences at top and bottom should be `[2,2,..2,1]` and `[1,2,..,2,2]`. When the input width is 2^k for some k, these choices result in a network whose last output is produced at depth k, while the overall depth is $2k-1$, see for example Figure 3. Such networks that produce their last outputs at minimum depth are called *restricted* in the literature [Fich, 1983]. It seems clear from Brent and Kung [1982] that they intended to produce restricted networks for all input widths. We have made a slightly different choice, resulting, for example, in the attractively symmetrical network shown in Figure 9.

3.6 The notions of depth- and waist-size optimality

The 23-input network shown in Figure 9 has depth 7 and size 37. (We do not count the buffers, but only the 2-input operators.) Snir has shown that for a prefix network of width n, depth d and size s, it is the case that $d + s \geq 2n - 2$ [Snir, 1986]. Thus, a network (like this one) that obeys $d + s = 2n - 2$ has reached that lower bound and is called *depth-size optimal* (DSO). For the given depth and width, there is no smaller network. Snir's work showed that the lower bound can be reached for depths ranging from $2log_2n - 2$ to $n - 1$. The serial construction produces DSO networks; for an n-input network, both depth and size are $n - 1$.

Following Lin *et al* [2003], we have introduced the related notion of *waist-size optimality* [Sheeran & Parberry, 2006]. The *waist* of a network is the difference in levels between the first fork on the leftmost input and the production of the rightmost output. For the network in Figure 9, the waist is 7, the same as the depth, while the 32-input Brent Kung network shown in Figure 3 has waist 5, but depth 9. A network with waist w, width n and size s is *waist-size optimal* (WSO) if $w + s = 2n - 2$ [Lin *et al.* , 2003].

The Brent Kung-like networks defined with `bk0` and `twos` are WSO. For some input widths, for example 23, the waist and depth are equal, and then they are also DSO.

How would we go about building Brent Kung-like networks that are DSO for all input widths? We need to keep the waist and depth equal, by making sure that the last element of the top sequence is 1 (and not zero). This means that our earlier function `twos` (used in the definition of `bK0`) could, for instance, be replaced by

```
twos' 1 = [1]
twos' 2 = [1,1]
twos' n = 2 : twos' (n-2)

bK1 _ [a] = [a]
bK1 op as = build (twos' (length as)) sers (bufall:bfans) bK1 op as
```

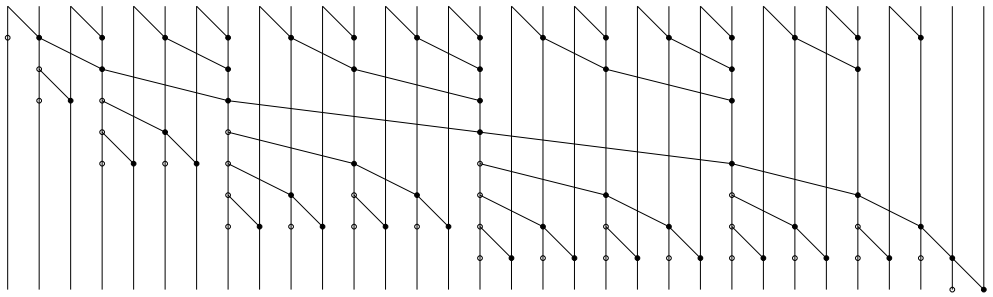


Figure 10: A Brent Kung-like pattern, generated using the sequences given by `twos'`. Networks generated in this way are always both WSO and DSO. Since the size of the waist has increased, but the network is still WSO, this 32 input, depth 9 network is smaller than the standard Brent Kung construction in Figure 3, 53 vs. 57 operators.

```
*Main> check bK1 59
True
*Main> check bK1 128
True
```

Then, even-width inputs are now divided as `[2,2,..2,1,1]`, while the division of odd-width inputs is unchanged. The result, for 32 inputs, is shown in Figure 10. This construction is due to Lin and Liu [1999]. For even-width inputs, Lin and Liu opted to place two ones at the end of the sequence of widths: `[2,2,..2,1,1]`. The last of those ones must be at the end of the sequence, but what about other placements for the other one? The reader might like to consider other possible solutions. A recent paper, also by Lin, studies this problem, comparing the Lin and Liu construction to two others [Lin & Hung, 2009]. We will move on to the largely open question of how to deal with fanout greater than two, both in the production of DSO networks and of larger but shallower networks.

4 Generating Depth Size Optimal (DSO) prefix networks

Our overall task is this: **for a given depth and width, find a prefix network with few operators and low fanout**. In this section, we explore the generation of DSO networks, while retaining control of fanout. In the following section, we attack the harder question of how to generate small shallow networks when DSO networks do not exist.

To generate DSO networks, we use the pattern shown in Figure 8, that we have encoded in the function `build`; but we use *search* to find appropriate partitions, rather than designing them a priori. To do this, we must introduce the notion of a *context* into which a prefix network should fit, and then we search for the best network (according to some measure) that fits in that context.

A *context* for a width n prefix network is a pair containing an n-list of input delay values, and a single integer representing the desired maximum depth.

```
type Context = ([Int],Int)
```

To check if a proposed network fits in a context, we can run it with the input delays as input, and with `delF` as component, and compare the delays of the outputs with the required maximum output delay:

```
fits pat (is,o) = and [out <= o | out <- pat delF is]
```

Measuring how good a network is will be done in somewhat similar style. We first run the network to produce a list of elements of a `Net` datatype that encodes the position, phase and input "wires" of each component, see Appendix A. Each measure function takes such a list of `Net`s and produces a number. For example, the function `nops2 :: [Net] -> Int` counts the number of 2-input operators in a network. The trouble is that we now want to run our proposed network at two different types, and we run into problems because lambda binding is monomorphic. The solution is to pass around not the naked connection pattern, but the same pattern wrapped in a constructor:

```
type ANW = forall a. NW a

newtype WNW = Wrap ANW
unWrap (Wrap a) = a
```

This requires using the flag `-fglasgow-exts` in calls to GHC.

The function `try` checks whether or not a network `p` fits in a context, returning either `Nothing` or `Just (Wrap p)`:

```
try :: ANW -> Context -> Maybe WNW
try p (is,o) | fits p (is,o) = Just (Wrap p)
             | otherwise     = Nothing
```

Now, we have the programming building blocks for a search. We will search for Brent Kung-like networks. In section 3.5, we introduced network descriptions based around the top partition $[a_0, a_1, \ldots, a_n]$. Now, what we will do is explore the effect of varying that partition, moving away from the restricted forms that we have seen so far. In particular, we will explore the effect of allowing fanout to be greater than 2. Given a context, we will explore various partitions, each of which will in turn lead to a new context for the recursive call. Each partition will either succeed or fail in a given context.

What partitions should we explore for a given context `(is,o)`, bearing in mind that considering all integer partitions of the length of `is` will, in general, be too costly? First, it makes sense to restrict the fanout in the small fans across the bottom of the resulting network, and for this we will introduce an integer parameter `f`. The last element of the partition should then be at most `f-1`, so that the matching fan is at most `f`; elements other than this one and the first should be at most `f`. As we are particularly interested in shallow networks, we choose also to limit the first value of the partition to be a maximum of `f`. This would not be a good idea if we were interested in rather loose (that is deep) contexts, and the reader might like to consider why this is. Also, although we have earlier seen that it can be useful to have ones in partitions, we choose here to only permit a one in the last element of the top partition. This seems not to remove interesting networks, while restricting the search space considerably. These decisions lead to a first definition of the function generating top partitions for consideration:

```
tops0 :: Int -> Int -> [[Int]]
tops0 f n = [rs ++ [r] | r <- [1..f-1], rs <- perms 2 f (n-r)]

perms :: Int -> Int -> Int -> [[Int]]
perms _ _ 0   = [[]]
perms l f n   = [x:ts | x <- [l..f], x <= n, ts <- perms l f (n-x)]

Prelude Main> perms 2 3 10
[[2,2,2,2,2],[2,2,3,3],[2,3,2,3],[2,3,3,2],[3,2,2,3],[3,2,3,2],[3,3,2,2]]
```

```
Prelude Main> tops0 3 12
[[2,2,2,2,3,1],[2,2,2,3,2,1],[2,2,3,2,2,1],[2,3,2,2,2,1],[2,3,3,3,1],
 [3,2,2,2,2,1],[3,2,3,3,1],[3,3,2,3,1],[3,3,3,2,1],[2,2,2,2,2,2],[2,2,3,3,2],
 [2,3,2,3,2],[2,3,3,2,2],[3,2,2,3,2],[3,2,3,2,2],[3,3,2,2,2]]

Prelude Main> length (tops0 4 32)
109322
```

Next, we make a version of the `build` pattern that does the necessary wrapping and unwrapping, but otherwise has the now familiar serial networks across the top (`sers`) and buffers followed by fans across the bottom (`bufall:bfans`).

```
build0 :: [Int] -> WNW -> WNW
build0 ss p = Wrap $ build ss sers (bufall:bfans) (unWrap p)
```

How should we formulate the search? First, assume the existence of a function, `prefix` that takes a limit on fanout `f` and a context and returns either `Nothing` or a (wrapped) prefix network with maximum fanout `f` that fits in the context. We would call such a function as follows:

```
parpre0 :: Int -> Context -> ANW
parpre0 f ctx = maybe (error "no fit") unWrap (prefix f ctx)
```

In defining the function, the type and the two first cases are easy:

```
prefix :: Int -> Context -> Maybe WNW
prefix _ ([i],o) = try wire ([i],o)
prefix _ (is,o) | fits bser (is,o) = Just (Wrap bser)

wire :: ANW
wire _ as = as
```

If the context contains only a single input, then wiring the input to the output should work (and fit in the context that expresses constraints on the resulting delay). For wider networks, we will be return a serial network if it fits in the context, giving the second base case. We choose to use a *buffered* serial network here so that fans that are composed after it in a larger construction "meet" the buffers and so retain the expected fanout.

For the step, we would like, if possible to make a suitable network for each of the candidate partitions, that is each element of `tops0 f (length is)`. So we will define a function called `makeNet` that, given a partition, returns either `Nothing` or a (wrapped) network in which `prefix` has again been used to deal with the recursive call in the middle of the sandwich. Then, the third case for `prefix` is

```
prefix f (is,o) = listToMaybe $ mapMaybe makeNet (tops0 f (length is))
  where
    makeNet ds = ....
```

The combination of `listToMaybe` and `mapMaybe` chooses the first successful answer. (This will be sufficient for generating DSO networks, where all such networks are of the same size. We will return to the use of measure functions later.) Finally, `makeNet` is defined in the `where` clause as

```
        makeNet ds = do let js = map (last.(ser delF)) $ split ds is
                        q <- try wire ([last js],o-1)
                        p <- prefix f (init js,o-1)
                        return $ build0 ds p
```

This code calculates the context of the sandwiched recursive call of `prefix`, checking as well that the output of the rightmost serial network also meets its timing constraint (the call of `try wire`). The function `prefix` is called on the resulting context, and if a network is successfully returned, it

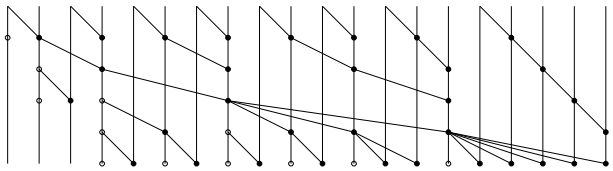


Figure 11: A DSO network found by the search embodied in the function `parpre0`.

is sandwiched (using `build0`) to produce the successful result for the partition being considered. If either of the calls to `try wire` or `prefix` fails, then the call of `makeNet` fails for the given partition.

This very simple approach works well for small examples. For example, drawing the result of calling the function `parpre0 f ctx` with its parameters defined by

```
f = 6
w = 5
n = 20
ds = replicate n 0
ctx = (ds,w)
```

gives the depth 5, fanout 6, width 20 network shown in Figure 11.

The way in which the `prefix` function builds up the final solution using solutions to smaller sub-problems is a form of *dynamic programming*. It is reminiscent of classic algorithms such as Dijkstra's shortest paths algorithm, which exploits the recursive structure of the problem – often called the *optimal substructure* in formulations of dynamic programming [Cormen *et al.* , 2001]. Typically, dynamic programming exploits the fact that the smaller sub-problems are *overlapping* in order to avoid repeated calculations. In the prefix network search, there are a great many overlapping sub-problems, and it makes sense to *memoise* the `prefix` function, and to move it inside the definition of `parpre0`, thus dropping its `f` parameter:

```
parpre1 :: Int -> Context -> ANW
parpre1 f ctx = maybe (error "no fit") unWrap (prefix ctx)
  where
    prefix = memo pm

    pm ([i],o) = try wire ([i],o)
    pm (is,o) | fits bser (is,o) = Just (Wrap bser)
    pm (is,o) = listToMaybe $ mapMaybe makeNet (tops1 f (length is))
      where
        makeNet ds = do let js = map (last.(ser delF)) $ split ds is
                        q <- try wire ([last js],o-1)
                        p <- prefix (init js,o-1)
                        return $ build0 ds p
```

The purely functional memo function used was provided by Koen Claessen; it is a refinement of Hinze's approach to the construction of memo functions [Hinze, 2000].

We also begin the process of adjusting the partition generation to consider fewer useless partitions. For shallow networks, a good move is to restrict attention to sequences starting with several 2s (as can be seen for example in the outer partition in figure 11). We have noted, simply by running a variety of searches, that for shallow networks all useful partitions are of this form, so in `parpre1` above we replace `tops0` by `tops1`.

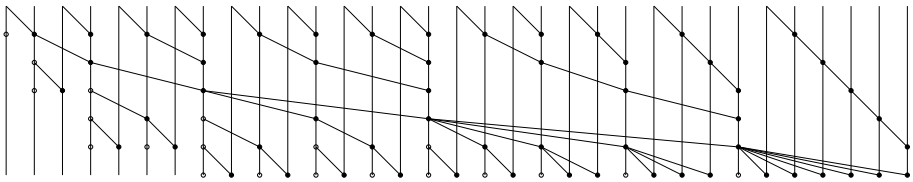


Figure 12: A depth size optimal network of width 33, depth 6 and with maximum fanout 7. This is the widest known minimum depth DSO network.

```
tops1 :: Int -> Int -> [[Int]]
tops1 f n = [replicate k 2 ++ rs ++ [r] | r <- [1..f-1], rs <- perms 2 f (n-r-2*k)]
  where
    k = max 0 ((n `div` 4)-1)
```

This allows us to reach larger-sized examples, for example the network shown in Figure 12, which, incidentally, is the widest known minimum depth DSO network.

The next step that we took was to consider vastly fewer permutations, by modifying the `perms` function to generate only sorted lists.

```
tops2 :: Int -> Int -> [[Int]]
tops2 f n = [replicate k 2 ++ rs ++ [r]| r <- [1..f-1], rs <- permsUp 2 f (n-r-2*k)]
  where
    k =  max 0 ((n `div` 4)-1)

permsUp :: Int -> Int -> Int -> [[Int]]
permsUp _ _ 0   = [[]]
permsUp l g n   = [x:ts | x <- [l..g], x <= n, ts <- permsUp x g (n-x)]
```

This is a major restriction in that it can cause us, in a few cases, to miss attractive solutions (see Figure 13). However, in these cases, it is always possible to increase the fanout by one and still find a DSO solution within the given constraints on width and depth. It is also such a great reduction in the search space that it allows us to consider much larger examples. To understand why this is, just consider the number of partitions being considered by `tops0` and `tops2` at some chosen sizes for fanout 4: for 32 inputs, `tops0` enumerates 109322 partitions, while `tops2` produces only 25; for 50 inputs, the two figures are 106373551 and 56! Calculating the length of `tops0 4` for 64 inputs did not complete when run overnight on a laptop, while the length of `tops2 4 64` is only 81. If we are interested in larger examples, we must take this step. For moderate width networks, this choice to narrow the search completely dominates the effect of the memoisation that we introduced earlier. However, since the memoisation seems to do little harm, we choose to persevere with it. It will come into its own later.

Next, we introduce a measure function, which should have type `[Net] -> Int`. Now, instead of choosing the first successful result from among the networks generated for the partitions at the top level, we choose the best according to the measure function `opt`. That is, a call of `listToMaybe` is replaced by `bestOn is opt`. See Appendix B for the definition of `bestOn`.

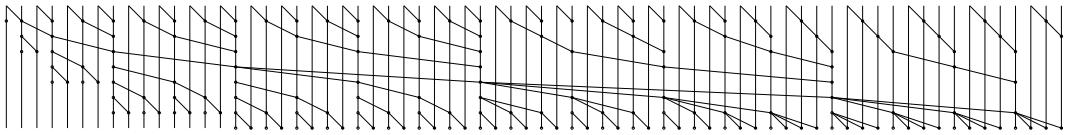


Figure 13: Considering only sorted partitions is indeed a restriction. This is the widest DSO network that we can find for fanout 4 and depth 8. But we know that there are networks of widths 71 and 72. However, those networks require the partition to be unsorted. The reader might like to try to find the two positions in this network at which it would have been possible to have three-input rather than two-input serial networks at the top.

```
parpre2 :: Int -> ([Net] -> Int) -> Context -> ANW
parpre2 f opt ctx = maybe (error "no fit") unWrap (prefix ctx)
  where
    prefix = memo pm

    pm ([i],o) = try wire ([i],o)
    pm (is,o) | fits bser (is,o) = Just bser
    pm (is,o) = bestOn is opt $ mapMaybe makeNet (tops2 f (length is))
      where
        ...
```

Measure functions allow the network designer to gain further control of the details. For instance, one can define a measure function that penalizes each occurrence of a higher fanout, and use this to more finely control fanout. This kind of fine control is most relevant in the context of designing and analyzing actual VLSI circuits. functions used in actual circuit design may demand more information in the context used in the search. For instance, it makes sense to include not only input delays but also integers representing position on a horizontal axis, to enable the calculation of wire-lengths. In this paper, we concentrate on parallel prefix networks at a more abstract level.

Depth Size Optimal networks are interesting, and have been much studied (see for example the recent paper by Lin et al [2009], which contains a good list of relevant references). Here, we have shown that finding DSO networks can be done simply and effectively by generalizing the Brent Kung construction and using dynamic programming to search for solutions. Our previous work on DSO network construction [Sheeran & Parberry, 2006] was the first to produce DSO networks while retaining control of fanout. However, it was a two stage process that first produced a maximum width DSO network for the given fanout and depth, and then reduced the width by deleting "wires". The search-based method presented here retains fine control of fanout, and also produces a DSO network directly for the given width, depth and fanout (if possible). Most other approaches to producing DSO networks either restrict attention to a particular small fanout, most often 2 or 4 (for example [Lin & Hung, 2009]), or tackle the simpler case in which the maximum fanout is the same as the depth, as in [Zhu *et al.* , 2006].

However, we are aiming for *shallow* networks, and in general DSO networks do not exist at minimum depth. Remember that Snir, who proved the key lower bound, explicitly only considered depths in the range 2 $log_2 n - 2$ to $n - 1$ for n inputs. Shallower networks than this are much less well understood, with the main work so far having been done by Fich in the 1980s [Fich, 1983; Fich, 1982]. We have the programming tools to explore the design of small shallow prefix networks in a rather experimental way. The idea of using search can still be used, but we need to further generalize. It turns out that what is needed is a generalization of the Ladner Fischer construction [Ladner & Fischer, 1980].

5 The Ladner Fischer family of networks

Ladner and Fischer [1980] is a wonderful paper that introduces a family of prefix networks. An additional parameter, the *slack*, indicates by how much the network is allowed to exceed the minimum depth. It produces *restricted* networks, in which the last output is produced at minimum depth. The base case is independent of the slack parameter (and of the operator):

```
ladF :: Int -> NW a
ladF _ _ [a]   = [a]
```

When the slack is zero, indicating a minimum depth network, we use a construction very similar to Sklansky:

```
ladF 0 op as   = init los ++ ros'
  where
    (los,ros) = (ladF 1 op las, ladF 0 op ras)
    ros'      = fan op (last los : ros)
    (las,ras) = splitAt ((length as + 1) `div` 2) as
```

Note the left-hand recursive call, with *slack* one instead of zero; the two recursive calls are different, unlike in the Sklansky construction. This is a key point, often missed by those referring to Ladner Fischer, leading to the wide-spread misconception that the Ladner Fischer and Sklansky constructions are identical. With *slack* one on the left, we make use of the available depth on the left-hand side, but produce the last output of the recursive call at minimum depth, so as not to disturb the overall depth of the network. This construction matches exactly Figure 3 in [Ladner & Fischer, 1980].

The following definition captures the case when the slack is greater than zero (Figure 4 in [Ladner & Fischer, 1980]):

```
fans = repeat fan

ladF :: Int -> NW a
ladF n op as = build (twos (length as)) sers (wire:fans) (ladF (n-1)) op as
  where twos 1 = [1,0]
        twos 2 = [2,0]
        twos n = 2 : twos (n-2)
```

We have chosen not to place buffers anywhere, as buffers are not mentioned in the Ladner Fischer paper. Thus, the placement of small unbuffered fan networks across the bottom is controlled by the list `wire:fans`, rather than by `bufall:bfans` as in the Brent Kung example.

Observing the recursive structure of the network, or indeed copying the recurrence in [Ladner & Fischer, 1980], it is easy to write down a function to calculate the network size:

```
ladSize :: Int -> Int -> Int
ladSize k 1 = 0
ladSize 0 n = ladSize 1 (ccc n)+ ladSize 0 (n `div` 2) + n `div` 2
ladSize k n | even n = ladSize (k-1) (ccc n) + n-1
ladSize k n | odd n  = ladSize (k-1) (ccc n) + n-2
  where
    ccc n = n - n `div` 2

*Main> ladSize 0 64
168
*Main> ladSize 1 33
63
```

(a) (b)

Figure 14: (a) LF_0, the minimum depth Ladner Fischer network, for 9 inputs; it has 13 operators and depth 4 (b) A smaller 9-input network (with 12 operators, depth 4) made from $LF_1 8$, and one further operator. This network is DSO. This illustrates the fact that LF_0 does not always give optimal networks.

Ladner and Fischer made a particular choice about how to divide up the network when applying the Sklansky- and Brent Kung-like patterns, and stated as an open problem the determination of just how to split the network to optimize the construction. They were well aware that their choice was not optimal, and gave the small concrete example shown in Figure 14. On the left is Ladner Fischer with zero slack for nine inputs. On the right is a network that adds one extra input and a single operator to Ladner Fischer with slack one for 8 inputs. The result is a smaller prefix network than that on the left. So it is reasonable to try to improve on the classic Ladner Fischer construction. Here, we will experimentally find particular solutions to the open problem.

In order to try to find better solutions than the choices made by Ladner and Fischer, we will again use the idea of searching through partitions, as we did with Brent Kung. To encode the Ladner Fischer pattern using `build`, one needs *two* recursive calls, one sandwiched between the top and bottom serial networks and fans as before, and one at the right-hand end of the top row of small prefix networks, the T_n in Figure 8.

Earlier, we used the function `build0` to encode the generalized Brent Kung pattern. Similarly, `build1` encodes the more general Ladner Fischer pattern, with two recursive calls (the q and p parameters):

```
build1 :: [Int] -> WNW -> WNW -> WNW
build1 ss q p = Wrap (build ss ((replicate (length ss -1) ser)++[unWrap q])
                                (bufall:bfans) (unWrap p))
```

The `q` parameter is placed at the end of the top list of prefix networks.

It now also makes sense to introduce a new integer input constraining the width of the new recursive call, and thus of the associated fanout. Earlier, the parameter `f` controlled the small serial networks and matching fans. Think of `g` as controlling the larger fans that will now arise as we aim for shallow networks. We also introduce a new base case for the case when there is no hope of fitting a prefix network in the given context, and we divide the partition generation into two cases, for shallow and deeper networks.

```
parpre3 :: Int -> Int -> ([Net] -> Int) -> Context -> ANW
parpre3 f g opt ctx = maybe (error "no fit") unWrap (prefix ctx)
  where
    prefix = memo pm

    pm ([i],o) = try wire ([i],o)
    pm (is,o) | 2^(maxd is o) < length is = Nothing
    pm (is,o) | fits bser (is,o) = Just (Wrap bser)
    pm (is,o) = bestOn is opt $ mapMaybe makeNet (tops3 permsUp (is,o) f g)
      where
        makeNet ds = do let sis = split ds is
                        let js = map (last.(ser delF)) $ init sis
```

```
                        pr <- prefix' $ last sis
                        p  <- prefix' js
                        return $ build1 ds pr p
        prefix' ins = prefix (ins,o-1)

maxd :: [Int] -> Int -> Int
maxd [] _    = 0
maxd (a:_) o = max 0 (o-a)
```

The new version of the partitions generation function not only takes the context and the two fanout parameters as inputs, but also the permutation function `p` to be used. This makes it easier to experiment with varying how the permutations are generated.

```
alog2 1 = 0
alog2 n = 1 + alog2 (n - n `div` 2)

tops3 p (is,o) f g | maxd is o == alog2 n
  = [replicate k 2 ++ rs ++ [r]| r <- [1..g-1], rs <- p 2 f (n-r-2*k)]
  where
    n = length is
    k = max 0 (n `div` 8)
tops3 p (is,o) f g | maxd is o > alog2 n
  = [l:replicate k 2 ++ rs ++ [r]|
     l <- [2..maxd is o], r <- [1..g-1], rs <- p 2 f (n-l-r-2*k)]
  where
    n = length is
    k = max 0 ((n `div` 4)-2)
```

This approach gives identical results to the classic Ladner Fischer construction for width a power of two and minimum depth. However, it improves upon Ladner Fischer in all other cases, in that it requires fewer operators and smaller fanout. For example, in the 9-input example shown in Figure 14(b), it does indeed find a DSO network.

Really, there are two separate generalizations: the search permits the small prefix networks across the top to be wider than two, and is also choosing how wide to make the topmost recursive call, and so making better choices than that hard-wired into the Ladner Fischer construction. Both lead to improvements. For example, for 64 inputs, depth 7, restricting `f` to 2 gives the network shown in Figure 15, with 129 operators, fanout 12. For comparison, Ladner Fischer has 137 operators and fanout 17 for the same width and depth. Allowing `f` to increase to 3 reduces fanout and size further in our construction, giving fanout 9, size 126 or fanout 10, size 125. The second of these networks is shown in Figure 16. It is as we begin to consider these wider networks that the memoisation finally comes into its own. For instance, for minimum depth, 256 inputs, small fanout 4, using memoisation reduces the time taken to find a smallest solution from well over an hour to well under 2 minutes (on one core of a Dell M1330 laptop with an Intel Core2 Duo 2.2GHz CPU T7500 and 3.5 GB of RAM). Our general aim has been to do everything necessary (including sacrifice some solutions) in order to keep the time to find a solution below 2 minutes.

This degree of improvement that we have achieved over classic Ladner Fischer is indeed encouraging. Figure 17 illustrates the improvement for both minimum depth and minimum depth plus one, and for both fanout and size. For minimum depth, both size and fanout match classic Ladner Fischer when input width is a power of two, only to take a jump downwards at the next width, then increasing again to finally match Ladner Fischer again at the next power of two. For minimum depth plus one, the same saw-tooth pattern is observed, but now both fanout and size are smaller than for the classic case, and increasingly so for increasing width.

We can go further, however, by questioning the choice to use serial networks along the top of the partition, when aiming for shallow networks. Remembering that it is only the last outputs of these small networks that are used in the following recursive call, it makes sense to consider using

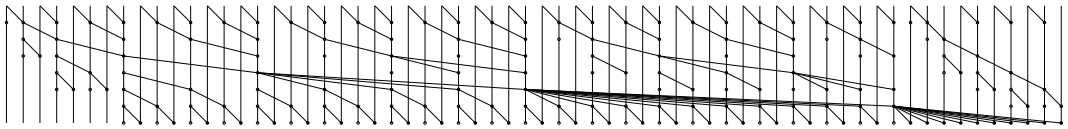


Figure 15: The prefix network of width 64, depth 7 found using dynamic programming, with the widths of the small serial prefix networks across the top still restricted to 2. It has size 129 and fanout 12. This is already a considerable improvement on the classic LF construction.

.

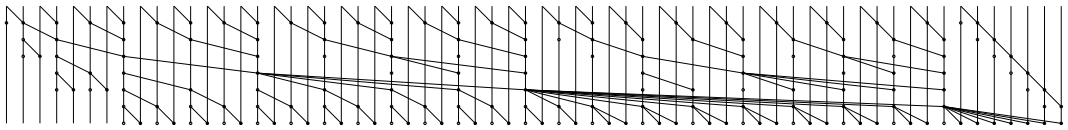


Figure 16: The prefix network of width 64, depth 7 found using dynamic programming, with the widths of the small prefix networks across the top now allowed to be 3. This gives further improvement on the classic LF construction. Here, the general construction gives fanout 10, size 125, while classic LF gives fanout 17, size 137.

.

restricted networks that produce that last output at minimum depth, but that save on size by being deeper for other outputs; but we already have such networks in the form of classic Ladner Fischer. (The new definition (`parpre4`) is the old one (`parpre3`) with the two occurrences of `ser` replaced by `adLadF`, see Appendix B.) We define that variant of Ladner Fischer as follows:

```
ln2 1 = 0
ln2 n = 1 + ln2 (n `div` 2)

adLadF comp as = ladF (ln2 (length as) - 2) comp as
```

Note how the slack parameter depends on the width of the input. It is chosen to give the minimum size Ladner Fischer network for the given width.

Now, in the search for non minimum depth networks, we can reduce fanout further, but sometimes at the expense of size, since we have replaced *all* the serial networks by Ladner Fischer and that is not always a good idea. However, for minimum depth examples, we gain a further improvement, even for width a power of two. For 64 and 128 inputs, we beat classic Ladner Fischer by one and five operators respectively, see Figure 18.

This is actually an unexpected and positive result. Ladner Fischer has often tacitly been assumed to produce the smallest possible prefix network for width a power of two and minimum depth. For instance, Fich is sometimes quoted as stating that Ladner Fischer gives optimal networks in that case, but actually her statement is only about the deepest variant, which is very similar to Brent Kung [Fich, 1983], and indeed Fich goes on to improve on the Ladner Fischer construction even for minimum depth. Here, we have concrete examples supporting the assertion that Ladner Fischer is not optimal for minimal depth. The two networks that we have found for 64 and 128 inputs are previously unknown, as far as we can ascertain, and are smaller than any known minimum depth networks for these widths. So we are entering unknown territory, and for a class of networks (depth d, width 2^d) that is of interest in many applications. Let us concentrate on this class, and see how far we can go. Our Haskell implementation makes it possible both to experiment with network design and with ways to constrain the search space. It is easy to experiment, though perhaps not so easy to convey the process in a paper. For instance, we were surprised to find that it is very useful to generate huge network diagrams that are much too large

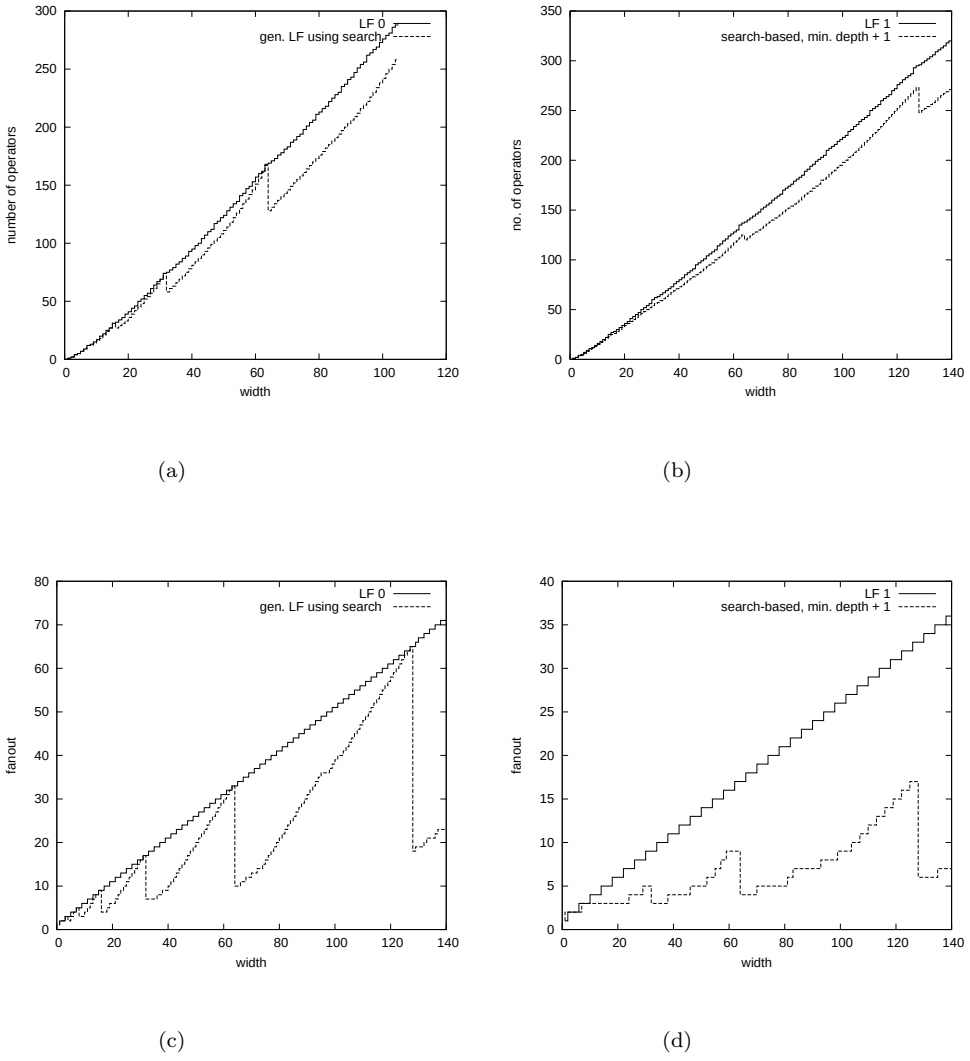


Figure 17: Comparing the new generalized search-based construction with classic Ladner Fischer for minimum depth (a) and (c) and for minimum depth + 1 (b) and (d)

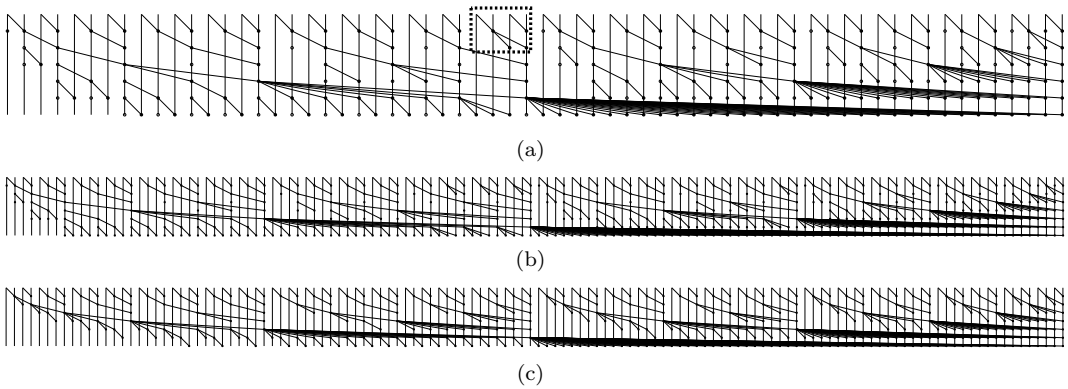


Figure 18: (a) The 64 input, depth 6, network obtained by allowing small Ladner Fischer networks to replace small serial networks. The 4 input LF network that leads to a saving of one operator over classic Ladner Fischer (Figure 4(a)) is marked with a dotted box. (b) The 128 input, depth 7, network obtained in the same way. This has 364 operators, compared to 369 for Ladner Fischer. For these widths, smaller minimum depth prefix networks are not known. (c) Classic Ladner Fischer for 128 inputs, minimum depth. Note how the big fans are still at the same places as in the more complicated construction in (b) just above.

to be printed and to browse them using `xfig`.

Examining the 64 and 128 input networks that we have just generated, we note that none of the small fans has width 3; only 2 and 4 are chosen. Might this be a pattern? It is easy to replace `permsUp` by `permsUp2`, to capture the notion that we consider only small restricted networks whose widths are powers of two.

```
permsUp2 :: Int -> Int -> Int -> [[Int]]
permsUp2 _ _ 0   = [[]]
permsUp2 l g n   = [x:ts | x <- [l,2*l..g], x <= n, ts <- permsUp2 x g (n-x)]
```

This allows us to get results for 256 and 512 inputs too, with 773 and 1614 operators respectively, compared to 792 and 1672 for classic Ladner Fischer. The important point, though, is that we can examine the "winning" partitions, and try to spot a pattern that would allow us to develop and analyse a new prefix construction. Examining the 64 and 128 input cases (Figure 18), the outermost partitions are `[2,..,2,4,32]` and `[2,..2,4,4,4,64]`. Those found for 256 and 512 inputs are `[2,..,2,4,4,4,4,4,4,4,4,8,128]` and
`[2,..,2,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,8,16,256]`.

Now, one can try reversing the order in which the possible partitions are considered, by replacing the call of the generator `(tops3 (permsUp2 (is,o) f g)` by `(reverse (tops3 (permsUp2 (is,o) f g))`. Then, the pattern for 256 inputs contains 11 4s and two 8s, while that for 512 inputs has 17 4s, 5 8s and one 16. It is also instructive to compare the resulting networks with classic Ladner Fischer instances of the same sizes. We note that the large fans occur at exactly the same points, both when the new pattern is used, and when it is not. These division points correspond to one half, one quarter, one eight, and so on of the overall network width. Because we are using small restricted prefix networks of widths a power of two along the top, we can always relate the delays of the inputs to the next stages of the network back to the width of the group of original inputs that are processed. Thus, we can, for instance, divide such a list of delays at its real half-way point, that is the half-way point of the original inputs:

```
getLeftHalf :: [Int] -> [Int]
getLeftHalf as = cutLeft as ((sum [2^i |i <- as]) `div` 2)
```

```
cutLeft :: Int -> [Int] -> [Int]
cutLeft as     0 = []
cutLeft (a:as) k | 2^a <= k = a:cutLeft as (k-2^a)

*Main> getLeftHalf [1,1,1,1,2,2,3,3]
[1,1,1,1,2,2]
```

Now, after some experimenting with patterns, we find that a working sequence of patterns has as at the right hand end the following sub-sequences: (2*4), (4*4,1*8), (8*4,2*8), (16*4,4*8,1*16), (32*4,8*8,2*16). The new patterns appear always on one quarter of the overall width, on the second quarter from the left. Thus, we define functions that indicate how a power of two inputs should be divided up:

```
pat :: Int -> [Int]
pat 0 = []
pat k | k < 4 = replicate (2^(k-1)) 2
pat k = replicate (2^(k `div` 2)) 2 ++ concat [replicate (2^(k-2*j+1)) (2^j) |
                                               j <- [2..(k+1) `div` 2]]

*Main> pat 5
[2,2,2,2,4,4,4,4,8]
*Main> pat 6
[2,2,2,2,2,2,2,2,4,4,4,4,4,4,4,4,8,8]
*Main> sum it
64
```

We can try forcing the function the partition generation to choose only sequences ending this way at appropriate sizes, to gain further insight into possible choices. Finally, we are able to define the generation function completely, so that it only presents one choice for each list of input delays.

```
tops :: [Int] -> [Int]
tops is = fill l (pat (a-2)) ++ [lis-l]
  where
    lis = length is
    a   = alog2 lis
    l   = length $ getLeftHalf is

fill :: Int -> [Int] -> [Int]
fill k as = replicate y 2 ++ as
  where y = (k - sum as) `div` 2

*Main> tops (replicate (2^6) 0)
[2,2,2,2,2,2,2,2,2,2,2,2,2,4,4,32]
*Main> tops (replicate (2^7) 0)
[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,4,4,4,4,8,64]
*Main> sum it
128
```

This places a sequence of twos along the left quarter and the new pattern along the next quarter. The result is the following, pleasingly simple, definition of a new parallel prefix construction.

```
build2 :: [Int] -> NW a -> NW a -> NW a
build2 ss q p = build ss ((replicate (length ss -1) adLadF)++[q]) (bufall:bfans) p

ppfinal :: Int -> NW a
ppfinal k = parpre (replicate (2^k) 0)
```

```
parpre :: [Int] -> NW a
parpre []      = wire
parpre [a]     = wire
parpre [i1,i2] = ser
parpre is      = build2 ss (parpre (last sis)) (parpre js)
  where
    ss  = tops is
    sis = split ss is
    js  = map (last.(adLadF delF)) $ init sis

*Main> check (ppfinal 10) (2^10)
True
```

The first parameter to `parpre` is the list giving input delays Figure 19 shows the new construction for 256 inputs.

Fich [1983] proposed a generalization of Ladner Fischer in which the small prefix networks are small Ladner Fischer networks of width 8, also in the second quarter from the left, where we have placed our wider networks. Fich's construction is larger than ours, as can be seen from Figure 20. It is again easy to transliterate Fich's recurrence for the size of her construction to Haskell:

```
fichK :: Int -> Int -> Int
fichK _ 0 = 0
fichK k n | n <= 3 = ladSize k (2^n)
fichK 0 n = fichK 1 (n-1) + fichK 0 (n-1) + 2^(n-1)
fichK 1 n = fichK 1 (n-2) + fichK 0 (n-4) + 27*(2^(n-4))-1
```

Here, n is the log of the input width, which is 2^n, while k is the slack, as in the Ladner Fischer construction. Looking at the resulting sizes (see Table 1), it clearly makes sense to instead use the Ladner Fischer construction for slack 0 for $n <= 8$. This is achieved here simply by adding a new case above that for `fichK 0 n`:

```
fichK 0 n | n <= 8 = ladSize 0 (2^n)
```

This gives slightly smaller networks.

Fich's paper also mentions that it is better to remove the restriction to sub-networks of width eight, and instead to have networks of increasing size; this is explored in her thesis [Fich, 1982]. Our construction is a little smaller even than that presented in Fich's thesis, and could be viewed as a refinement of it. It gives, as far as we know, the smallest known depth d parallel prefix networks for 2^d inputs. That we could improve on the best known available results was, we think, due to the fact that we could fine tune the construction with the help of the Haskell implementation.

For completeness, Appendix B contains the recurrence `fl` and associated functions that characterize the size of our new construction (and match exactly the results generated from the `ppfinal` function above). These functions seem more complicated than necessary, and could doubtless be simplified. (We hope to find simpler versions for inclusion in the final paper.) The functions are provided for readers who may be aiming to produce smaller networks for these widths and depths.

Figure 20 compares prefix network size divided by number of inputs for minimum depth, with 2^k inputs, for our construction, that in Fich [1983] and Ladner Fischer. To put it concisely, the classic construction requires approximately 4 operators per input, Fich requires a little under 3.55, and our construction brings that number below 3.5. The following table also lists network sizes for the three constructions, for depth d networks of width 2^d. There is no reason to believe that one cannot do better, so interested readers may wish to try to improve on these results.

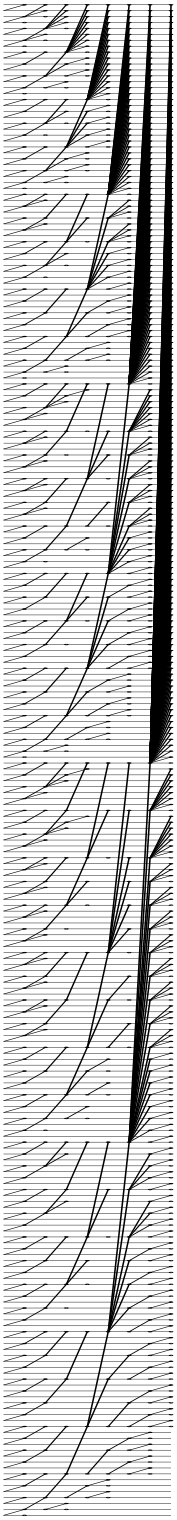


Figure 19: Our final construction, generated by the function `ppfinal`, for 256 inputs. It has 773 operators, an improvement on the 792 of the classic Ladner Fischer construction.

.

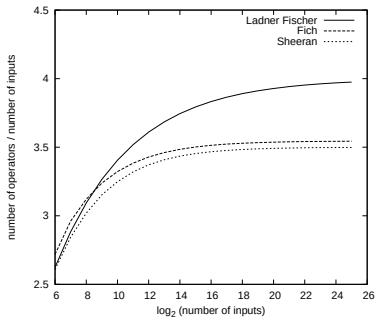


Figure 20: Diagram giving number of operators per input for the three constructions (Sheeran, Ladner Fischer and Fich).

Table 1: Sizes of minimum depth networks for our new construction (Sheeran), Ladner Fischer[1980] (LF) and Fich[1983]

depth	Sheeran	LF	Fich
6	167	168	174
7	364	369	379
8	773	792	799
9	1614	1672	1658
10	3327	3487	3402
11	6800	7206	6930
12	13809	14788	14044
13	27922	30185	28354
14	56275	61356	57093
15	113172	124308	114740
16	227221	251199	230280
17	455702	506578	461714
18	913175	1019920	925095
19	1828888	2050785	1852597
20	3661337	4119280	3708669
21	7327770	8267216	7422354
22	14662683	16580799	14851947
23	29335580	33236622	29714342
24	58685469	66594636	59443763
25	117391390	133385689	118909290

6 Discussion

With the help of simple functional programming techniques, we have been able to solve some open problems in prefix network design. We have shown how to generate Depth Size Optimal networks for a given input width and required depth, while retaining control of fanout. This has not been achieved before. In addition, we have shown how to generate small shallow networks when DSO networks do not exist for the given width and depth. Here, we generate the smallest known networks for the given constraints. Our work highlights a surprising gap in the theory of prefix networks. Little is known about small, shallow networks and optimality. Ideally, we would like to find a result like Snir's for DSO networks [Snir, 1986]. It is possible that the kind of experimentation that our Haskell implementation of prefix networks permits will contribute to the development of the necessary theory. We encourage our readers both to contribute to the theory and to push the limits by improving on the concrete prefix network constructions presented here.

For modest network widths (up to about 256 inputs), we have an immediate route from the kinds of network descriptions shown here and VLSI circuits. This is done via the Wired system, which is a DSL for low level hardware design, embedded in Haskell [Axelsson, 2008]. Wired allows the layout and analysis for power and speed of prefix networks for particular operators, and one can then search (for example) for low power networks. Liu et al have used Integer Linear Programming, with quite fine modelling of wire lengths, capacitance and other physical details, to find optimal prefix networks. However, the fineness of the modelling seems to have limited the approach to working on 8-input networks – although the results can then be used to build hierarchical networks [Liu *et al.* , 2007]. We are aware, through discussions with J. Vuillemin, that search was also used in finding good topologies for 64-bit adders in Alpha microprocessors at DEC Research Labs in Paris in the 1990s, but unfortunately that work has not been published [Vuillemin, 2006]. Thus, the route from the results presented here to hardware or FPGA implementation seems relatively straightforward. We are not so sure, though, how our results for very large prefix networks might be used to create software implementations. Here, the fact that there is some irregularity in the resulting networks may make it difficult to generate code for a multi- or manycore machine, such as a GPU. We plan to investigate this question in the context of our work on a DSL for GPU programming [Svensson *et al.* , 2009]. It should be noted that parallel prefix is a very important library function in GPU programming, see for example [Sengupta *et al.* , 2007].

For parallel prefix networks at a more abstract level, we have shown that functional programming can, through modest programming effort, provide a workbench that enables hands-on design exploration and ultimately new insights into prefix networks. Our hope is that others interested in prefix networks will find this useful in teaching or in research.

The central function is the very general `build` combinator. Hinze presents a very similar combinator when himself generalizing the Brent Kung construction (p. 16 of [Hinze, 2004]). In our notation, his combinator and the associated definition of the binary Brent Kung construction would be written

```
buildH :: [Int] -> [NW a] -> [NW a] -> NW a -> NW a
buildH ss ts bs p comp = split ss ->- parL [t comp | t <- ts]  ->-
                         toLasts (p comp) ->- concat ->-
                         split (shift ss) ->- parL [b comp | b <- bs] ->- concat
  where shift (a:as) = a-1:as ++ [1]

bk2 _ [a] = [a]
bk2 op as = buildH (twos (length as)) sers (bufall:bfans) bk2 op as
  where twos n = replicate (n `div` 2) 2 ++ [1 | odd n]
```

The hard-wiring of a 1 as the last element of the bottom partition (given by the function `shift` above) restricts the choice of network shapes that can be described. The function `bk2` always produces its last output at minimum depth, and indeed one might want this. However, it is not easy to modify the definition to produce the kind of symmetrical network that we produced earlier using `bk0`.

For ease of reading, we include our definition of `build` again:

```
build :: [Int] -> [NW a] -> [NW a] -> NW a -> NW a
build ss ts bs p comp = split ss ->- parL [t comp | t <- ts]  ->-
                        toInit (toLasts (p comp)) ->- concat ->-
                        split (shift ss) ->- parL [b comp | b <- bs] ->- concat
  where
    shift (a:as) = a-1 : init as ++ [last as + 1]
```

The `build` combinator is more general and a little more symmetrical in how it treats the top and bottom partitions. We could have defined `buildH` in terms of `build` as

```
buildH ss ts = build (ss++[0]) (ts++[ser])
```

but the reverse is not possible. The function `build` can be defined using `buildH` and Hinze's serial and horizontal scan combinators. That we choose to have one very general combinator is closely related to our decision to use search to find good networks. We have concentrated on ways to make the description of the search itself simple, and would assert that we have achieved this goal. For the purposes of reasoning about network constructions, Hinze's choice of a small set of combinators is likely better. Calculational methods like Hinze's and O'Donnell's (see for instance [O'Donnell & Ruenger, 2004]) are indeed powerful and instructive, as are the methods used by those who manually construct prefix networks with particular properties (for example [Lin *et al.* , 2003]). Hinze's elegant derivations are combined with the ability to generate diagrams of the resulting networks (as in our case). We certainly found the concrete diagrams extremely helpful in both our own work and in reading Hinze's work. We note too that several purported derivations of Ladner Fischer in the literature are actually of the simpler Sklansky construction. So one wonders where the limits of calculation are. We feel that the approach proposed here, in which the user describes how the required network should be constructed, but allows the details to be found through search, offers new possibilities.

It should be noted that we have concentrated here on prefix networks whose input delay profile is flat. We do have the notion of context, which enables the consideration of other input delay profiles. However, the simple search described here is designed to work only for flat delay profiles and for the increasing profiles that tend to appear in sub-networks when the outermost profile is flat. To deal with profiles that are increasing and then decreasing, of the sort found on the input to the fast adder in standard multiplier constructions, it is necessary to complicate the method a little.

We regard this paper as contributing not only new ideas about prefix network design but also a new programming idiom that may have wider application. The combination of combinators and search, implemented as a simple form of lazy dynamic programming, is an appealing one. Our emphasis has been on keeping things simple. The key idea is to describe the shape of the required construction or data-type and to allow search to fix the small details. It is a little surprising that such a simple approach worked in the case of prefix networks, when the initial search space is huge. It must also be admitted that our initial attempts to solve this problem were not nearly as simple as the final search-based solution shown here. We have exploited higher order functions, laziness and the notion of non-standard interpretation to form this new idiom. We have avoided the need for more sophisticated search strategies by finding ways to restrict the search space, and accepting less than optimal results. Our instincts tell us that a good understanding of the search space is always going to be necessary, so that it is better to concentrate ones intellectual efforts on understanding the problem at the higher level, so that the resulting generators remain very simple. Note that we have not needed to think in terms of matrices, recurrences, tabulation and so on, as would be usual in more traditional dynamic programming approaches. In this, our approach resembles a simple variant of Algebraic Dynamic Programming (ADP) [Giegerich *et al.* , 2002]. For our purposes so far, we have not needed sophisticated ways to construct and examine the search space. However, if we wanted to search for prefix networks using much more complicated objective functions (as in [2007], for example), then it would make sense to investigate the ADP framework.

Is there a danger that the work presented here forms a one-off result that is not repeatable for other algorithms or in more complicated settings? Having had experience of describing and reasoning about both sorting networks [Claessen *et al.* , 2001] and median networks [Sheeran, 2003], we feel confident that such networks could also be explored and possibly improved upon using some of the ideas presented here. In addition, we have found that many algorithms used in general purpose GPU programming come into this category of "circuit-like", so we see possible application of these ideas also in our work on a DSL for GPU programming [Svensson *et al.* , 2009]. We have been inspired by the results of the SPIRAL project at CMU, in which platform-tuned DSP and numerical kernels are generated using a variety of methods including search [Püschel *et al.* , 2005; Franchetti *et al.* , 2009]. We note that the project to develop the Feldspar DSL for Digital Signal Processing that we are engaged in with colleagues from Chalmers, Ericsson and ELTE University Budapest [Feldspar, 2009] provides ample opportunities to find, merge or transform data-flow like networks that form signal processing chains; we believe that search can play an important role here. This is where we expect the new programming idiom to be most practically useful.

7 Conclusion

This paper has shown how simple functional programming techniques can be used to make a rather deep investigation of an important topic in algorithm design – parallel prefix networks. For those who know about functional programming, it can be a tutorial on prefix networks – and there is a need for such a tutorial as the literature is littered with misconceptions. More importantly, though, we hope that the paper can convince some readers that functional programming can play the role of an experimental workbench in research and teaching about an important class of algorithms. It has been fun to push the limits of prefix network design, and we hope that readers will contribute new ideas, both theoretical and practical. We expect the programming idiom that combines combinators and search to have a broader application; here too, we hope that this paper will be a starting point for new research.

Acknowledgements

This research was funded by a grant from the Swedish Basic Research Funding Agency (Vetenskapsrådet). Thanks to Emil Axelsson and Satnam Singh for constructive criticism of an earlier draft. Thanks to Joel Svensson for providing Figure 2(a), and to Koen Claessen for providing his purely functional memo function. My fascination with prefix networks grew out of my contacts with researchers at Intel Strategic CAD Labs.

References

Axelsson, Emil. 2008. *Functional Programming Enabling Flexible Hardware Design at Low Levels of Abstraction*. Ph.D. thesis, Chalmers University of Technology.

Bjesse, P., Claessen, K., Sheeran, M., & Singh, S. 1998. Lava: Hardware Design in Haskell. *Pages 174–184 of: International Conference on Functional Programming, ICFP*. ACM.

Blelloch, Guy E. 1990. *Prefix sums and their applications*. Tech. rept. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University. Also appears in Synthesis of Parallel Algorithms, Reif (ed.), Morgan Kaufmann, 1993.

Brent, R.P., & Kung, H.T. 1982. A regular layout for parallel adders. *IEEE Trans. Comp.*, **C-31**.

Claessen, K., Sheeran, M., & Singh, S. 2001. The Design and Verification of a Sorter Core. *Pages 355–369 of: Correct Hardware Design and Verification Methods, CHARME*. Lecture Notes in Computer Science, vol. 2144. Springer.

Cormen, Thomas H, Leiserson, Charles E, Rivest, Ronald L, & Stein, Clifford. 2001. *Introduction to algorithms*. Second edn. Cambridge, MA: MIT Press.

Feldspar. 2009. http://feldspar.sourceforge.net/. Feldspar (Functional Embedded Language for DSP and PARallelism), developed jointly by Ericsson, Chalmers and ELTE, Budapest.

Fich, Faith E. 1983. New bounds for parallel prefix circuits. *In: STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of Computing*. ACM Press.

Fich, Faith Ellen. 1982. *Two problems in concrete complexity: cycle detection and parallel prefix computation*. Ph.D. thesis, University of California, Berkeley.

Franchetti, Franz, de Mesmay, Frédéric, McFarlin, Daniel, & Püschel, Markus. 2009. Operator Language: A Program Generation Framework for Fast Kernels. *Pages 385– 410 of: Proc. IFIP Working Conference on Domain Specific Languages (DSL WC)*. Lecture Notes in Computer Science, vol. 5658. Springer.

Giegerich, Robert, Meyer, Carsten, & Steffen, Peter. 2002. Towards a Discipline of Dynamic Programming. *Pages 3–44 of: Informatik bewegt: Informatik 2002 - 32. jahrestagung der gesellschaft für informatik e.v. (gi)*. Lecture Notes in Informatics. Bonner Köllen Verlag.

Han, T., & Carlson, D. 1987. Fast Area-Efficient VLSI Adders. *In: Int. Symposium on Computer Arithmetic*. IEEE.

haskell.org. 2009. The web page gathers information about Haskell, compilers, tutorial materials, packages and much more.

Hinze, R. 2000 (July). Memo functions, polytypically! *In: In Johan Jeuring, editor, Proceedings of the Second Workshop on Generic Programming, WGP 2000*.

Hinze, Ralf. 2004. An Algebra of Scans. *Pages 186–210 of: Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 3125. Springer.

Knowles, S. 1999. A Family of Adders. *Int. Symp. on Computer Arithmetic*, 277–284.

Kogge, P.M., & Stone, H.S. 1973. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, **C-22(8)**, 786–793.

Ladner, R. E., & Fischer, M. J. 1980. Parallel Prefix Computation. *J. ACM*, **27**(4), 831–838.

Lin, Y.-C., & Hung, L.-L. 2009. Straightforward Construction of Depth-Size Optimal, Parallel Prefix Circuits with Fan-Out 2. *ACM Transactions on Design Automation of Electronic Systems*, **14**(1).

Lin, Y.-C., Hsu, Y.-H, & Liu, C.-K. 2003. Constructing H4, a Fast Depth-Size Optimal Parallel Prefix Circuit. *The Journal of Supercomputing*, **24**(3), 279–304.

Lin, Yen-Chun, & Liu, Chun-Keng. 1999. Finding optimal parallel prefix circuits with fan-out 2 in constant time. *Inf. Process. Lett.*, **70**(4), 191–195.

Liu, Jianhua, Zhu, Yi, Zhu, Haikun, Cheng, Chung-Kuan, & Lillis, J. 2007. Optimum Prefix Adders in a Comprehensive Area, Timing and Power Design Space. *Pages 609–615 of: Asp-dac '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*. Washington, DC, USA: IEEE Computer Society.

O'Donnell, J.T., & Ruenger, G. 2004. Derivation of a Logarithmic Time Carry Lookahead Addition Circuit. *Journal of Functional Programming*, **14(6)**, 697–713.

Pippenger, Nicholas. 1987. The complexity of computations by networks. *IBM J. Res. Dev.*, **31**(2), 235–243.

Püschel, Markus, Moura, José M. F., Johnson, Jeremy, Padua, David, Veloso, Manuela, Singer, Bryan, Xiong, Jianxin, Franchetti, Franz, Gacic, Aca, Voronenko, Yevgen, Chen, Kang, Johnson, Robert W., & Rizzolo, Nicholas. 2005. SPIRAL: Code generation for DSP transforms. *Proceedings of the ieee, special issue on "program generation, optimization, and adaptation"*, **93**(2), 232– 275.

Sengupta, S., Harris, M., Zhang, Y., & Owens, J.D. 2007. Scan Primitives for GPU Computing. *Graphics Hardware, Aila and Segal (Editors)*.

Sheeran, M. 2003. Finding regularity: describing and analysing circuits that are almost regular. *Pages 4–18 of: Correct Hardware Design and Verification Methods, CHARME*. Lecture Notes in Computer Science, vol. 2860. Springer.

Sheeran, M., & Parberry, I. 2006. *A new approach to the design of optimal parallel prefix circuits*. Tech. rept. 2006:1. Dept. of Computer Science and Engineering, Chalmers.

Singh, S. 1992. Circuit Analysis by Non-Standard Interpretation. *Pages 119–138 of: Designing Correct Circuits*. IFIP Transactions, vol. A-5. North-Holland.

Sklansky, J. 1960. Conditional-sum addition logic. *IRE Trans. Electron. Comput.*, **EC-9**, 226–231.

Snir, Marc. 1986. Depth-size trade-offs for parallel prefix computation. *J. Alg.*, **7**(2), 185–201.

Svensson, Joel, Sheeran, Mary, & Claessen, Koen. 2009. Obsidian: A domain specific embedded language for general-purpose parallel programming of graphics processors. *In: Proc. of Implementation and Applications of Functional Languages (IFL)* . Lecture Notes in Computer Science. Springer Verlag.

Voigtländer, Janis. 2008. Much Ado about Two: A Pearl on Parallel Prefix Computation. *Pages 29–35 of:* Wadler, Philip (ed), *35th Symposium on Principles of Programming Languages*. SIGPLAN Notices, vol. 43, no. 1. ACM Press.

Vuillemin, J. 2006. Use of dynamic programming to find best topology for given technology for 64 bit adder, work done at Digital in 1992. *private communication*.

Zhu, Haikun, Cheng, Chung-Kuan, & Graham, Ronald. 2006. On the construction of zero-deficiency parallel prefix circuits with minimum depth. *Acm Transactions on Design Automation of Electronic Systems*, **11**(2), 387–409.

A Generating diagrams of prefix networks

Here, we consider the question of how the many parallel prefix diagrams in this paper are generated from the Haskell descriptions we have seen. We introduce a data-type `Net`, which is used to gather information about the network in a form of non-standard interpretation. It encodes information about a particular abstract wire of the network, the current phase, and the components that have appeared on that wire so far. Each component is represented by a pair of its first input wire and a (possibly empty) list of the remaining input wires. An operator is considered to belong to the last of its input wires, and so the operator is recorded as being associated with that wire. In the diagrams, the operator is also drawn on that wire.

```
data Net =
  Net
  { comps :: [(Int,[Int])]
  , phase :: Int
  , wire  :: Int
  }

net :: Int -> Net
net i = Net []  0 i

nets ds = [netd d i | (d,i) <- zip ds [0..length ds-1]]

netd d i = Net [] d i

netdi xs ds = [netd i d | (d,i) <- zip xs ds]

instance Eq Net where
  n == m = wire n == wire m && phase n == phase m

instance Ord Net where
  n <= m = phase n <= phase m

instance Show Net where
  show n = show (wire n) ++ "/" ++ show (comps n)

ops ns = concat (map comps ns)

nops2 ns = length [p | (p,xs) <- ops ns, length xs > 1]
```

We also introduce an operator, `opn`, which operates on a list of `Net`s and produces a list of nets of the same length. (In this paper, those lengths are always one or two.) The operator builds up information about the netlist, adding the information about the current component to the last of its outputs.

```
opn ns = init ns ++ [o]
    where
      ph = maximum (map phase ns)
      m = last ns
      o = Net  { comps = (ph,map wire ns) : comps m
               , phase = ph+1
               , wire  = wire m
               }
```

Now, we can simply run the `skl` function implementing the Sklansky construction on some suitable nets and look at the result:

```
*Main> skl opn (nets (replicate 8 0))
[0/[],1/[(0,[0,1])],2/[(1,[1,2])],3/[(1,[1,3]),(0,[2,3])],
 4/[(2,[3,4])],5/[(2,[3,5]),(0,[4,5])],6/[(2,[3,6]),(1,[5,6])],
 7/[(2,[3,7]),(1,[5,7]),(0,[6,7])]]
```

This gives us, for each abstract wire in the diagram (starting from 0), a list of the operators that appear upon it and their phases (starting from phase 0). Figure 2(b) shows an eight-input Sklansky network. It can be seen that wire zero has no operators on it, wire one has one at phase zero, operating on wires zero and one, and so on. To demonstrate the representation of buffers, we include the `Net`s associated with Figure 10:

```
*Main> bK1 opn (nets (replicate 32 0))
[0/[(0,[0])],1/[(2,[1]),(1,[1]),(0,[0,1])],2/[(2,[1,2])],
3/[(4,[3]),(3,[3]),(2,[3]),(1,[1,3]),(0,[2,3])],4/[(4,[3,4])],
5/[(4,[5]),(3,[3,5]),(0,[4,5])],6/[(4,[5,6])],
7/[(6,[7]),(5,[7]),(4,[7]),(3,[7]),(2,[3,7]),(1,[5,7]),(0,[6,7])],
8/[(6,[7,8])],9/[(6,[9]),(5,[7,9]),(0,[8,9])],10/[(6,[9,10])],
11/[(6,[11]),(5,[11]),(4,[7,11]),(1,[9,11]),(0,[10,11])],
12/[(6,[11,12])],13/[(6,[13]),(5,[11,13]),(0,[12,13])],14/[(6,[13,14])],
15/[(7,[15]),(6,[15]),(5,[15]),(4,[15]),(3,[7,15]),(2,[11,15]),
    (1,[13,15]),(0,[14,15])],
16/[(7,[15,16])],17/[(7,[17]),(6,[15,17]),(0,[16,17])],18/[(7,[17,18])],
19/[(7,[19]),(6,[19]),(5,[15,19]),(1,[17,19]),(0,[18,19])],
20/[(7,[19,20])],21/[(7,[21]),(6,[19,21]),(0,[20,21])],22/[(7,[21,22])],
23/[(7,[23]),(6,[23]),(5,[23]),(4,[15,23]),(2,[19,23]),(1,[21,23]),(0,[22,23])],
24/[(7,[23,24])],25/[(7,[25]),(6,[23,25]),(0,[24,25])],26/[(7,[25,26])],
27/[(7,[27]),(6,[27]),(5,[23,27]),(1,[25,27]),(0,[26,27])],
28/[(7,[27,28])],29/[(7,[29]),(6,[27,29]),(0,[28,29])],
30/[(8,[30]),(7,[29,30])],31/[(8,[30,31])]]

*Main> nops2 (bK1 opn (nets (replicate 32 0)))
53
```

Note how the even numbered wires have only a single binary operator on them.

To produce the pictures, we have written a small Haskell program that takes a list of `Net`s and produces the diagram as a .fig file.

B Definitions of functions used but not defined in the paper

```
toLasts :: ([b] -> [b]) -> [[b]] -> [[b]]
toLasts f as = [is++[l] | (is,l) <- zip (map init as) (f (map last as))]

bestOn :: (Num a,Ord a) => [Int] -> ([Net]-> a) -> [WNW] -> Maybe WNW
bestOn _  _ [] = Nothing
bestOn ds f as = Just (minimumBy (compOn ds f) as)

compOn :: (Num a,Ord a) => [Int] -> ([Net]-> a) -> WNW -> WNW -> Ordering
compOn ds f (Wrap c1) (Wrap c2) = compare (f (c1 opn netdis)) (f (c2 opn netdis))
 where netdis = netdi [0..length ds-1] ds

parpre4 :: Int -> Int -> ([Net] -> Int) -> Context -> ANW
parpre4 f g opt ctx = maybe (error "no fit") unWrap (prefix ctx)
  where
    prefix = memo pm

    pm ([i],o) = try wire ([i],o)
    pm (is,o) | 2^(maxd is o) < length is = Nothing
    pm (is,o) | fits bser (is,o) = Just (Wrap bser)
    pm (is,o) = bestOn is opt $ mapMaybe makeNet (tops3 permsUp (is,o) f g)
      where
        makeNet ds = do let sis = split ds is
                        let js = map (last.(adLadF delF)) $ init sis
                        pr <- prefix' $ last sis
                        p  <- prefix' js
                        return $ build1 ds adLadF pr p
        prefix' ins = prefix (ins,o-1)
```

The function `fl k` calculates the size of our new construction for input width 2^k. It and its two helper functions `fc` and `c` arise from the observation that the prefix networks tend to work either on flat input delays (`fl`), on input delays that are first flat and then increasing or "curved" (flat-curved, `fc`) or on curved (increasing) input delays (`c`).

```
fl 0 = 0
fl 1 = 1
fl 2 = 4
fl k = fc (k-2) + cstfc (k-1) + fl (k-1) + 2^(k-1)

fc k | k <3 = fl k
fc k        = c (k-1) + fc (k-2) + cstfc (k-1) + lpat k

c 1 = 1
c 2 = 2
c k = 2*c (k-2) + cstc (k-2) + lpat (k-1)

cstfc k = curve (k-1) + 2^k - 1 - lpat (k-1)
cstc k  =  curve (k-1) + 2^(k-1) - lpat (k-1) + lqpat k - 1

curve a = 2^da2 + sum [2^(a-k) - ((2^(a-(2*k)-3))*(k+4) )| k <- [0..(a+1) `div` 2- 2]]
  where da2 = a `div` 2

lqpat k | k < 3 = 2^k
lqpat k | k >= 3 = 2^(k-1)+ 2^((k+1) `div` 2)

lpat 0 = 0
lpat a | even a = 2*lpat (a-1)
lpat a = 2^da2 + sum [2^(2*j) | j <- [0..da2-1]]
  where da2 = a `div` 2
```