JFP **21** (1): 59–114, 2011. © Cambridge University Press 2010
doi:10.1017/S0956796810000304 First published online 6 December 2010

Functional and dynamic programming in the design of parallel prefix networks

MARY SHEERAN

CSE Department, Chalmers University of Technology, Göteborg, SE-41296, Sweden
(*e-mail:* ms@chalmers.se)

Abstract

A parallel prefix network of width n takes n inputs, $a_1, a_2, \ldots, a_n$, and computes each $y_i = a_1 \circ a_2 \circ \cdots \circ a_i$ for $1 \leqslant i \leqslant n$, 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 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 a functional programming language, 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. In a parallel prefix *network*, operators are connected in an arrangement 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 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. We are particularly interested in controlling depth, number of operators and allowed fanout from the operators. 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 do not exist, and still maintaining control over fanout. The resulting networks improve on the best known networks. Finally, our experiments with using search lead us to a new construction of minimum depth networks that does not rely on 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 (haskell.org 2009), 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/PPSearch/.

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. 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 $y_i = a_1 \circ a_2 \circ \cdots \circ a_i$ for $1 \leqslant i \leqslant n$, for an associative operator $\circ$. Thus, a prefix network takes n inputs and produces n outputs.

Prefix networks are built from two kinds of nodes, *combining* and *duplicating* nodes. A combining node takes two inputs, a and b, and combines them using the associative operator $\circ$; it has one or more outputs, each carrying $a \circ b$. A duplicating node has one input, a, and two or more outputs, each carrying a. The standard notations for these nodes are black and white circles, respectively, as shown in Figure 1.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

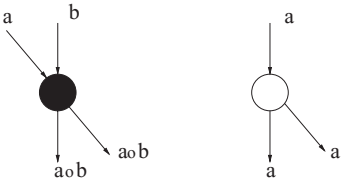


Fig. 1. Building blocks of prefix networks, combining nodes (black) and duplicating nodes (white).

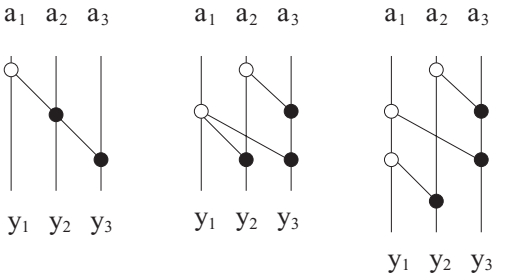


Fig. 2. Three 3-input prefix networks. In each case, $y_1 = a_1$, $y_2 = a_1 \circ a_2$ and $y_3 = a_1 \circ a_2 \circ a_3$.

An n input prefix network consists of at least one duplicating node and at least $n - 1$ combining nodes. The network is said to have *width* n. Figure 2 shows three different 3-input prefix networks, drawn using a notation used in the literature. Each input is represented by a vertical line, and data flows from top to bottom, with the least significant input, a_1, entering at top left. The diagrams also have levels that we count from top to bottom. The topmost level, level zero, contains only duplicating nodes. Combining nodes on level i must receive their inputs from level $i - 1$. We can think of the nodes as taking one time step, and the diagrams encode this time in levels. The highest numbered level (at the bottom of the diagram) corresponds to the *depth* of the network. Here, the leftmost two networks have depth 2, while that on the right has depth 3. In real circuits implementing prefix networks, depth is related to delay, where one time unit is the time taken to perform one operation.

The *size* of a network is the number of combining nodes that it contains. The *fanout* of a node is its out-degree, and of a network is the maximum of the fanouts of its nodes. In Figure 2, the left- and rightmost networks have fanout two, while that in the middle has fanout three. We will typically use w or n for width, s for size, d for depth and f for fanout. To ease presentation, we write S_j^k for $a_j \circ a_{j+1} \circ \cdots \circ a_k$.

Many authors choose to omit the white duplicating nodes in diagrams, as their presence can be inferred whenever a line branches in the absence of a combining node. Here, we make the same choice, viewing the problem of parallel prefix network generation as being synonymous with the problem of generating diagrams of the form shown in Figure 3.

The leftmost network shown in Figures 2 and 3 is a small example of a *serial* prefix network, which has size 2, depth 2 and fanout 2. On each non-zero level, there

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

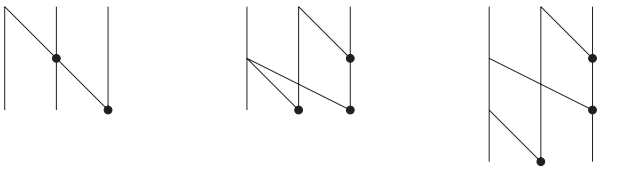


Fig. 3. The three 3-input prefix networks from Figure 2, shown in the style that will be used in the remainder of this paper. The Haskell code used to generate these diagrams is discussed in Section 3.

is exactly one combining node and computation proceeds serially through the levels, and from left to right. Figure 4 shows a serial prefix network of width 8.

In a *parallel* prefix network, there may be more than one combining node on 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 5.

Another very well known network, shown in Figure 6, is due to Brent & 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 fewer operators. In order to maintain a fanout of two at each level, this construction uses duplicating nodes. In Very Large Scale Integrated (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 patterns (Ladner & Fischer 1980), (Figure 7).

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 8. 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

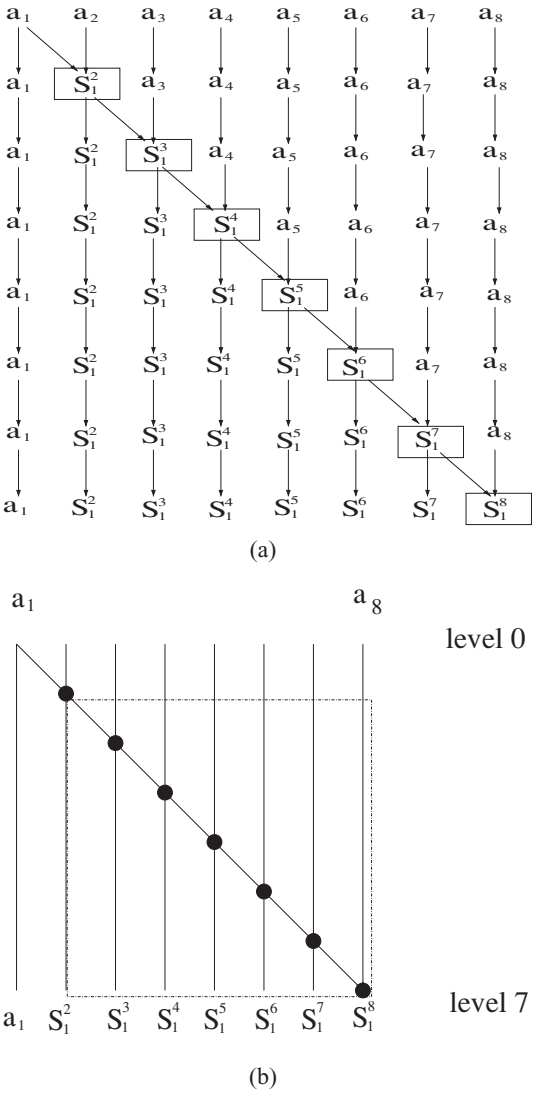


Fig. 4. The serial prefix structure of *width* 8. (a) Shows how the data flows through the network from top to bottom. A box with two incoming arrows is a combining node and shows the result of its computation. (b) Shows how the same network is drawn in the style that is standard for prefix networks. The network shown contains seven combining nodes, and so is said to be of *size* 7. It also has *depth* 7.

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.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

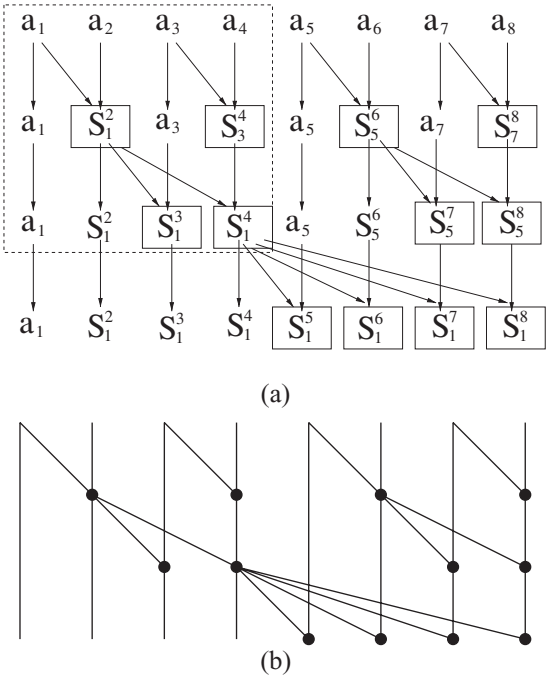


Fig. 5. The Sklansky construction for eight inputs. The construction recursively computes the parallel prefix for each half of the inputs and then combines the last output S_1^4 of the lower half (shown by a dotted box in (a)) with each of the outputs of the upper half. The result, for eight inputs, is that there are four operators on each of three levels.

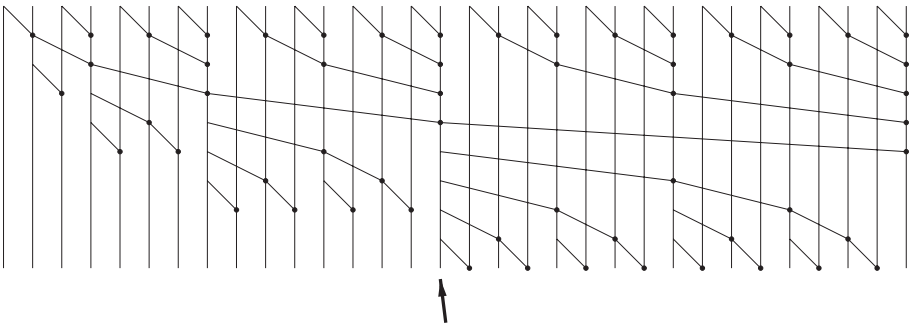


Fig. 6. The Brent–Kung construction, fanout 2, 32 inputs, depth 9, 57 operators. The arrow below the diagram points to a line that has (from top to bottom) first four combining nodes and then four duplicating nodes. The output of the bottom-most combining node on that line is fanned out to a total of six different destinations, but this fanout occurs at successive levels, so that the overall fanout of the network is only 2.

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 the Brent–Kung construction, and show how dynamic programming can be used to find good networks. A lazy functional programming language proves

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

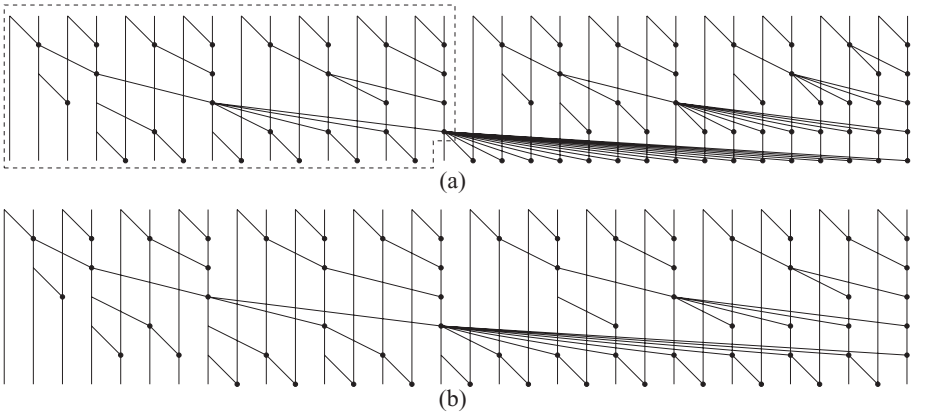


Fig. 7. (a) LF_0, the minimum depth Ladner–Fischer network, for 32 inputs and (b) LF_1, which is one level deeper (depth 6) but contains fewer operators (62 versus 74).

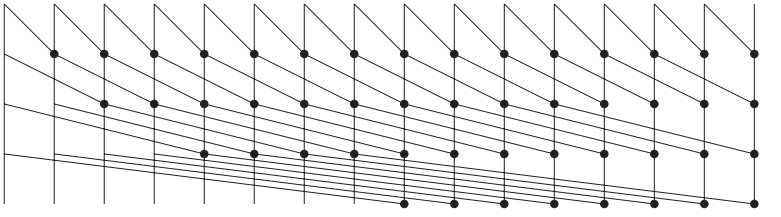


Fig. 8. The basic Kogge–Stone construction, with 16 inputs, depth 4 and fanout 2. For 16 inputs, it has 49 operators, compared to 31 for the minimum depth Ladner–Fischer network.

to be a suitable setting for this approach. To generate small shallow networks, we must generalizefurther; we introduce the classic Ladner–Fischer construction, generalize it, and then use dynamic programming to find solutions that improve upon the classic solution. The insights gained in searching for networks in this way lead finally to a new generalization 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 argue that the functional style of algorithm description presented here 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 Describing some simple networks

A prefix network takes a list of inputs, and returns a list of the same length. We will concentrate on the patterns (or higher order functions) used to construct such networks. The networks are built from k input, k output tiles that are defined as

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
type Fan a = [a] -> [a]

mkFan :: (a -> a -> a) -> Fan a
mkFan op (i:is) = i:[op i k | k <- is]
```

This is a fanout structure, in which the binary operator is applied between the first input and each of the remaining inputs. In the particular case of an input list of length two, it gives a two-input, two-output prefix network, which is a much-used building block. Wider instances of the tile are not themselves prefix networks, but are used particularly in the bottom halves of prefix networks, as we shall see. For example, if the binary operator is addition, then we can make a suitable tile as follows:

```
pplus :: Fan Int
pplus = mkFan (+)

*Main> pplus [1,2]
[1,3]

*Main> pplus [1..8]
[1,3,4,5,6,7,8,9]
```

Now, we want to study higher order functions that capture patterns of building prefix networks from such components.

```
type PP a = Fan a -> [a] -> [a]
```

As a first exercise, let us capture in Haskell the small network patterns shown in Figure 3. The three input serial network is written

```
ser3 :: PP a
ser3 f [a,b,c] = [a1,b2,c2]
  where
    [a1,b1] = f [a,b]
    [b2,c2] = f [b1,c]

*Main> ser3 pplus [0,2,4]
[0,2,6]
```

The middle network contains two tiles, one of width 2 and one of width 3.

```
f31 :: PP a
f31 f [a,b,c] = [a1,b2,c2]
  where
    [b1,c1]    = f [b,c]
    [a1,b2,c2] = f [a,b1,c1]
```

The rightmost network contains three tiles, and the one in the middle connects only the first and last lines.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

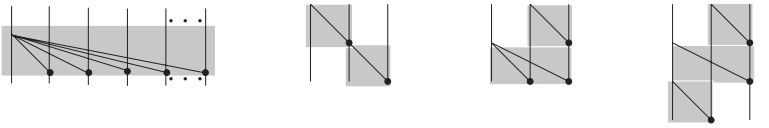


Fig. 9. The general tile that we used to build prefix networks, with three example networks from Figure 3, showing how each is constructed. The composition of tiles is described directly in the Haskell functions.

```
f32 :: PP a
f32 f [a,b,c] = [a2,b2,c2]
  where
    [b1,c1] = f [b,c]
    [a1,c2] = f [a,c1]
    [a2,b2] = f [a1,b1]
```

These examples are illustrated in Figure 9. Notice that the operators in a tile always take their inputs from exactly one level earlier than that on which the operators are placed (a standard constraint in the literature). Each tile also has a flat bottom edge, meaning that it produces all of its outputs simultaneously. This is indicated in the diagrams by the grey shading. The bottom edges of prefix networks are not necessarily flat, in the sense that some outputs may be available earlier than others; when sub-networks are composed, we assume that later tiles appear at as early a level as possible. These small examples are used to illustrate the style of description that we use. More usually the networks that we describe are defined by recursion over the input list, and are designed to work for any input width.

The function `ser` captures the serial connection pattern from Figure 4. It consists of two obvious base cases and a recursive case. The latter includes one use of the building block (the `f` parameter) and a recursive call of `ser`. Figure 4 illustrates this recursive decomposition by showing a network of width 7 in a dotted box. This corresponds to the recursive call of `ser` in the definition.

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

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

3.2 Analysing networks by non-standard interpretation

To analyse the generated networks, we typically use non-standard interpretation (NSI), see for instance (Singh 1992). In NSI, we simply replace the building blocks

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

by ones designed to give the required information when the resulting network is run. For instance, to calculate the delay through the network, we model the fanout structure as

```
delFan :: [Int] -> [Int]
delFan [d] = [d]
delFan ds  = [maximum ds + 1 | i <- ds]
```

Note that the outputs of an individual tile are all assigned the same delay. For an input list of width one, the tile acts as the identity. Otherwise, the output delay of all outputs is one greater than the maximum of the input delays. This approach gives 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> f32 delFan [0,0,0]
[3,3,2]

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

This corresponds to what we would expect by examining Figure 9, and the reader might like to consider where the tiles are in Figure 4, in order to understand the output delays from the serial prefix network defined in the function `ser`.

A variant of `delFan` computes not with delays, but with pairs of a wire number and a delay:

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

wdFan :: WDels -> WDels
wdFan [wd] = [wd]
wdFan wds  = [(w,maximum ds + 1) | w <- ws]
  where (ws,ds) = unzip wds

zdel :: Int -> WDels
zdel n = [(i,0) | i <- [1..n]]

*Main> f32 wdFan (zdel 3)
[(1,3),(2,3),(3,2)]
```

Wire numbers pass through unchanged. The `wd` in the name stands for wire and delay.

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.

Rather in the style of Lava (Bjesse *et al.* 1998), the Haskell functions describing networks are a way to express the *netlist* of components; such a function indicates

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

which fan components are present and how they are connected. In Lava, we would run such a function on symbolic inputs in order to produce an internal data-structure for the netlist, for later processing and analysis. We could easily do something similar here, in order to produce a data-structure very similar to that used by Hinze to describe prefix networks (Hinze 2004). (Our use of the fan structure as a building block follows Hinze.) However, what is of interest to us here is not the netlist alone but rather a version of it in which the fans are assigned a phase, and so can easily be placed precisely on a diagram such as those we have seen. To visualize the process, think of taking a netlist that is just a jumble of fans and wires, and painstakingly nailing fans, one at a time, onto the correct points and wires in a diagram. Assigning phases to fans is a not as easy as one might first assume. The assigned phases of fans later in the netlist will depend on the phases of earlier fans. In our view, a straightforward way to perform the necessary propagation of phases is to *evaluate* the fans and have them propagate delays in a way similar to the delay calculations that we have just seen. The fans must now also propagate the information that is being gathered. We introduce a data-type `Net` for this purpose (see Appendix A for further code related to this data type). It encodes information about a particular abstract wire of the network, the current phase, and the fans that have had that wire as their first input so far. Each fan is represented by a pair of its phase and the list of wires to which it fans out, starting with the current wire and having length at least two.

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

netsz :: Int -> [Net]
netsz n = [Net [] w 0 | w <-  [1..n]]
```

We also introduce an operator, `netFan`, which operates on a list of `Nets` and produces a list of nets of the same length. It corresponds to the fanout component used to build networks. It records the fanout, with its phase and wires, on its leftmost wire (by which we mean that it adds information about this fanout to the list that comes in, before outputting it). For the remaining wires, it changes the phase to the output phase of the entire fanout tile.

```
netFan [i]    = [i]
netFan (i:is) = (j:js)
  where
    ph = maximum  map phase (i:is)
    j  = Net { fans  = (ph, map wire (i:is)):fans i
             , wire  = wire i
             , phase = ph + 1
             }
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
    js = [Net { fans  = fans k
              , wire  = wire k
              , phase = ph + 1
              }  | k <- is ]
```

The definition records the fact that the outputs of a fan appear at a level one greater than the maximum of the levels of its inputs. Now we can simply run some patterns (e.g. the three shown in Figure 3), and look at the results:

```
*Main> netsz 3
[1/[],2/[],3/[]]

*Main> ser netFan (netsz 3)
[1/[(0,[1,2])],2/[(1,[2,3])],3/[]]

getNets0 f n = f netFan (netsz n)

*Main> getNets0 f31 3
[1/[(1,[1,2,3])],2/[(0,[2,3])],3/[]]

*Main> getNets0 f32 3
[1/[(2,[1,2]),(1,[1,3])],2/[(0,[2,3])],3/[]]
```

Each fanout tile is listed, on the wire from which the fanout happens, and with the phase and the numbers of all of the input (and output) wires. The number of binary operators in a fan is one less than the number of input wires. Appendix A lists some functions that take a list of nets and return size, maximum fanout and other measures. To produce the pictures of prefix networks used throughout the paper, we have written a small Haskell program that takes a list of `Nets` and produces the diagram as a .fig file, enabling easy production of many other formats.

3.3 The Sklansky network

The definition of the Sklansky network is straightforward:

```
skl :: PP a
skl _ [a] = [a]
skl f as  = init los ++ ros'
  where
    (los,ros) = (skl f las, skl f ras)
    ros'      = f (last los : ros)
    (las,ras) = splitAt (cnd2 (length as)) as

cnd2 n = n - n `div` 2 -- Ceiling of n/2
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

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> 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 wdFan (zdel 8)
[(1,1),(2,2),(3,2),(4,3),(5,3),(6,3),(7,3),(8,3)]
```

As expected, the last outputs are produced after three units of time.

We can view the nets corresponding to an instance of the Sklansky construction:

```
*Main> getNets0 skl 8
[1/[(0,[1,2])],2/[(1,[2,3,4])],3/[(0,[3,4])],4/[(2,[4,5,6,7,8])],
5/[(0,[5,6])],6/[(1,[6,7,8])],7/[(0,[7,8])],8/[]]
```

Note that the largest fanout appears on wire 4 at phase 2 and that wire 4 fans out to wires 5–8. All the outputs of this fan are produced at phase 3, which can be seen from the previous delay calculation.

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 binary operator is the function that appends two lists. The result should then be `[0]`, `[0,1]`, `[0,1,2]`, and so on. If this is the case, then the pattern (for that width) is correct for any associative binary operator. This idea is encoded in the following function:

```
check0 :: (Num a, Enum a) => PP [a] -> a -> Bool
check0 func m = func (mkFan (++)) [[a]| a <- l] == tail (inits l)
  where l = [0..m-1]

*Main> check0 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.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

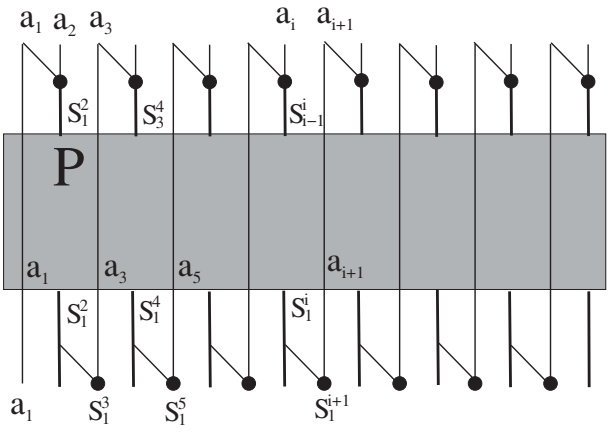


Fig. 10. The recursive decomposition used in the Brent–Kung network. The inputs to the recursive call (the shaded box P) are $S_1^2, S_3^4, \ldots, S_{i-1}^i, S_{i+1}^{i+2}, \ldots$, and the corresponding outputs are $S_1^2, S_1^4, \ldots S_1^i, S_1^{i+2}, \ldots$. The odd numbered inputs are interleaved with these values, to give $a_1, S_1^2, a_3, S_1^4, \ldots, S_1^i, a_{i+1}, S_1^{i+2}, \ldots$. The final (bottom) row of operators combines adjacent values (such as S_1^i and a_{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 are dropped.

3.5 The Brent–Kung network and variations upon it

Consider again the network shown in Figure 6. 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 10. 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. Figure 11 shows a generalization of the Brent–Kung pattern. 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 higher order function `build0` captures the important case in which each T_i is a serial prefix network. (We will consider further generalizations later.)

```
type Partition = [Int]

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

shift :: Partition -> Partition
shift (a:as) = a-1:init as ++ [last as + 1]
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

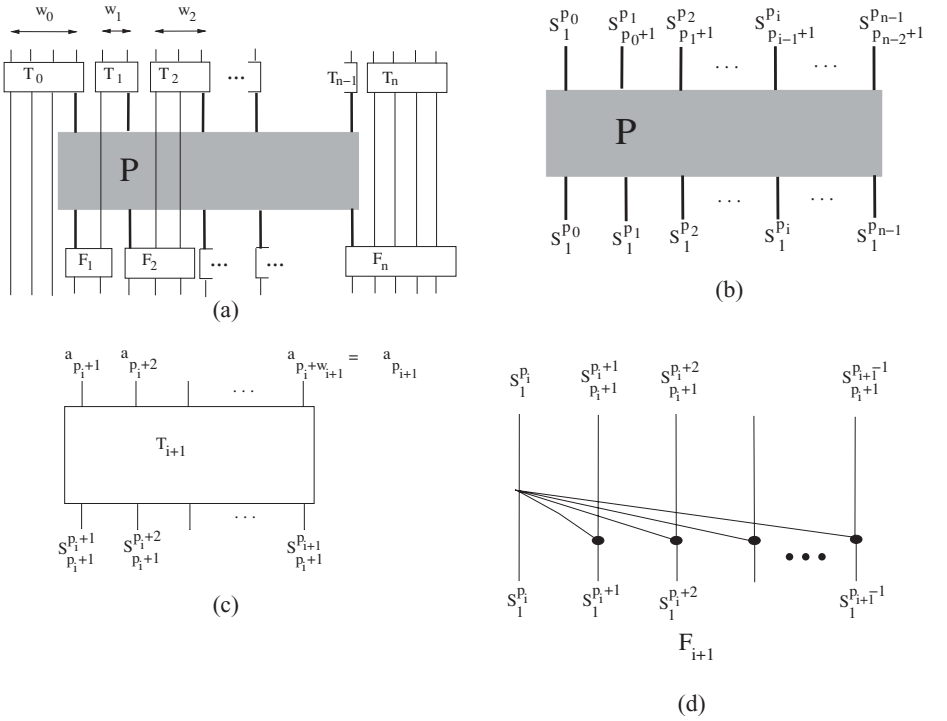


Fig. 11. (a) Generalizing the Brent–Kung pattern. The widths of the small prefix networks marked T across the top of the network in (a) can vary, and are no longer restricted to be 2. All but the leftmost prefix network (T_0) are matched by corresponding *fan* networks across the bottom (marked F). If the partition on the top is $[w_0, w_1, \ldots, w_n]$, then that across the bottom is $[w_0 - 1, w_1, \ldots, w_n + 1]$. Each T_i has width w_i. Each F_i has width w_i for $0 < i < n$. F_n has width $w_n + 1$. (b) The inputs and outputs to the recursive call marked P in (a). (c) The inputs and outputs to one of the small prefix networks across the top. The last output goes to P and the remaining outputs cross over it. (d) The shape of each fan at the bottom of the network, showing that the outputs are as required. $p_i = \Sigma_0^i w_i$, so $p_{i+1} = p_i + w_{i+1}$.

```
build0 :: Partition -> PP a -> PP a
build0 ws p f = concat . toTail (map f) . split (shift ws) .
                concat . toInit (toLasts (p f)) .
                map (ser f) . split ws
```

The three distinct phases correspond to the three lines in the definition of `build0`, and are composed using function composition. Reading from the right-hand end of the chain of functions, in the first phase, using the function `split`, the input is split according to the parameter `ws`, and the resulting sub-lists each become the input to a serial prefix network. In the second phase, `(p f)` is applied to the last elements of the outputs of all but the last of those serial networks (corresponding to the n last outputs of T_0 to T_{n-1} in the diagram). `f` is a parameter to `build0` and is a fanout tile. All other 'wires' are passed through unchanged, and the result is then concatenated back to a single list. In the third phase, `split` is once again used to

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

divide up the inputs to that phase, but this time slightly differently. If the partition in the first phase is $[w_0, w_1, \ldots, w_n]$, then that in the third is $[w_0 - 1, w_1, \ldots, w_n + 1]$. This is what is captured by the function `shift`. When the division is made, all but the first list is then input to a fan (`toTail (map f)`). (The functions `toLasts` and `toTail` are defined in Appendix B.)

Think of the recursive call as being the filling of a sandwich, with small prefix networks on one side and fans on the other corresponding to the two pieces of bread. If the sandwiched network is a prefix network, then so is the network built by `build0`. This recursive decomposition is inspired by (but not exactly the same as) one by Snir (1986). Snir's construction merges T_n and F_n into a prefix network that is composed with P. We have separated out the fan in order to make it easier to understand the constraints on the recursive call P, given a particular choice of top level partition. In this formulation, the bottom row of operators in the network now consists only of fans, so that it is easy to calculate the maximum allowed depth of the P network above the fans.

The use of the partitions parameter in `build0` 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 `build0` 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 6.4.

The rest of the paper is about investigating possible parameters to `build0` and to two further variants of it, and thus exploring the design space of prefix networks.

The following choice of parameters to `build0` gives the standard 'steps of two' Brent–Kung construction (and `bK0` was the definition used to produce both Figures 6 and 12):

```
twos :: Int -> Partition
twos n = replicate (n `div` 2) 2 ++ [n `mod` 2]

bK0 :: PP a
bK0 _ [a] = [a]
bK0 f as  = build0 (twos (length as)) bK0 f as
```

The prefix network sandwiched between 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 have width one (Figure 11). This means that that output will be produced earlier than the outputs just to its left. When the input width is 2^k for some k, this choice results in a network whose last output is produced at depth k, while the overall depth is $2k - 1$, see for example Figure 6. Such networks that produce their last outputs at minimum depth are called *restricted* in the literature (Fich 1983). (Note that we now allow some elements of the top partition to be zero. This means that those functions

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

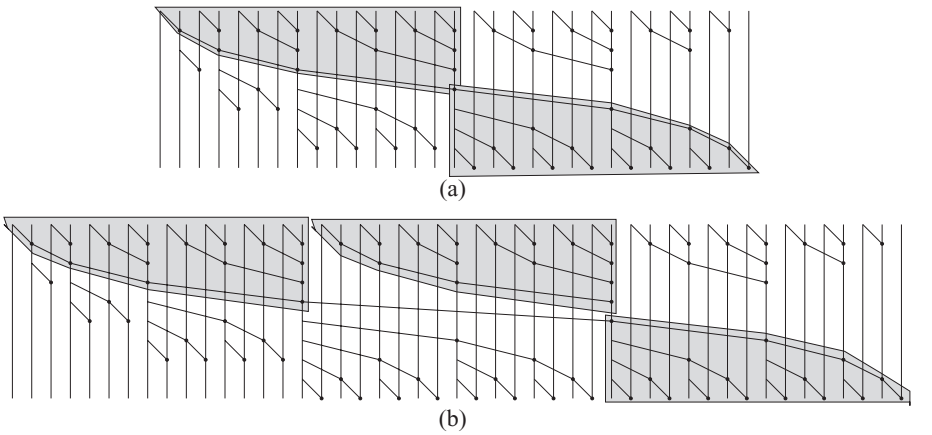


Fig. 12. The Brent–Kung-like pattern for 31 and 47 inputs, produced from the function `bK0`. For depth d, the width of this construction is $2k-1$ if d is even, and $3k-1$ if it is odd, where $k = 2^{\lfloor d/2 \rfloor}$ is the width of the binary trees (as indicated in the diagrams).

that can be mapped across partitions, and in particular `ser`, must have a case 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]`. 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 12.

3.6 The notions of depth and waist size optimality

The 31-input network shown in Figure 12(a) has depth 8 and size 52. Snir (1986) has shown that for a prefix network of width n, depth d and size s, it is the case that $d + s \geqslant 2n - 2$. 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 showed that the lower bound can be reached for depths ranging from $2\log_2 n - 2$ to $n - 1$. Serial networks are DSO; for n inputs, 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 duplication node on the leftmost input and the production of the rightmost output. For the network in Figure 12(a), the waist is 8, the same as the depth, while the 32-input Brent–Kung network shown in Figure 6 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 input widths for which all recursive calls besides the last are on odd widths, for example 23, 31 and 47, the waist and depth are equal, and then they are also DSO.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

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' :: Int -> Partition
twos' 1 = [1]
twos' 2 = [1,1]
twos' n = 2 : twos' (n-2)

bK1 :: PP a
bK1 _ [a] = [a]
bK1 f as  = build0 (twos' (length as)) bK1 f as

*Main> check0 bK1 59
True
```

Even-width inputs are now divided as `[2,2,..2,1,1]`, while the division of odd-width inputs is unchanged. For 32 inputs, the following are the resulting `Nets` for zero-delay inputs (with some additional formatting added by hand):

```
*Main> getNets0 bK1 32
[1/[(0,[1,2])],   2/[(2,[2,3]),  (1,[2,4])],
 3/[(0,[3,4])],   4/[(4,[4,5]),  (3,[4,6]),  (2,[4,8])],
 5/[(0,[5,6])],   6/[(4,[6,7]),  (1,[6,8])],
 7/[(0,[7,8])],   8/[(6,[8,9]),  (5,[8,10]), (4,[8,12]), (3,[8,16])],
 9/[(0,[9,10])], 10/[(6,[10,11]),(1,[10,12])],
11/[(0,[11,12])],12/[(6,[12,13]),(5,[12,14]),(2,[12,16])],
13/[(0,[13,14])],14/[(6,[14,15]),(1,[14,16])],
15/[(0,[15,16])],16/[(7,[16,17]),(6,[16,18]),(5,[16,20]),(4,[16,24])],
17/[(0,[17,18])],18/[(7,[18,19]),(1,[18,20])],
19/[(0,[19,20])],20/[(7,[20,21]),(6,[20,22]),(2,[20,24])],
21/[(0,[21,22])],22/[(7,[22,23]),(1,[22,24])],
23/[(0,[23,24])],24/[(7,[24,25]),(6,[24,26]),(5,[24,28])],
25/[(0,[25,26])],26/[(7,[26,27]),(1,[26,28])],
27/[(0,[27,28])],28/[(7,[28,29]),(6,[28,30])],
29/[(0,[29,30])],30/[(7,[30,31])],
31/[(8,[31,32])],32/[]]
```

The fans are all of width two. Note how each odd numbered wire has exactly one fanout on it, to the adjacent wire. This network is shown in Figure 13. This construction is due to Lin & 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 much harder question of how to deal with fanout greater than two, both in the production of DSO networks and of larger but shallower networks.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

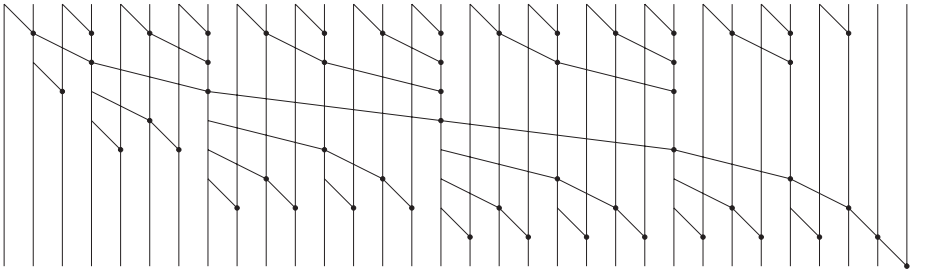


Fig. 13. A Brent–Kung-like network generated using the function `bK1`. Networks generated in this way are always both WSO and DSO. Since the waist has increased, but the network is still WSO, this 32 input, depth 9 network is smaller than the standard construction in Figure 6, 53 versus 57 operators.

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 11, with each T_i being a serial prefix network. This is the pattern that is encoded in the function `build0`; 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 wire number/input delay pairs, and a single integer representing the desired maximum depth.

```
type Context = (WDels,Int)

width :: Context -> Int
width (cin,_) = length cin

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

Think of a context as representing a hole into which a prefix network must be fitted. The first part of the context is the top edge or fringe of the hole, and is not necessarily flat, in the sense that the delay values may not all be the same. The second part corresponds to a straight line across the bottom of the diagram, representing the maximum delay at which any output should be produced. All fans must fit between these two edges. (It would also be possible to use a list of wire-delay pairs to indicate maximum depth per output, but we have not found that necessary for our purposes.) The difference between the output delay and the first element of

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

the input delays is a measure of the depth (or waist) available to the network, and is computed by the function `maxd`.

To check if a proposed network fits in a context, we can run it with the top fringe (the `WDels` part) of the context as input and with `wdFan` as the fan component, and compare the delays of the outputs with the required maximum output delay. We can also extend the correctness checking function with a check that the network fits in its context.

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

check :: WPP -> Bool
check (WPat ctx func) = check0 func m && fits func ctx
  where l = [1..m]
        m = width ctx
```

A network pattern generated for a given context is wrapped together with the context in which it was created.[1]

```
type APP = forall a. PP a

data WPP = WPat Context APP

getContext :: WPP -> Context
getContext (WPat c p) = c
```

This could be used to avoid trying to use a pattern in an inappropriate context. In addition, it allows networks to be passed around and run on different types (which is necessary because of our use of NSI) without leading to problems because lambda binding is monomorphic. The type of `fits` above is now `APP -> Context -> Bool`.

4.1 Measure functions

Next, we need *measure functions*, which should take a wrapped pattern and return an element of the `Ord` type. Examples are the functions `size`, which measures the number of binary operators in a network and `maxfo`, which returns the maximum fanout:

```
size :: WPP -> Int
size = sizeN . getNetsW

maxfo :: WPP -> Int
maxfo = maxfoN . getNetsW
```

Here, we calculate the nets, taking account of the context, and then take the size or maximum fanout (see Appendix A for the definitions of `sizeN` and `maxfoN`, and Appendix B for the definition of `getNetsW`). These particular measure functions

[1] Because we now have a forall type inside a constructor, we are now using Rank 2 types. Placing {-# LANGUAGE Rank2Types #-} at the beginning of ones code file enables the use of Rank 2 types in Haskell.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

could be calculated more simply, using NSI or a simpler data structure, but we find it convenient to use `Nets`, as we also use it for drawing network diagrams.

We have used wire numbers in the `Nets` datatype, in order to be able to capture some information about wire length. When prefix networks of the types discussed here are implemented as circuits, it is the effects of the *horizontal* wires that are most dominant (as for a given width and depth, the vertical wires that appear one for each input (and connect input i to output i) will be of the same length in each case). Each fan in fact results in a single horizontal wire that joins each of the operators fed by the wire that is fanned out. The lengths of these wires can vary greatly in different network topologies, and a good first rule of thumb is to try to keep those wires as short as possible. The measure function `sumspan` sums the spans of each fanout in a network to give a measure of total horizontal wire length.

It is easy to combine measure functions. Two combinations that we use very often are `sizefo`, which minimizes first size and then maximum fanout, and `fosize`, which minimizes first fanout and then size.

```
sizefo :: WPP -> (Int,Int)
sizefo wp = (sizeN ns, maxfoN ns)
  where ns = getNetsW wp

fosize :: WPP -> (Int,Int)
fosize wp = (maxfoN ns, sizeN ns)
  where ns = getNetsW wp
```

The function `try` checks whether or not a network pattern `p` fits in a context, returning either `Nothing` or the pattern wrapped in that context:

```
try :: APP -> Context -> Maybe WPP
try p ctx | fits p ctx = Just (WPat ctx p)
          | otherwise  = Nothing
```

Now, we have the programming building blocks for a search. In Section 3.5, we introduced network descriptions based around the top partition $[w_0, w_1, \ldots, w_n]$. Now, we will explore the effect of varying that partition, moving away from the restricted forms seen so far. In particular, we will explore the effect of allowing fanout to be greater than 2. Given a context, we will consider 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.

4.2 Designing the search space: choosing partitions

What partitions should we explore for a given context, bearing in mind that considering all integer partitions of the width of the context 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`. This restriction is introduced both because large fanouts are best avoided in VLSI circuits and because it conveniently reduces the search space. The last element of the top partition should then be at most `f-1`, so that the matching fan is at most

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

f; elements other than this one and the first should be at most f. In addition, the overall depth of the context limits the possible depth (and hence width) of each small prefix network. Note that the first element of the top partition, w_0, does not give rise to a matching fan, so it could be larger than f without causing increased fanout in the network. The function parts0 captures these constraints on the partitions:

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

parts0 :: Int -> Context -> [Partition]
parts0 f ctx = [l:rs ++ [r]|
                l <- [2..d], r <- [1..rr], rs <- perms 2 ff (n-r-l)]
  where
    n  = width ctx
    d  = maxd ctx
    rr = min (f-1) d
    ff = min f d
```

It is tempting, at this point, to tweak the partition generation by adding further constraints based on what we know about the network shape. We have found that it is better to avoid such tweaks as they will anyway be dominated by later, more sweeping changes aimed at reducing the search space. For now, let us take a quick look at the scale of the problem:

```
*Main> length (parts0 4 (zdel 16,5))
277
*Main> length (parts0 4 (zdel 32,6))
136523
*Main> length (parts0 4 (zdel 45,7))
20730338
```

Next, we make a version of the build0 pattern that does the necessary wrapping and unwrapping, but is otherwise unchanged.

```
buildW0 :: Context -> Partition  -> WPP -> WPP
buildW0 ctx ws (WPat _ p) = WPat ctx (build0 ws p)
```

As performance is an issue, and because we have not run into problems with trying to compose non-matching sub-networks, we have chosen not to check compatibility of the outside context with the partition or the context of the wrapped network that is the input. However, these checks could be performed should the need arise.

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

```
dso :: (Ord a) => Int -> (WPP -> a) -> Context -> WPP
dso f mf ctx = fromJust (prefix ctx)
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
  where
    prefix :: Context -> Maybe WPP
    prefix ctx = ...
```

The `fromJust` function raises an exception if `prefix` returns `Nothing` and returns the solution otherwise. The first cases in the function `prefix` are easy:

```
prefix ctx | width ctx == 1           = try wire ctx
prefix ctx | 2^(maxd ctx) < width ctx = Nothing
prefix ctx | fits ser ctx             = Just (WPat ctx ser)

wire :: APP
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). If the width of the context is greater than 2^d for depth d, then there is no solution and `Nothing` is returned. In the next case, a serial network is returned if it fits in the context. If the context is deeper than that required for the serial network, a serial network is returned in any case.

For the step, we would like, if possible, to make a suitable network for each of the candidate partitions, that is each element of `parts0 f ctx`. So we define a function called `makeNet` that, given a partition, returns either `Nothing` or a (wrapped) network in which the function `prefix` has once again been used to deal with the recursive call in the middle of the sandwich. Using `mapMaybe` to map `makeNet` across the partitions results in a (possibly empty) list of possible networks for that context, and `bestOn mf` chooses the best of these according to the measure function `mf`. The third case for `prefix` is

```
prefix ctx@(is,o) = bestOn mf $ mapMaybe makeNet (parts0 f ctx)
  where
    makeNet ws = ...
```

and `bestOn` is defined as follows:

```
bestOn :: (Ord a) => (WPP -> a) -> [WPP] -> Maybe WPP
bestOn _  [] = Nothing
bestOn mf as = Just (minimumBy (compareOn mf) as)

compareOn :: (Ord a) => (WPP -> a) -> WPP -> WPP -> Ordering
compareOn f c1 c2 = compare (f c1) (f c2)
```

Finally, `makeNet` is defined in the `where` clause as

```
makeNet ws = do let js = map (last.(ser wdFan)) $ split ws is
                q <- try wire ([last js],o-1)
                p <- prefix (init js,o-1)
                return $ buildW0 ctx ds p
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

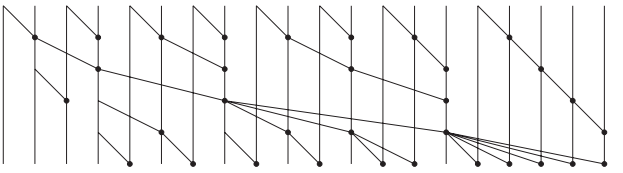


Fig. 14. A DSO network of width 20 found by the search embodied in the function `dso`.

This code calculates the context of the sandwiched recursive call of `prefix`, checking as well that the last output of the rightmost serial network also meets its timing constraint (the call of `try wire`). (The last output of a serial network is always produced last; if the last output fits in the hole that is the context, then so will all other outputs.) The function `prefix` is called on the calculated context, and if a network is successfully returned, it is sandwiched (using `buildW0`) to produce the successful result for the partition being considered. If either of the calls to `try wire` or `prefix` fails (that is produces `Nothing`), then the call of `makeNet` fails for the given partition. (For readers unfamiliar with Haskell, we are using the `Maybe monad` to ease the expression and composition of computations that can fail. Wadler's paper provides a good introduction (Wadler 1992) and there are many more recent tutorials about monads available on the web.)

This very simple approach works well for small examples. For example, drawing the result of calling the function `dso 6 (zdel 20,5)` gives the network shown in Figure 14.

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).

4.3 Refining the search space

Let us now begin the process of adjusting the partition generation to consider fewer useless partitions. First, note that if a network is sufficiently deep, then fanout 2 will do. Earlier, we saw the `bK0` pattern, which, for depth d, could reach width $2k-1$ for even d and $3k-1$ for odd d, with $k = 2^{\lfloor d/2 \rfloor}$, see Figure 12. For a given depth, this width gives us the limit for the construction of a fanout 2 network. It is easy to calculate these limits, given the depth permitted by the context, and to generate a partition that results in fanout 2 when the network width is small enough, as specified by the first equation in `parts1`:

```
parts1 :: Int -> Context -> [Partition]
parts1 f ctx
  | k2 > n  = [l : twos1 (n-l-1) ++ [1] | l <- [2..d]]
  | k2 == n = [replicate (fnd2 n) 2]
  | k2 < n  = [replicate k 2 ++ rs ++ [r]|
               r <- reverse [1..maxr], rs <- perms 2 ff (n-r-2*k)]
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

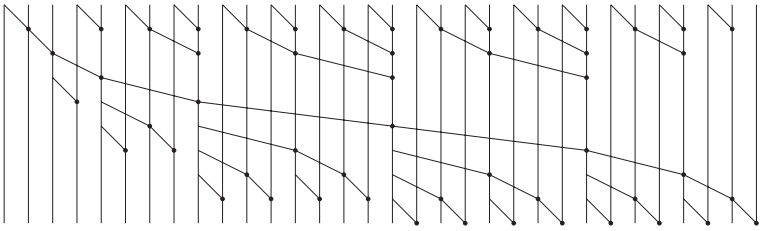


Fig. 15. Fanout two network in which all partitions stem from the first equation of `parts1`. The top partition is `[3,2,2,2,2,2,2,2,2,2,2,2,2,2,1]`. The next (for the recursive call that applies to the last elements of all but the last (width one) serial network) is `[2,2,2,2,2,2,1]`.

```
  where
    maxr = min (f-1) d
    ff = min f d
    n  = width ctx
    d  = maxd ctx
    k  = 2^(fnd2 d)
    k2 = if even d then 2*k else 3*k

twos1 n = replicate (fnd2 n) 2 ++ [1 | odd n]

fnd2 n = n `div` 2     -- Floor of n/2
```

The partitions generated by the first equation start with a number in the range `2` to `d`; the resulting serial network will not give rise to a matching fan. For larger widths of this initial serial network, deeper networks will result, as the wider serial networks use up more of the available waist. The final element of the partition must be `1`, giving matching fanout 2. The definition of `twos1` permits the partition ending in two ones that we saw earlier. Figure 15 shows a 32 input depth 9 network in which all partitions are generated using this first equation of `parts1`. The case where fanouts greater than 2 are needed is covered by the second and third equations. Here, we begin the partition with a sequence of `k` 2s, as we assume that the left-hand part of the network will look look like one of the arrangements shown in Figure 16. Separating the case for when the input width is exactly twice `k` avoids having to place only `k-1` 2s before the call to `perms` in the first case. The reduction in the number of partitions that must be considered for a given width and depth is considerable.

```
*Main> length (parts1 4 (zdel 16,5))
6
*Main> length (parts1 4 (zdel 32,6))
241
*Main> length (parts1 4 (zdel 45,7))
34729
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

Fig. 16. Even when a network uses fanout greater than two, it will have a left-hand part that uses only fanout two, or possibly one fanout of three at the waist. The arrangement on the left (which shows only part of the network working on an initial subsequence of the inputs) shows how the resultant binary trees of operators are arranged for odd depth. For even depth, we squeeze one of the trees up against the waist. In either case, the top partition will contain at least k 2s, where $k = 2^{\lfloor d/2 \rfloor}$, for depth d. The concrete trees shown here are of depth 3, but a similar pattern applies for deeper trees.

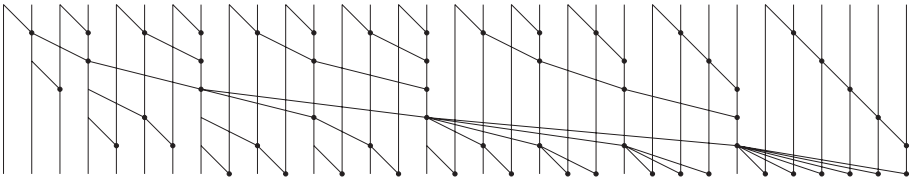


Fig. 17. A depth size optimal network of width 33, depth 6 and with maximum fanout 7.

This allows us to reach wider examples, for example the network shown in Figure 17, which is identical to the LYD construction for 33 inputs (Lakshmivarahan *et al.* 1987).

From our earlier analysis of the DSO network construction problem (Sheeran & Parberry 2006) and a search of the literature, we have concluded that this is the widest known minimum depth DSO network. The question of whether or not wider minimum depth DSO prefix networks can be constructed remains open.

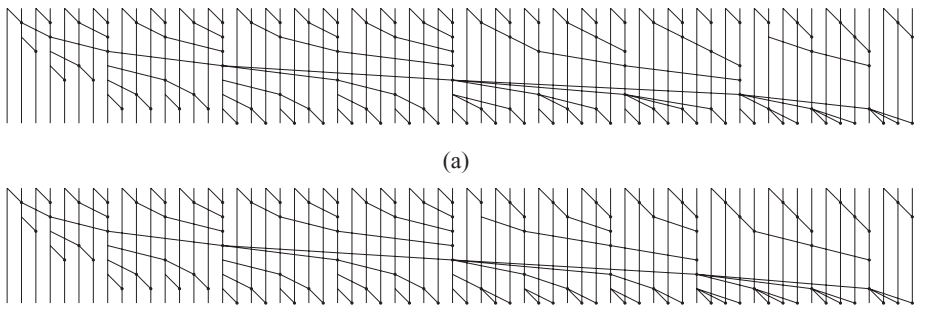
4.4 A drastic refinement of the search space

The next step is to consider vastly fewer permutations, by modifying the `perms` function to generate only sorted lists. The call of `perms` in `parts1` is replaced by one of `permsUp`. The numbers of generated partitions are again greatly reduced.

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

*Main> length (parts1 4 (zdel 32,6))
20
*Main> length (parts1 4 (zdel 45,7))
64
*Main> length (parts1 4 (zdel 64,8))
72
```

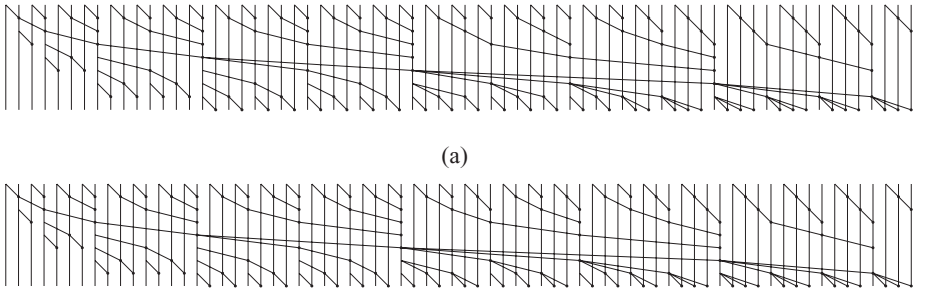
https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press



(a)

(b)

Fig. 18. Two different DSO networks of width 64, depth 8 and fanout 4. Each has size 118. The measure function used in the search were (a) number of operators (`size`) and (b) sum of horizontal wire spans (`sumspan`) (see Section 4.1). Note how the left hand part of each network has the arrangement illustrated on the right of Figure 16



(a)

(b)

Fig. 19. Considering only sorted partitions is indeed a restriction. Above is the widest DSO network found using sorted partitions for fanout 4 and depth 8. It has width 70, 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. The answer is below, and it can be generated by specifying the top partition explicitly. Alternatively, increasing the fanout to 5 allows the search to find a DSO network of depth 8 and width 72, with sorted top partition.

With so few partitions to choose from, larger examples come into reach, see for example the width 64, depth 8, fanout 4 networks shown in Figure 18.

Considering only sorted partitions is, however, a major restriction in that it can cause us, in a few cases, to miss attractive solutions (Figure 19). The function `maxdso` given in Appendix B encodes what we know from our own earlier work on DSO networks. For example, `maxdso 4 8` is 72, which indicates that current known constructions for DSO networks reach width 72 for fanout 4 and depth 8. Playing with this function and comparing with our generated networks tells us that we sometimes miss a few DSO networks close to the width limits for a given fanout and depth. In those cases, one can increase the fanout to ensure that the required width is well away from the limit for the given depth and the new fanout,

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

so that the search based method can still find a DSO solution (though perhaps with fanout higher than absolutely necessary). A second option is to examine the longest sorted partition achieved, and to manually specify the partition that results when it is adjusted to reach greater width (and is then unsorted). The required partition can be explicitly specified in the partition generation function:

```
parts1 f ctx
  | n==72  = [replicate 16 2 ++ [2,2,3,2,2,3,2,2,3,4,4,4,4,3]]
  | k2 > n = ...
```

For wider and deeper examples, it may be necessary to use unsorted partitions, even in the recursive calls. In that case, one can change the call of `permsUp` to one that uses `perms` to form an initial part of the partition, and `permsUp` to form the rest. The use of sorted partitions is necessary if the search is to get beyond 40 or so inputs.

Measure functions allow the user to customize her DSO networks, as illustrated in Figure 18. For a given width and depth, though, *all* DSO networks, no matter what the partitions used in their generation, will have the same size. So choosing `size` as the measure function, as we did in generating Figure 18(a) actually amounts to finding the first working solution in the search. In cases where we care only about finding DSO networks quickly, as for example when we are simply trying to break exisitng records, we can replace the choice of the best option according to the measure function (the code `bestOn mf` in the `dso` function definition) with just taking the first working (non-`Nothing`) option, corresponding to the Haskell function `listToMaybe`. Because we are in a lazy language, computations that are no longer necessary to produce the result simply do not take place. Once this change is made, the order in which one examines potential solutions becomes much more important. This is the reason why the first case of `parts1` contains a reversed list; we want to consider partitions whose last element is longer first, as we typically need long last elements when trying to produce wide networks for a given fanout and depth. This approach, by-passing measure functions, was used to produce some of the wider DSO networks that we report in Section 6.1. Further discussion of the results for DSO network generation is postponed to that section.

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 1982, 1983). Networks that are shallow, but as close as we can get to DSO are more likely to be of practical interest than DSO networks. Shallow networks, in general, run faster when made into circuits, while keeping the network small keeps the power consumption down. 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 generalise. It turns out that what is needed is a generalization of the Ladner–Fischer construction (Ladner & Fischer 1980).

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

5 Searching for shallow parallel prefix networks

5.1 The Ladner–Fischer construction in Haskell

Ladner and Fischer (1980) is a wonderful paper that introduces a family of prefix networks. The authors introduce an additional parameter, the *slack*, to indicate by how much the network is allowed to exceed the minimum depth. The construction described in the paper 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 -> PP a
ladF _ _ [a]  = [a]
```

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

```
ladF 0 f as  = init los ++ ros'
  where
    (los,ros) = (ladF 1 f las, ladF 0 f ras)
    ros'      = f (last los : ros)
    (las,ras) = splitAt (cnd2 (length as)) 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 (Figure 7). 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):

```
ladF n f as  = build0 (lp (length as)) (ladF (n-1)) f as
  where lp 1 = [1,0]
        lp 2 = [2,0]
        lp n = 2 : lp (n-2)
```

Figure 7 shows two width 32 networks, for slacks 0 and 1. In the slack 0 network, the slack 1 recursive call on the left is marked with a dotted box.

Observing the recursive structure of the network description, 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 (cnd2 n) + ladSize 0 (fnd2 n) + fnd2 n
ladSize k n | even n = ladSize (k-1) (cnd2 n) + n-1
ladSize k n | odd n  = ladSize (k-1) (cnd2 n) + n-2
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

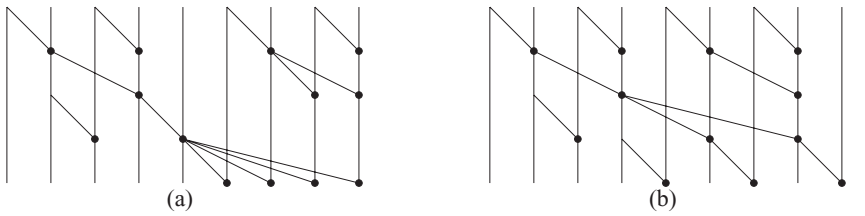


Fig. 20. (a) LF_0, the minimum depth Ladner–Fischer network, for nine inputs; it has 13 operators and depth 4 (b) A smaller 9-input network (with 12 operators, depth 4) made from LF_18, and one further operator to take care of the last input. This network is DSO. This illustrates the fact that LF_0 does not always give optimal networks.

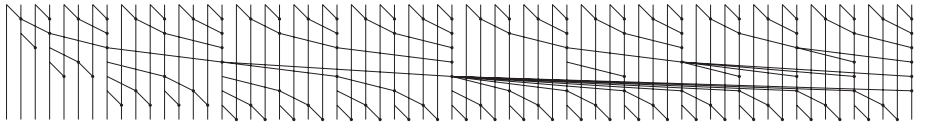


Fig. 21. The Ladner–Fischer construction for slack 2, width 64. It has size 125 and maximum fanout 18. Note how it produces its last output at minimum depth. Compare with the fanout 4 DSO network in Figure 18.

```
*Main> ladSize 0 64
168
*Main> ladSize 2 64
125
```

Similarly, one can calculate the maximum fanout (in the sense used here) in a Ladner–Fischer network of a given width and slack (see functions `maxlfo` and `maxlfo'` in Appendix B). The networks produced by the `ladF` function correspond exactly in size and fanout to those predicted by these calculations.

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 20. 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 eight inputs. The result is a smaller prefix network than that on the left. Another example in which Ladner–Fischer makes poor choices is shown in Figure 21.

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.

5.2 Improving on the standard Ladner–Fischer construction using search

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 when searching for DSO networks. To encode a new generalization of the Ladner–Fischer pattern,

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

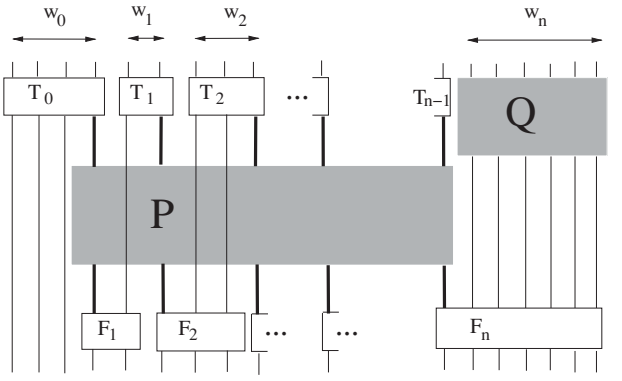


Fig. 22. The recursive parallel prefix network construction captured by the `build1` function. There are now two recursive calls rather than the one that we have had so far. The second one is marked Q and corresponds to the `q` parameter.

one needs *two* recursive calls, one sandwiched between the serial networks and fans as before, and one at the right-hand end of the top row of small prefix networks. That is, we want the network T_n at the top right of the generalised Brent–Kung pattern shown in Figure 11 to be a recursive call, rather than just a serial network, as illustrated in Figure 22. Earlier, we used the functions `build0` and `buildW0` to encode the generalised Brent–Kung pattern. Similarly, `build1` and `buildW1` encode the more general pattern (Appendix B).

When considering how to build prefix networks using the construction shown in Figure 22, we could introduce a new integer parameter constraining the width, w_n, of the new recursive call, and thus of the associated fanout F_n. However, after some experiment with this, we decided to omit that parameter, and to *calculate* the maximum allowed width of the upper recursive call, using the maximum fanout of the Ladner–Fischer construction of the same width and depth as the limit. The advantage of this approach is that it calculates successively smaller limits in recursive calls during generation, while a single externally supplied parameter would likely be much too large in recursive calls. (Should users prefer to manually control the limit, however, one could easily add an additional parameter to the generation function.) Our choice means that the interface of the new network generation function is the same as before. Indeed, a lot of the function remains unchanged:

```
gen :: (Ord a) => Int -> (WPP -> a) -> Context -> WPP
gen f mf ctx = fromJust (prefix ctx)
  where
    prefix = memo pm
    pm ctx | width ctx == 1 = try wire ctx
    pm ctx | 2^(maxd ctx)< width ctx = Nothing
    pm ctx | fits ser ctx = Just (WPat ctx ser)
    pm ctx@(is,o)  = bestOn mf $ mapMaybe makeNet (parts2 f g ctx)
      where
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
makeNet ds
  = do let sis = split ds is
       let js = map (last.(ser wdFan)) $ init sis
       p  <- prefix (js,o-1)
       pr <- prefix (last sis,o-1)
       return $ buildW1 ctx ds p pr
d = maxd ctx
w = width ctx
mind = alog2 w
slack = d - mind
g = maxlfo slack w
```

We have taken the opportunity to memoize the `prefix` function. 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. We have found that memoization can pay off for larger input widths. The purely functional memo function used was provided by Koen Claessen and is available to the reader at URL http://www.cse.chalmers.se/~ms/PPSearch/; it is a refinement of Hinze's approach to the construction of memo functions (Hinze 2000). Any other memo function could be substituted for this one.

The cases considered in the new network generation function `gen` are the same as in the earlier `dso` function (width one context, hopeless context, and room for a serial network being the base cases). What must change are the definition of the `makeNet` function, which generates a network for a given partition, and the partition generation function. The `makeNet` function now has two recursive calls of `prefix`, and it must calculate the new context for each of them.

The new partition generation function `parts2` is listed in Appendix B. It is modelled closely on the earlier function `parts1`. It has a new parameter `g` giving the maximum size of the last element of any generated partition. It differs from the earlier function only in how it calculates a suitable range of values for the last element of the partition, in the case when fanouts are used.

How good are the results? First, the call `gen 2 sizefo (replicate 9 0,4)` does indeed produce the DSO network shown in Figure 20(b). Comparing the new networks to Ladner–Fischer, two separate generalisations have been made: the search permits the small prefix networks across the top to be wider than two, and is also choosing how wide to make the new recursive call, and so making better choices than those hard-wired into the Ladner–Fischer construction. Both lead to improvements, so let us explore their effects. For 64 inputs, depth 7, restricting `f` to 2, and using measure function `sizefo` gives the network shown in Figure 23, with 128 operators, fanout 13, while minimizing fanout first and then size gives fanout 12, size 129. 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 size 126 with fanout 9. This network is shown in Figure 24. Allowing small fanouts of 4 reduces the size to 125, but increases the fanout to 10.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

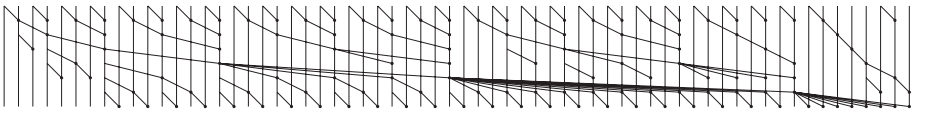


Fig. 23. The prefix network of width 64, depth 7 found using dynamic programming and measure function `sizefo`, with the widths of the small serial prefix networks across the top still restricted to 2. It has size 128 and fanout 13. This is already a considerable improvement on the classic LF construction.

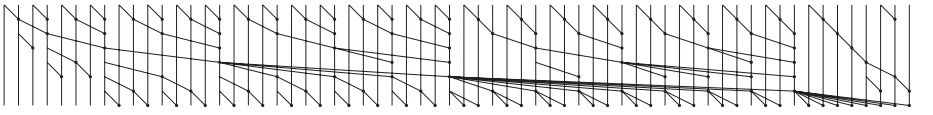


Fig. 24. The prefix network of width 64, depth 7 found using dynamic programming, with the widths of the small prefix networks across the top now allowed to be 3. This network results from both measure functions `sizefo` and `fosize`. It gives further improvement on the classic LF construction, as it has size 126 and fanout 9. Classic LF gives size 137 and fanout 17.

To compare the results generated by the `gen` function and the Ladner–Fischer construction for a wider range of depths and sizes, it is convenient to write small Haskell functions to generate the required information, see the functions `info` and `res` in Appendix B. The call `res 0 3 sizefo 16 2 9` considers minimum depth networks produced by `gen` with small fanout limited to 3 in the range of widths 32–144 at intervals of 16, and gives

```
[(32,5,(74,17),(74,17)),(48,6,(102,18),(117,25)),
 (64,6,(168,33),(168,33)),(80,7,(173,21),(211,41)),
 (96,7,(223,35),(262,49)),(112,7,(286,50),(313,57)),
 (128,7,(369,65),(369,65)),(144,8,(322,26),(412,73))]
```

Each element of the list gives width, depth and the results for `gen` and Ladner–Fischer, in this case a pair of size and fanout. For input width a power of two, the results for `gen` are identical to Ladner–Fischer. We have noted that the results are also identical for input width one less than a power of two. For other input widths, search gives results that are smaller and have lower fanout than Ladner–Fischer, with the differences being largest just above a power of two, and reducing as one approaches the next power of two, at which the results are identical again.

Interestingly, increasing the small fanout to 4 gives very little improvement, with a difference in size and fanout between this and the fanout 3 case first observed at width 96 in these samples.

```
[(32,5,(74,17),(74,17)),(48,6,(102,18),(117,25)),
 (64,6,(168,33),(168,33)),(80,7,(173,21),(211,41)),
 (96,7,(223,34),(262,49)),(112,7,(286,50),(313,57)),
 (128,7,(369,65),(369,65)),(144,8,(321,28),(412,73))]
```

The above calculation is done using the memoised version of `gen`. It takes just over 30 seconds of processing and 8 seconds of garbage collection on one core of a Dell M1330 laptop with an Intel Core2 Duo 2.2 GHz CPU T7500 and 3.5 GB of RAM. Without memoization, the calculation takes approximately 430 seconds.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

Increasing the slack to one gives results that improve on Ladner–Fischer for all input widths that we can reach, including powers of two. Further details are given in Section 6.

5.3 A new construction: replacing small serial networks by Ladner–Fischer

The degree of improvement that we have achieved over classic Ladner–Fischer is indeed encouraging. 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 restricted networks that produce that last output at minimum depth, but that save on size by being deeper for other outputs. We already have such networks in the form of classic Ladner–Fischer. We define a variant of Ladner–Fischer as follows:

```
adLadF :: PP a
adLadF f as = ladF (ln2 (length as) - 2) f 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. The new definition `gen1` is very similar to `gen`, with one occurrence of `ser` replaced by `adLadF`. New versions of the functions `build1` and `buildW1`, now called `build2` and `buildW2`, are also needed (Appendix B).

Now, in the search for networks that are not minimum depth, 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, we can 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 (Figure 25).

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 and width a power of two. Here, we have concrete examples supporting the assertion that Ladner–Fischer is not optimal in these cases. 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

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

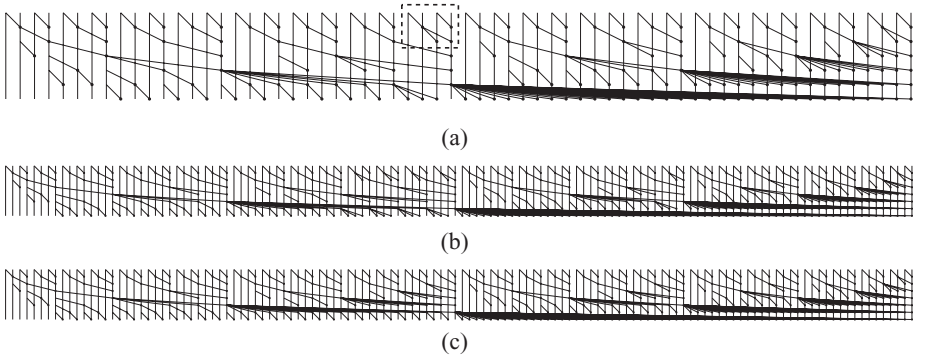


Fig. 25. (a) The 64 input, depth 6, network obtained by allowing small Ladner–Fischer networks to replace small serial networks. The four input LF network that leads to a saving of one operator over classic Ladner–Fischer (Figure 7(a)) is marked with a dotted box. (b) The 128 input, depth 7, network 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.

huge network diagrams that are much too large 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 -> [Partition]
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 25), 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,..,2,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,8,16,256]`. Now, one can try reversing the order in which the possible partitions are considered, by replacing the call of the generator `(parts3 f g ctx)` by `(reverse (parts3 f g ctx))` in the function `gen1`. Then, the pattern for 256 inputs contains eight 4s and two 8s, while that for 512 inputs has 12 4s, 4 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 small Ladner–Fischer networks are used along the top, and when they are not. These division points correspond to one half, one quarter, one eight,

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

and so on of the overall network width. After some experimenting with patterns, we find that a working sequence has 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), for input widths 2^6, 2^7 and upwards. The function `pat`, given in Appendix B, generates the pattern that is appropriate for 2^k inputs, for example:

```
*Main> pat 8
[4,4,4,4,4,4,4,4,8,8]
```

Now, for the outer partition of network with 2^k inputs, it is easy to make up the entire partition, as it is just `(pat k) ++ [2\^(k-1)]`, with an appropriate number of twos appended on the left.

Things become more interesting in recursive calls in which the length of the input is not exactly a power of two. This can happen in the sandwiched recursive call just inside the top partition. If the left-hand half of the top partition were all 2s, then this recursive call would work on a power of two inputs, but this is not the case because of the patterns of 4, 8 and larger input prefix networks across the top. The recursive call has fewer inputs, and we must find a way to divide those inputs in the right place, so that the large fan for the next partition appears in exactly the right place. For example, for 256 inputs, the top level partition is 40 2s, followed by $[4, 4, 4, 4, 4, 4, 4, 4, 8, 8, 128]$. This means that the sandwiched network has only 50 inputs, rather than 64. Those inputs are at wires $[2, 4, 6, \ldots, 80, 84, 88, \ldots, 120, 128]$ and the last element of its partition should span those inputs that have wire number greater than 64, which is the wire on which the big fan should appear. Calculations about where to put the fans are enabled by numbering each wire at the top level, and passing those numbers into recursive calls. Knowing which of the original input wires the recursive call operates upon enables the correct placement of the fanouts. We write a new partition generation function that given a list of wire numbers produces a single partition, starting with some twos (introduced by the `fill` function, see Appendix B), then the pattern and finally a single number at the correct half-way division point, corresponding to the second half of the input.

```
partf :: [Int] -> Partition
partf is = fill lis (pat (alog2 lis) ++ [r])
  where
    (lis,his) = (length is,head is)
    mid = (last is + his - 1) `div` 2
    r   = length [ k |k <- is, k > mid]

*Main> partf [1..256]
[2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,4,4,4,4,4,4,4,4,8,8,128]

*Main> map last $ init (split (partf [1..256]) [1..256])
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48
,50,52,54,56,58,60,62,64,66,68,70,72,74,76,78,80
,84,88,92,96,100,104,108,112,120,128]
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
*Main> partf it
[2,2,2,2,2,2,2,2,2,2,2,2,4,4,18]
```

Thus, we manage to incorporate our two new insights: the pattern of use of small Ladner–Fischer networks and the placement of the large fans. The result is the following definition of a new parallel prefix construction:

```
ppf :: Int -> PP a
ppf k = pp [1..2^k]

pp :: [Int] -> PP a
pp [_]   = wire
pp [_,_] = ser
pp is    = build2 ss (pp js) (pp (last sis))
  where
    ss  = partf is
    sis = split ss is
    js  = map last \$ init sis

*Main> check0 (ppf 10) (2^10)
True
```

The first parameter to `pp` is the list giving wire numbers. At the outer level, this is the list containing the numbers 1 to 2^k. The two base cases of function `pp` introduce a single wire or a two-input serial network containing a single operator. For the step, `partf` generates a partition `ss` for the given input list of wires `is`. From this, the wires that must be input to the two recursive calls of `pp` are calculated, and `build2` constructs the final network using the `ss` partition. Figure 26 shows the new construction for 256 inputs, generated by the function call `ppf 8`.

For completeness, Appendix B contains the function `ppsize` that characterizes the size of our new construction, following the same recursive pattern (and matching exactly the results generated from the `ppf` function above). This is provided for readers who may be aiming to produce smaller networks of width a power of two and minimum depth.

We have seen a number of solutions to different parallel prefix network design problems. In the following sections, we summarize the results first from the point of view of parallel prefix network design and then from a functional programming perspective.

6 Results in parallel prefix network generation

With the help of simple functional programming techniques, we have been able to solve some open problems in prefix network design.

6.1 Search based generation of Depth Size Optimal Networks

We have shown how to generate Depth Size Optimal networks for a given input width and required depth, while retaining control of fanout. The user may also

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

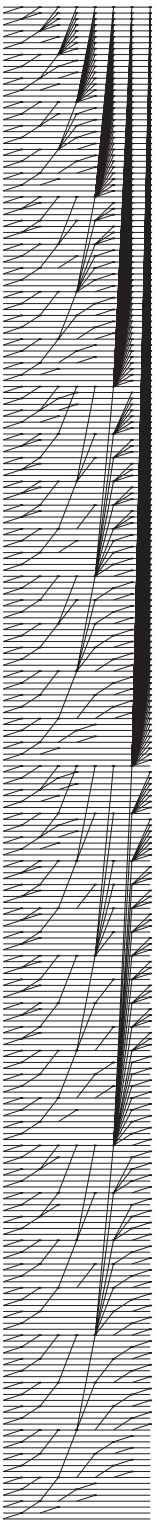


Fig. 26. Our final construction, generated by the function `ppf 8`, for 256 inputs. It has 773 operators, an improvement on the 792 of the classic Ladner–Fischer construction.

.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

Table 1. *Maximum width DSO network generated using search and manual extension of sorted partitions, for a selection of depths and fanouts*

Depth	6	7	8	9	10
Fanout					
3	24	38	58	88	136
4	29	46	72	114	179
5	31	50	80	128	203
6	32	52	84	135	217

Table 2. *Comparing widths of the Lin and Su (2005) construction SU4 with those generated using search and manual extension of partitions. In both cases, fanout is limited to* 4

Depth	SU4	Sheeran
7	30–46	30–46
8	47–72	47–72
9	73–114	73–114
10	115–165	115–179
11	166–250	180–281

control other aspects of the resulting networks using the measure functions. This has not been achieved before.

The following table lists the widest DSO networks that we have generated using the `dso` function for selected depths and maximum fanout. A measure function was not used (as we want to minimize only size and all networks generated by the `dso` function for a given context have the same size). The method can also generate narrower networks for the same widths and fanouts. When close to these limits, manual editing of the top partition, guided by the widest network with a sorted top partition was used (as described in the width 72, fanout 4, depth 8 example in Section 4.4). These networks are, in all cases, the widest known DSO networks for the given fanout and depth, see our earlier technical report (Sheeran & Parberry 2006). What is lacking, though, is a suitable theory that either proves that this is as well as we can do, or indicates that there is further scope for improvement. Such a theory would be of considerable interest, as it would need to take fanout into account in a way that earlier theories (such as Snir's) have not done.

The parallel prefix literature studies fanout four prefix networks in particular, as these are particularly suitable for VLSI implementation. Our approach allows us to improve on some of the best results from this form of manual network construction for medium width networks, as shown in Table 2. The improvements for depths 10 and 11 are substantial. The manual intervention to extend partitions is surprisingly easy and effective, but it is slightly unsatisfactory because it interferes with the use of measure functions, replacing decisions guided by them by the user's choices, at least for the outermost partition. Right at the limit, there seems to be only one choice of partition, but for narrower networks, it would be good to be able to consider a variety of choices and to choose using the measure function, completely

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

Table 3. *Size/fanout for generated networks (with small fanout 3, slack 0) and classic LF*

Width		32	48	64	80	96	112	128	144
Depth		5	6	6	7	7	7	7	8
gen	size	74	102	168	173	223	286	369	322
	fo	17	18	33	21	35	50	65	26
LF	size	74	117	168	211	262	313	369	412
	fo	17	25	33	41	49	57	65	73

Table 4. *Size/fanout for generated networks and classic LF, with slack 1, that is depth one greater than minimum. sfo stands for small fanout*

Width		32	48	64	80	96	112	128	144	160
Depth		6	7	7 8	8	8	8	9	9	
gen	size	56	87	126	150	186	226	273	281	318
	fo	6	5	9	6	9	13	19	10	11
	sfo	4	4	4	5	5	4	4	4	4
LF	size	62	97	137	172	212	252	295	330	376
	fo	9	13	17	21	25	29	33	37	41

automatically. Automating this step would not be difficult; some form of genetic algorithm might work well, but this has not been investigated.

Depth Size Optimal networks have been much studied (see e.g. the recent paper by Lin *et al.* (2009), which contains a good list of relevant references). 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. Most approaches to producing DSO networks either restrict attention to a particular small fanout, most often 2 or 4 (e.g. 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). Our approach gives the user fine control over each generated network via the measure function used in its generation, and in some cases by explicit control of the outermost partition.

6.2 Search based generation of shallow networks: a generalization of Ladner–Fischer

We have shown how to generate small shallow networks, given width and depth, see Tables 3 for slack 0 and 4 for slack 1. In the latter table, the networks for widths 32, 48 and 80 are DSO. For the remaining widths, the results for slack 2 are shown in Table 5. Then, all of the generated networks are DSO, and all improve on the Ladner–Fischer construction in both size and maximum fanout. 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

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

Table 5. *Size/fanout for generated networks and classic LF, with slack 2*

Width		64	96	112	128	144	160
Depth		8	9	9	9	10	10
gen	size	118	181	213	245	276	308
	fo	4	4	5	6	4	4
	sfo	3	3	4	6	4	4
LF	size	125	192	227	264	296	331
	fo	9	13	15	17	19	21

delay profiles. However, the simple search described here is designed to work only for flat outermost 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.

6.3 A new parallel prefix network construction for 2^d inputs, depth d.

The use of search allowed us to improve on the Ladner–Fischer construction both for slack zero with width not a power of two, and for slack greater than zero. Examining those results led us finally to a construction that gives the smallest known prefix networks for the remaining case: width a power of two and slack zero (minimum depth).

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 6. 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. The Fich construction produces *restricted* networks whose last output is produced at minimum depth. Looking at the resulting sizes (Table 6), 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

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

Table 6. *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	1,614	1,672	1,658
10	3,327	3,487	3,402
11	6,800	7,206	6,930
12	13,809	14,788	14,044
13	27,922	30,185	28,354
14	56,275	61,356	57,093
15	113,172	124,308	114,740
16	227,221	251,199	230,280
17	455,702	506,578	461,714
18	913,175	1,019,920	925,095
19	1,828,888	2,050,785	1,852,597
20	3,661,337	4,119,280	3,708,669
21	7,327,770	8,267,216	7,422,354
22	14,662,683	16,580,799	14,851,947
23	29,335,580	33,236,622	29,714,342
24	58,685,469	66,594,636	59,443,763

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.

Table 6 also lists network sizes for the three constructions, for depth d networks of width 2^d. Dividing network size by number of inputs, 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.

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.

6.4 Comparision with Hinze's approach to network description

The style of circuit or network description used here and in a number of instances of the Lava approach to hardware description (Bjesse *et al.* 1998; Singh 2000; Naylor 2008; Gill *et al.* 2010) concentrates on the use of *functions* to describe the networks.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

These functional descriptions look like plain structural circuit descriptions, but are in fact circuit *generators*. They are *run* in order to produce circuit representations for analysis and implementation. This approach to circuit description seems to work well in the hands of expert users, leading to novel approaches to synthesis, see for example (Sheeran 2004). It has also proved surprisingly accessible to many novice users, in the context of an undergraduate course on hardware description and verification (Axelsson *et al.* 2005). The most difficult aspect of this approach is the construction of the building blocks that are used to permit the analysis of circuits using non-standard interpretation (NSI). In this paper, that amounts to defining the `Nets` data-type and the function `netFan` that does the necessary gathering of information. The advantage of the approach is that once one has managed to define such a building block, one can freely use existing Haskell functions, and new combinators defined using them, to build circuit descriptions, secure in the knowledge that simply running the resulting functions will give the required analysis. This is a lightweight approach that requires little work.

In building the circuit descriptions, we have relied on ordinary Haskell-style list programming. Although, during the search, the resulting prefix networks are wrapped with the context used to generate them, we have chosen not to make use of this information to give greater type safety during network construction, relying instead on a post hoc correctness check and examination of the generated diagrams. Our main concern has been speed of generation. We have not had any problems with mis-matched sizes, and have therefore chosen not to move towards any form of sized types. It is clear, though, that more careful size checking would benefit potential users of the code discussed in this paper.

Hinze chose a more direct approach to parallel prefix network description, defining a DSL for constructing networks from combining nodes, fan nodes and parallel, sequential and partition combinators (Hinze 2004). This more syntactic approach means that the computation of properties of the networks can be done without the kind of trick that we use to enable NSI in our approach. The syntactic approach lends itself to the kinds of transformation used in Hinze's paper, and our circuit descriptions are unwieldy in comparison. Hinze makes much greater use of the Haskell type system, for instance using classes to capture algebras, and providing size checking of compositions. Nonetheless, it is the case that a Hinze-style description captures something resembling a netlist, and not the kind of decorated netlist in which fans have been assigned a level (as we discussed in Section 3.2). Hinze's paper does not make clear how such a 'nailed down' netlist or (equivalently) the diagrams in the paper are produced from the syntactic descriptions. As far as we can judge, producing the necessary information about delays, so that diagrams are correctly constructed, will essentially amount to walking over the network description, doing something very close to evaluation. And to make matters more interesting, Hinze chose to place operators as late as possible in his diagrams, as opposed to as early as possible, which was the choice made here. Hinze's paper does not reveal how the diagrams were produced. It is to avoid having to write evaluators that we choose to describe networks and circuits as functions and to play the NSI trick. Our motto is 'Why write evaluators when you already have the Haskell evaluator?'. It would

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

be possible to incorporate the syntactic approach into the search-based method, but this has not been tried. Our dilemma is that we wish to have a single high level description, without explicit phases, but at the same time to exert control over the details of the 'nailed down' netlist. We speculate that Hinze made different choices because his main concern was to perform reasoning at the higher level.

The use of fan as the main building block is an idea that we have borrowed from Hinze's approach. Our `build0` combinator is also very similar, but not identical to one used by Hinze when himself generalizing the Brent–Kung construction (p. 16 of Hinze 2004). Viewed in our notation, the difference between the two combinators is in the definition of the `shift` function that converts the top partition into the corresponding partition for the matching fans at the bottom of the network. Our function is:

```
shift :: Partition -> Partition
shift (a:as) = a-1:init as ++ [last as + 1]
```

while Hinze's would be defined in our notation as

```
shiftH :: Partition -> Partition
shiftH (a:as) = a-1:as ++ [1]
```

The hard-wiring of a 1 as the last element of the bottom partition restricts the choice of network shapes that can be described, but was suited to Hinze's purposes in describing and reasoning about standard networks. Our more general combinator has enabled the successful search for new DSO and small, shallow networks.

6.5 The link to VLSI circuits

This paper has concentrated on a more abstract analysis of parallel prefix network topologies. Our experience (and indeed the received wisdom among colleagues in VLSI design) is that staying at as high a level of abstraction as possible gives the greatest possible benefits, even when one is aiming for a non-functional property such as low power consumption. A reasonable way to proceed towards real circuit implementations is, thus, to find good topologies using the methods described here and then to choose among a variety of candidates making use of detailed circuit layouts and CAD tools capable of accurate performance and power estimations for the chosen process. We have made a first link between the kinds of network descriptions given here and the Wired design system, which is a DSL for low-level hardware design, embedded in Haskell (Axelsson 2008). It proved possible to perform the kinds of search mentioned here, but using a CAD tool (Cadence Encounter) that returns power consumption or speed estimates and thus acts as the measure function. The `Nets` datatype was used as the interface between the generation described here and Wired, with the actual VLSI layouts being created using Wired. This enabled a fine-grained choice between related topologies, but did not give any unexpected results, rather confirming that the approach is feasible.

Our guess is that making a first high-level analysis in the abstract and then refining a smaller number of candidates using very fine modelling in this way is a suitable

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

approach, but further experiments are needed to confirm this. This view seems to be in line with related research in the VLSI community. Oklobdzija and coauthors pioneered the use of dynamic programming and search in VLSI design (Chan *et al.* 1992; Martel *et al.* 1995). 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).

7 Results: functional programming

We regard this paper as contributing not only new ideas about prefix network design and exploration, 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.

What are the building blocks of the approach? Let us assume a singly recursive decomposition of the problem. Adding more recursive calls is straightforward. (In our prefix network examples, we started with a singly recursive decomposition and moved on to one with two recursive calls.) We distinguish constructions that can form solutions (call this type A for answer), from other sub-structures used to build them (call these E for extra). Now, we need

1. A combinator *build* that composes a smaller recursive instance of the construction r, plus possibly an additional sub-structure e to make a larger instance, with the shape of the instance determined by a divisor d of type D, giving *build d r e*. The type of *build* is $D \to A \to E \to A$. In our example, a divisor was a partition, which indicated the exact shape of the top-level recursive decomposition, and the additional sub-network was a single wire on the right of the network.
2. A notion of context and the ability to check whether a proposed answer a fits in a context c of type C, which we will write *fits* a c. The function *try* checks whether a sub-structure e fits in a context c. If it does not, *Nothing* is returned, otherwise *Just e*. We also need known solutions for some base case contexts. These will take the form of pairs of type $(C \to Bool, Maybe\ A)$, containing a property of a context and the associated result (which for bad contexts might be *Nothing*), say (p_0, r_0) and (p_1, r_1).
3. A function that takes a context and returns a list (or set) of possible divisors: $divs\ ::\ C \to [D]$.
4. Functions that, given a context and a divisor, compute the new contexts for the recursive call r and for the additional sub-network e. $f_1, f_2\ ::\ C \to D \to C$. We

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
gen                          ::  (A → V) → C → Maybe A
gen  _    c  | p0 c          =   r0
gen  _    c  | p1 c          =   r1
gen  mf   c                  =   bestOn mf $ mapMaybe ans (divs c )
                 where
                   ans d  =  do   r ← gen mf (f0 c d)
                                  e ← try e1 (f1 c d)
                                  return $ build d r e
```

Fig. 27. The overall structure of the code to generate either an answer or *Nothing* for a given measure function and context, for the combinator *build*. We have abstracted away from the wrapping of solutions with their contexts.

also need a proof (formal or otherwise) that $d \in (divs\ c_0)$ & $fits\ r\ c_1$ & $fits\ e\ c_2$ implies $fits\ (build\ d\ r\ e)\ c_0$, where $c_1 = f_1\ c_0\ d$ and $c_2 = f_2\ c_0\ d$.

5. A measure function that takes an answer to a value that can be compared; call this type of values V. Measure functions have type $A \to V$.

Now, we are in a position to write a pseudo-program to find a solution for a given context (Figure 27).

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 the problem of generating best known prefix networks 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. In some cases, we have examined the generated results when near the limits imposed by our restrictions, manually extending them, to get around the restrictions. Similarly, the generated results led finally to a new construction not requiring search. This interplay between automatic and manual methods is important in practice (and not something to be avoided). 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; we have chosen, instead, to refine the search space manually, resulting in a sequence of partition generation functions. This manual refinement process has been essential to our success in finding better networks than those currently known. It would be interesting to develop a library to support the process. For instance, we used permutation generation functions with different costs, in the sense that some are only feasible to use on small sub-problems,

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

while others are less costly (but risk missing solutions). In some cases, we used cheap generation on part of the input, and expensive on the remainder, using trial and error to find the right division. We need a set of combinators for combining such functions, and giving the user the ability to slide the borders between them, giving fine control over the cost of search.

8 Future work

This work opens a number of avenues for future research.

8.1 Making the method more systematic and applying it to other domains

To gain a greater understanding of the possibilities provided by the combination of search and combinators, it will be necessary to develop a framework that makes it easy to describe the search space and how it is to be restricted. Our first step will be to pick a second domain to explore while developing the framework, most likely sorting networks. 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.

8.2 Prefix networks on FPGA

As prefix networks are so ubiquitous, it would make sense to make a serious study of their implementation on advanced Field Programmable Gate Arrays (FPGAs). Such a study appears not to have been done, and this will be our next step. This will involve finding out which topologies best match the exisiting facilities for speeding up carry chains on modern FPGAs.

8.3 Higher radix networks

The work described here considers networks made from two-input prefix networks. Things become even more interesting if one uses larger prefix networks as building blocks. It would be useful to understand such higher radix algorithms, and to investigate the resulting trade-offs in VLSI implementations.

8.4 Exploring the use of other programming languages

The approach described here has demonstrated that a functional approach to parallel prefix network description enables effective exploration of the design space. We have made use of NSI that runs sub-networks on specialised components, in order to analyse networks and compute contexts for recursive calls and sub-networks. In all cases, this has been a form of *forward* analysis, fitting well into the functional paradigm. However, there are cases when one would like to push information about constraints that the context places on *outputs* backwards through sub-networks.

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

Here, we did not need this because we had a single depth constraint on all outputs of the network, and because our recursive decomposition guaranteed that the outputs of recursive calls would then have to have depth one less than this value (because all fans have depth one). But in other examples, we might want to have the sub-networks that are composed *after* recursive calls, through which constraints should be pushed backwards. This leads one to think of relational descriptions, and indeed of the author's earlier work on a relational hardware description language called Ruby (Jones & Sheeran 1990) and of later work by Axelsson on the Wired system (Axelsson 2008). In addition, the fact that we would like to enrich the forms of search that we use leads one naturally to think of logic programming languages. For these reasons, we think that it would be very interesting to explore the development of the ideas introduced here in a functional logic language such as Curry (Antoy & Hanus 2010) or in a library supporting logic programming in Haskell (Naylor *et al.* 2007).

8.5 Search in DSP algorithm development

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 (Axelsson *et al.* 2010) 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. Although it is a longer shot, we are also interested in extending the parallel prefix search to permit the limiting of the number of operators *per level*. From earlier experiments, we know that this then results in the kinds of structures that arise in loop parallelization. This would possibly be a way to adapt to the fixed processor resources on a GPU and to implement prefix networks in a pipelined manner. In that case, the number of inputs to the network would be far greater than the number of available processors, but the restriction to a fixed number of operators per level would give an indication of what work needs to be done at each processor at each phase of the algorithm. This work will be done in the context of a DSL for GPU programming (Svensson *et al.* 2010).

8.6 The missing theory

We need both an extended theory of DSO networks that takes proper account of fanout and an extension of that theory into the realm of small shallow networks.

9 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

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

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. We have improved on the current, published state of the art in the development of DSO network while retaining control of fanout, in the generation of small shallow networks that improve on the classic Ladner–Fischer construction for minimum depth networks of widths not a power of two and for networks of slack one and two. Finally, we filled in the gap (minimum depth network of width a power of two) by proposing a new construction that improves on the smallest known networks. 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. Finally, we would like this paper to remind its readers that even when solving hard problems, one can get far with simple solutions.

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 ideas on how to present the prefix networks, and to Koen Claessen for providing his purely functional memo function. Many thanks to the anonymous referees who made many constructive suggestions and forced me to think harder. My fascination with prefix networks grew out of my contacts with researchers at Intel Strategic CAD Labs.

References

Antoy, S. & Hanus, M. (2010) Functional logic programming, *Commun. ACM*, **53**(4), 74–85.

Axelsson, E., Björk, M. & Sheeran, M. (2005) Teaching hardware description and verification. In *International Conference on Microelectronic Systems Education, MSE*. IEEE, pp. 119–120.

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

Axelsson, E., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J. & Vajda, A. (2010) Feldspar: A domain specific language for digital signal processing algorithms. In *Proceedings of the Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MemoCode*. IEEE Computer Society, pp. 169–178.

Bjesse, P., Claessen, K., Sheeran, M. & Singh, S. (1998) Lava: Hardware design in Haskell. In *International Conference on Functional Programming, ICFP*. ACM, pp. 174–184.

Blelloch, G. 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. Comput.*, **C-31**, 260–264.

Chan, P. K., Schlag, M. D. F., Thomborson, C. D. & Oklobdzija, V. J. (1992) Delay optimization of carry-skip adders and block carry-lookahead adders using multi-dimensional dynamic programming, *IEEE Trans. Comput.*, **41**(8), 920–930.

Claessen, K., Sheeran, M. & Singh, S. (2001) The design and verification of a sorter core. In *Correct Hardware Design and Verification Methods, CHARME*. Lecture Notes in Computer Science, vol. 2144. Springer, pp. 355–369.

Cormen, T. H., Leiserson, C. E, Rivest, R. L. & Stein, C. (2001) *Introduction to Algorithms*. 2nd ed. Cambridge, MA: MIT Press.

Fich, F. E. (1982) *Two Problems in Concrete complexity: Cycle Detection and Parallel Prefix Computation*. Ph.D. thesis, University of California, Berkeley.

Fich, F. E. (1983) New bounds for parallel prefix circuits. In *STOC '83: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. ACM Press, pp. 100–109.

Franchetti, F., de Mesmay, F., McFarlin, D. & Püschel, M. (2009) Operator language: A program generation framework for fast Kernels. In *Proceedings of IFIP Working Conference on Domain Specific Languages (DSL WC)*. Lecture Notes in Computer Science, vol. 5658. Springer, pp. 385–410.

Giegerich, R., Meyer, C. & Steffen, P. (2002) Towards a discipline of dynamic programming. In *Informatik bewegt: Informatik 2002–32. Jahrestagung der Gesellschaft für Informatik e.v. (gi)*. Lecture Notes in Informatics. Bonner Köllen Verlag, pp. 3–44.

Gill, A., Bull, T., Kimmell, G., Perrins, E., Komp, E. & Werling, B. (2010) Introducing Kansas Lava. In *Proceedings of the 21st Symposium on Implementation and Application of Functional Languages, IFL'09*. Lecture Notes in Computer Science, vol. 6041. Springer, pp. 18–35.

Han, T. & Carlson, D. (1987) Fast area-efficient VLSI adders. In *Proceedings of International Symposium on Computer Arithmetic*. IEEE, pp. 49–56.

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

Hinze, R. (2000) Memo functions, polytypically! In *Proceedings of the Second Workshop on Generic Programming, WGP 2000*, Jeuring, J. (ed), pp. 17–32.

Hinze, R. (2004) An Algebra of scans. In *Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 3125. Springer, pp. 186–210.

Jones, G. & Sheeran, M. (1990) Circuit design in Ruby. In *Formal Methods for VLSI Design*, Staunstrup, J. (ed). North-Holland, pp. 13–70.

Knowles, S. (1999) A family of adders. In *Proceedings of International. Symposium on Computer Arithmetic*. IEEE Press, pp. 277–284.

Kogge, P. M. & Stone, H. S. (1973) A parallel Algorithm for the efficient solution of a general class of recurrence equations, *IEEE Trans. Comput*., C-22(8), 786–793.

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

Lakshmivarahan, S., Dhall, S. K. & Yang, C.-M. (1987) On a new class of optimal parallel prefix circuits with (Size+Depth) $= 2n - 2$ and $\lceil logn \rceil \leqslant$ depth $\leqslant (2\lceil logn \rceil - 3)$. In *Proceedings of International Conference on Parallel Processing*. Pennsylvania State University Press, pp. 58–65.

Lin, Y.-C. & Hung, L.-L. (2009) Straightforward construction of depth-size optimal, parallel prefix circuits with fan-out 2, *ACM Trans. Des. Autom. Electron. Syst*., 14(1), 15:1–15:13.

Lin, Y.-C., Hsu, Y.-H, & Liu, C.-K. (2003) Constructing H4, a fast depth-size optimal parallel prefix circuit, *J. Supercomput*., **24**(3), 279–304.

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

Lin, Y.-C. & Su, C.-Y. (2005) Faster optimal parallel prefix circuits: New algorithmic construction, *J. Parallel Distrib. Comput*., 65(12), 1585–1595.

Liu, J., Zhu, Y., Zhu, H., Cheng, C.-K. & Lillis, J. (2007) Optimum prefix adders in a comprehensive area, timing and power design space. In *ASP-DAC'07: Proceedings of the*

2007 Asia and South Pacific Design Automation Conference. Washington, DC, USA: IEEE Computer Society, pp. 609–615.

Martel, C., Oklobdzija, V. G., Ravi, R. & Stelling, P. (1995) Design strategies for optimal multiplier circuits. In *Proceedings 12th IEEE Symposium on Computer Arithmetic*. IEEE, pp. 42–49.

Naylor, M. (2008) *Hardware-Assisted and Target-Directed Evaluation of Functional Programs.* Ph.D. thesis, University of York.

Naylor, M., Axelsson, E. & Runciman, C. (2007) A functional-logic library for wired. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, pp. 37–48.

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

Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W. & Rizzolo, N. (2005) SPIRAL: Code generation for DSP transforms. *Proceedings of IEEE, Special Issue on Program Generation, Optimization and Adaptation*, **93**(2), 232–275.

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

Sheeran, M. (2004) Generating fast multipliers using clever circuits. In *Formal Methods in Computer-Aided Design, FMCAD*. Lecture Notes in Computer Science, vol. 3312. Springer, pp. 6–20.

Sheeran, M. & Parberry, I. (2006) *A New Approach to the Design of Optimal Parallel Prefix Circuits.* Tech. rept. 2006:1. Chalmers: Department of Computer Science and Engineering.

Singh, S. (1992) Circuit analysis by non-standard interpretation. In *Designing Correct Circuits.* IFIP Transactions, vol. A-5. North-Holland, pp. 119–138.

Singh, S. (2000) Death of the RLOC? In *FPGAs for Custom Computing Machines (FCCM)*. IEEE Computer Society Press, pp. 145–152.

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

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

Svensson, J., Sheeran, M. & Claessen, K. (2010) GPGPU Kernel Implementation and Refinement using Obsidian. In *Proceedings of the Seventh International Workshop on Practical Aspects of High-level Parallel Programming, ICCS*. Procedia, pp. 2059–2068.

Voigtländer, J. (2008) Much ado about two: A pearl on parallel prefix computation. In *Proceedings of the 35th Symposium on Principles of Programming Languages*, Wadler, P. (ed), SIGPLAN Notices, vol. 43, no. 1. ACM Press, pp. 29–35.

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).

Wadler, P. (1992) Monads for functional programming. In *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*, vol. 118. Springer-Verlag, NATO ASI Series F: Computer and systems science.

Zhu, H., Cheng, C.-K. & Graham, R. (2006) On the construction of zero-deficiency parallel prefix circuits with minimum depth, *ACM Trans. Des. Autom. Electron. Syst*., **11**(2), 387–409.

Appendix A. Generating diagrams of prefix networks: code associated with the `Net` data type

```
nets :: WDels -> [Net]
nets wds = [Net [] w d | (w,d) <- wds ]
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

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

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

-- shows only operators (and not fanouts)
-- could be modified to show both
instance Show Net where
  show n = show (wire n) ++ "/" ++ show (dots n)


getNets f wds = f netFan (nets wds)

allfansN ns = concat (map fans ns)


nop (_,ws) = length ws-1
fanout (_,ws) = length ws
spanf (_,ws) = last ws - head ws

square x = x*x
cube x = x*x*x

sizeN = sum . map nop . allfansN

maxfoN = maximum . map fanout . allfansN


sumspanN = sum . map spanf . allfansN
```

Appendix 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))]

toTail :: ([b] -> [b]) -> [b] -> [b]
toTail f (a:as) = a:f as

toLast :: (t -> t) -> [t] -> [t]
toLast f (a:as) = init (a:as) ++ [f (last (a:as))]


getNetsW :: WPP -> [Net]
getNetsW (WPat (is,_) p) = getNets p is


size :: WPP -> Int
size = sizeN . getNetsW

maxfo :: WPP -> Int
maxfo = maxfoN . getNetsW
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
sumspan :: WPP -> Int
sumspan = sumspanN . getNetsW


-- computes the max. fanout *minus one*  in a LF network of given slack and
-- width
maxlfo :: Int -> Int -> Int
maxlfo 0 n = fnd2 n
maxlfo k n = maxlfo (k-1) (cnd2 n)

-- computes the max fanout of a LF network of slack k, width n
--   (fanout in our sense, not that used in the LF paper where fanouts from
--    different levels are added)
maxlfo' k n = 1 + maxlfo k n




-- Transliteration of the recurrence from Sheeran and Parberry 2006 giving
-- the width
-- of the widest known DSO network for a given fanout f and depth d
-- The construction that it captures appears still to be the best known.
maxdso :: Int -> Int -> Int
maxdso f 0 = 1
maxdso f 1 = 2
maxdso f d | f >  d = 1 + maxdso f (d-1) + maxdso f (d-2)
maxdso f d | f <= d = f - 1 + sum [maxdso f (d-1-j) |j <- [1..f-1] ++ [f-1]]


build1 :: Partition -> PP a -> PP a -> PP a
build1 ws p q f
   = concat . toTail (map f) . split (shift ws) .
     concat . toInit (toLasts (p f)) .
     toLast (q f) . toInit (map (ser f)) . split ws

buildW1 :: Context -> Partition -> WPP -> WPP -> WPP
buildW1 ctx ws (WPat _ p) (WPat _ q) = WPat ctx (build1 ws p q)


ln2 :: Int -> Int
ln2 1 = 0
ln2 n = 1 + ln2 (n `div` 2)

alog2 :: Int -> Int
alog2 1 = 0
alog2 n = 1 + alog2 (cnd2 n)


maxdk :: Int -> Context -> Int
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
maxdk k (ls,o) = max 0 (o-pk-1)
  where
    (_,pk) = head (drop (length ls - k) ls)

-- the partition function used with the gen function
parts2 :: Int -> Int -> Context -> [Partition]
parts2 f g ctx
  | k2 >  n = [ l : rs ++ [1] | l <- [2..d], rs <- twosE (n-l-1) ]
  | k2 == n = [replicate (fnd2 n) 2]
  | k2 < n  = [ replicate k 2 ++ rs ++ [r]|
                r <- [minr..maxr], rs <- permsUp 2 ff (n-r-2*k) ]
  where
    ff = min f d
    n  = width ctx
    d  = maxd ctx
    k  = 2^(fnd2 d)
    k2 = if even d then 2*k else 3*k
    m = maxdk g ctx
    maxr = min g (2^m)
    minr = max 1 (n - 2^(d-1))

twosE n = [replicate (fnd2 n) 2 | even n]



gen1 :: (Ord a) => Int -> (WPP -> a) -> Context -> WPP
gen1 f mf ctx = fromJust (prefix ctx)
  where
    prefix = memo pm  -- to turn off memoizationsimply delete memo
    pm ctx | width ctx == 1 = try wire ctx
    pm ctx | 2^(maxd ctx)< width ctx = Nothing
    pm ctx | fits ser ctx = Just (WPat ctx ser)
    pm ctx@(is,o)  = bestOn mf $ mapMaybe makeNet (parts3 f g ctx)
      where
        makeNet ds
          = do let sis = split ds is
               let js = map (last.(adLadF wdFan)) $ init sis
               p  <- prefix (js,o-1)
               pr <- prefix (last sis,o-1)
               return $ buildW2 ctx ds p pr
        d = maxd ctx
        w = width ctx
        mind = alog2 w
        slack = d - mind
        g = maxlfo slack w

parts3 :: Int -> Int -> Context -> [Partition]
parts3 f g ctx
  | k2 >  n = [ l : rs ++ [1] | l <- [2..d], rs <- twosE (n-l-1) ]
  | k2 == n = [replicate (fnd2 n) 2]
  | k2 <  n = [ replicate k 2 ++ rs ++ [r]|
                r <- [minr..g], rs <- permsUp2 2 ff (n-r-2*k) ]
  where
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
    ff = min f (2^(d-1))
    n  = width ctx
    d  = maxd ctx
    k  = div n 8 -- adjusted to produce more 2s to reacher larger widths
    k2 = if even d then 2*k else 3*k
    minr = max 1 (n - 2^(d-1))


build2 :: Partition -> PP a -> PP a -> PP a
build2 ws p q f
   = concat . toTail (map f) . split (shift ws) .
     concat . toInit (toLasts (p f)) .
     toLast (q f) . toInit (map (adLadF f)) . split ws

buildW2 :: Context -> Partition -> WPP -> WPP -> WPP
buildW2 ctx ws (WPat _ p) (WPat _ q) = WPat ctx (build2 ws p q)
```

The function `info`, for given slack, small fanout, measure function and width, constructs both the Ladner–Fischer and `gen` networks and records the results of the measure function for each:

```
info :: (Ord a) =>
        Int -> Int -> (WPP -> a) -> Int
        -> (Int, Int, (Int, Int), (Int, Int))
info k f mf i = (i,depth,dp,lf)
  where
    depth = (alog2 i) + k
    ctx = (zdel i,depth)
    dp = mf $ gen f mf ctx
    lf = mf $ WPat ctx (ladF k)
```

Now, it is easy to review results for a number of widths.

```
res k f mf fac n1 n2 = map ((info k f mf).(fac*)) [n1..n2]
```

```
pat :: Int -> Partition
pat k | k < 6 = []
pat k = concat [replicate (2^(k-2*j-1)) (2^j) | j <- [2..(k-1) `div` 2]]

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

To calculate the size of the final construction for width 2^k, use `ppsize k`.

```
ppsize k = pps [1..2^k]

pps :: [Int] -> Int
pps [a]     = 0
pps [i1,i2] = 1
pps is = sum (map aSize (init ss)) + pps js
           + pps (last sis) + length is - length ss
  where
    ss   = tops is
    sis  = split ss is
    js   = map last $ init sis
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press

```
-- aSize gives the size of the small LF networks for input
--  width n a power of two (same as BK size)

aSize :: Int -> Int
aSize 2 = 1
aSize 4 = 4
aSize 8 = 11
aSize n = aSizeL (ln2 n)

aSizeL 1 = 1
aSizeL n = 2*(aSizeL (n-1)) + n
```

https://doi.org/10.1017/S0956796810000304 Published online by Cambridge University Press