# Delay Abstraction in Combinational Logic Circuits

Noriya Kobayashi

C&C Research Laboratories
NEC Corp.
Miyamae-ku, Kawasaki 216 Japan
Tel: +81-44-856-2134
Fax: +81-44-856-2235
e-mail: nk@sbl.cl.nec.co.jp

Sharad Malik

Department of Electrical Engineering
Princeton University
Princeton, New Jersey 09544-5263
Tel: 609-285-4625
Fax: 609-285-3745
malik@princeton.edu

**Abstract— In this paper we propose a data structure for abstracting the delay information of a combinatorial circuit. The particular abstraction that we are interested in is one that preserves the delays between all pairs of inputs and outputs in the circuit. The proposed graphical data structure is of size proportional to $(m + n)$ in best case, where $m$ and $n$ refer to the number of inputs and outputs of the circuit. In comparison, a delay matrix that stores the maximum delay between each input/output pair has size proportional to $m \times n$. We present heuristic algorithms for deriving these concise delay networks. Experimental results shows that, in practice, we can obtain concise delay network with the number of edges being a small multiple of $(m + n)$.**

## I. Introduction

In this paper we propose a data structure for abstracting the delay information of a combinatorial circuit. The particular abstraction that we are interested in is one that preserves the delays between all pairs of inputs and outputs in the circuit. There are several applications of such an abstraction:

- Consider the problem of determining the delay of a pair of cascaded operation units in high level synthesis. (Such a cascade is also referred to as *operator chaining*.) The worst case delay of the cascade is not necessarily the sum of the worst case delays of the individual units. This is because the critical paths in the two units need not be concatenated in the cascade. However, if the delays between all pairs of inputs and outputs of the original units are known, then this information can be used to derive the correct worst case delay.
- Consider the case in logical and physical synthesis when only one module is modified and the change in the delay needs to be propagated through the entire design. A complete pass through the design may be avoided, if for each combinational block we can make available the delay between each input and output pair.

One such delay abstraction is a *delay matrix* that stores the delays for each input-output pair for each combinational block. This requires large memory space since it has $m \times n$ entries, where $m$ and $n$ are the number of input and output terminals of a circuit. All entries of a delay matrix have to be referred during delay computation. Large matrices make the delay computation task slower. Another alternative is to use a network with the same topology as the original circuit. Such a network is typically quite large and it makes the delay computation task much slower.

In [1], delay matrices are used as the timing model for high-level synthesis. In [2], bipartite graphs equivalent to delay matrices are used as the timing model, and [2] mentions a method to reduce the size of the bipartite graphs. Since the method is a sort of relaxation, the reduced bipartite graphs do not keep the complete information.

In order to make the delay computation task faster, a data structure for delay information must be simple and small. In this paper, we propose an efficient data structure for the delay abstraction of a combinatorial circuit. We call it the *concise delay network*. It consists of source vertices, sink vertices, internal vertices and directed edge with weight. Source vertices and sink vertices in a concise delay network correspond to input terminals and output terminals of a circuit, and the maximum weighted path length from a source vertex to a sink vertex is equal to the delay between the corresponding input and output. A concise delay network requires only $m + n$ edges in the best case, and $m \times n$ edges in the worst case.

We present heuristic algorithms for deriving concise delay networks. Our algorithms are very simple and very powerful and are based on applying network transformation rules. Experimental results shows that we can obtain concise delay networks having number of edges that is a small multiple of $m + n$

The proposed data structure and our algorithm can be applied widely in high level synthesis, logic synthesis and layout synthesis.

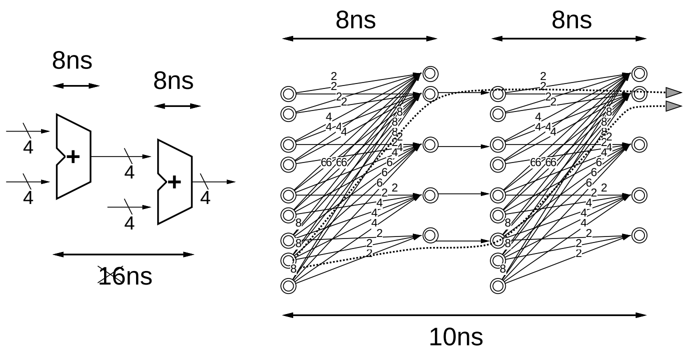

Fig. 1. Two adders in series and their delay computation based on delay bipartite graphs.

$$\begin{array}{c|ccccc} & S0 & S1 & S2 & S3 & COUT \\ \hline CIN & 2 & 4 & 6 & 8 & 8 \\ A0 & 2 & 4 & 6 & 8 & 8 \\ B0 & 2 & 4 & 6 & 8 & 8 \\ A1 & \_ & 2 & 4 & 6 & 6 \\ B1 & \_ & 2 & 4 & 6 & 6 \\ A2 & \_ & \_ & 2 & 4 & 4 \\ B2 & \_ & \_ & 2 & 4 & 4 \\ A3 & \_ & \_ & \_ & 2 & 2 \\ B3 & \_ & \_ & \_ & 2 & 2 \end{array}$$

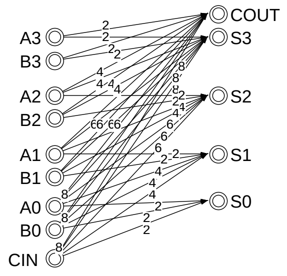

Fig. 2. The delay matrix and the delay bipartite graph of a 4-bit ripple adder.

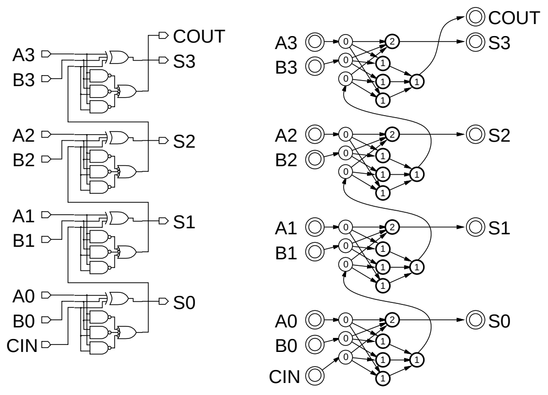

Fig. 3. A gate-level circuit for a 4-bit ripple adder and its original delay network.

## II. Data Structures for Delay Computation

The simplest delay model for a combinatorial circuit block is to use a single value representing the maximum delay among all input-output pairs of the circuit. However, such delay information is too relaxed to permit precise delay computation. According to this single-value delay information, the delay of two adders in cascade is equal to two times of the delay of one adder (see Figure 1). But the actual delay is much smaller than this, since a path consisting of critical paths on both adders never exists in the cascade adders.

What is needed for providing more precise delay computation, is information regarding the delay between all pairs of inputs and outputs in each combinational circuit. There are two kinds of data structures that are currently used to represent this; *delay matrix* and *delay network*. Both permit precise delay computation but require large storage. *Delay matrix* is a $[n \times m]$ matrix, whose $[i, j]$ entry stores the delay value from primary input $PI_i$ to primary output $PO_j$, where $m$ and $n$ are the number of primary inputs and primary outputs, respectively. The $[i, j]$ entry is empty when there is no signal propagation from $PI_i$ to $PO_j$, i.e. there is no data dependency between $PI_i$ and $PO_j$. A delay matrix can be represented by a *directed bipartite graph* $(V_s \cup V_t, E)$. Source vertex set $V_s$ corresponds to primary inputs and sink vertex set $V_t$ corresponds to primary outputs. A directed edge in $E$ from $s_i \in V_s$ to $t_j \in V_t$ corresponds to $[i, j]$ entry and has the same weight as the corresponding value in the matrix. The edge from $s_i$ to $t_j$ can be omitted when no data dependency between $PI_i$ and $PO_j$. (See Figure 2.)

*Delay network* is a directed graph whose topology is the same as the original combinatorial circuit. Its vertices correspond to primary inputs, gates, and primary outputs. Its directed edges correspond to wires. Vertices and edges are weighted by the delay values of corresponding gates and wires. (See Figure 3.)

A delay matrix (a delay bipartite graph) and a delay network can be used for precise delay computation, but both require large memory space and also require large running time for delay computation. As shown in Figure 1, the maximum delay from primary inputs to primary outputs corresponds to a longest path in the weighted graph. A delay matrix store $m \times n$ entries, and a delay bipartite graph stores $m \times n$ edges in worst case. A delay network stores $m + n$ edges in the best case but it may store more than $m \times n$ edges in the worst case. Delay computation using the longest path takes $O(|V| + |E|)$ time in this case. Since these data structures for delay information have a large number of edges, the corresponding delay computation task is slower. This is exacerbated by the fact that this computation may need to be done repeatedly (possibly in an inner loop) in several synthesis applications.

In this paper, we will consider reducing the number of edges and vertices in data structures. Let us consider Figures 3 and 4. Internal vertices in a delay network correspond to gates. If we associate an internal vertex to a 1-bit full adder instead of a gate, we can save edges and vertices while keeping the same delay information of the original delay network. When we use such a delay network, we can reduce the delay computation time. This example suggests that the network topology for the delay abstraction does not have to be the same as the origi-

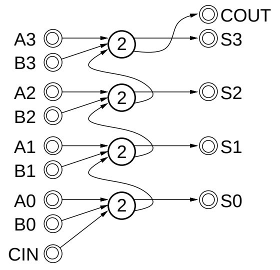

Fig. 4. Reduced delay network of the 4-bit ripple adder.

nal circuit. It is sufficient for the delay network that the maximum weighted length from a source vertex to a sink vertex on a delay network is equal to the delay between the corresponding primary input and primary output. We call such a data structure, the *concise delay network*. The goal of this paper is to construct the minimum concise delay network.

## III. An Algorithm for Concise Delay Networks

In this section, we present a heuristic algorithm to generate a concise delay network with minimum size. Our algorithm transforms a given delay network by applying two transformation rules repeatedly to reduce the number of edges and vertices. We consider only delay networks with edge weight (with no vertex weight). This assumption does not lose any generality, since vertex weight can be easily transformed into edge weight. We denote the weight of edge $e$ by $W(e)$.

First, we define two transformation rules as follows:

**Transformation Rule: TR.B-X** (See Figure 5)

For any arbitrary four vertices $v1, v2, v3, v4$, if there exist edges $p1(v1 \rightarrow v3)$, $p2(v2 \rightarrow v4)$, $x1(v1 \rightarrow v4)$, $x2(v2 \rightarrow v3)$ and equation $W(p1) - W(x1) = W(p2) - W(x2)$ holds. Then delete edges $p1, p2, x1, x2$, and create a vertex $v$ and edges $s1(v1 \rightarrow v)$, $s2(v2 \rightarrow v)$, $t1(v \rightarrow v3)$, $t2(v \rightarrow v4)$. The weight of created edges are defined as follows:

$$\begin{aligned} W(s1) &:= 0 \\ W(s2) &:= W(x2) - W(p1) \\ W(t1) &:= W(p1) \\ W(t2) &:= W(x1) \end{aligned}$$

**Transformation Rule: TR.Y-V** (See Figure 6.)

(Case 1) For arbitrary vertex $v$, if the in-degree of $v$ is 1 and the out-degree of $v$ is greater than 1. Then, delete edge $e(vs \rightarrow v)$, each edge $e_i(v \rightarrow v_i)$ and vertex $v$, and create each edge $f_i(vs \rightarrow v_i)$. The weight of created edges are defined as $W(f_i) := W(e) + W(e_i)$.

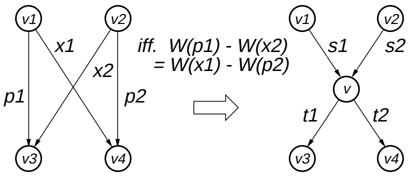

Fig. 5. TR.B-X.

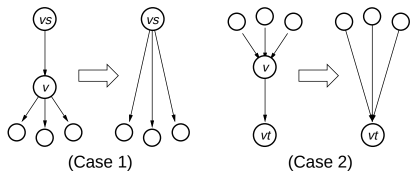

Fig. 6. TR.Y-V.

(Case 2) For arbitrary vertex $v$, if the in-degree of $v$ is greater that 1 and the out-degree of $v$ is 1. Then, delete edge $e(v \rightarrow vt)$, each edge $e_i(v_i \rightarrow v)$ and vertex $v$, and create each edge $f_i(v_i \rightarrow vt)$. The weight of created edges are defined as $W(f_i) := W(e_i) + W(e)$.

On applying a TR.B-X or a TR.Y-V, the maximum weighted length from each source vertex to each sink vertex is not changed, and the number of paths from each source vertex to each sink vertex is not changed. (We omit the proofs of these facts, since they are easy to prove.) A TR.B-X increases the number of vertices by one, while keeping the number of edges the same. A TR.Y-V decreases the number of vertices by one and decreases the number of edges by one. By applying the two transformation rules repeatedly, we can reduce the number of edges in a network. A TR.B-X does not reduce the size of a network, but it reduces the degree of vertices. The decrease of degree of vertices increases the possibility of applying TR.Y-V subsequently. Figure 7 shows delay networks obtained by applying two transformation rules in series.

There are two possibilities to construct an initial delay network. A delay bipartite graph, which is equivalent to the delay matrix, is itself a delay network. An original delay network (based on original circuit topology) is the other possibility and it has to be transformed into a delay network without vertex weight before we start to apply transformation rules. We consider the use of both of them.

The following is a summary of the algorithm:

**Algorithm A** :

1. Given a delay network $N$.
2. Apply TR.B-X and TR.Y-V to $N$ repeatedly in any order until no more application of these two transformation rules is possible.

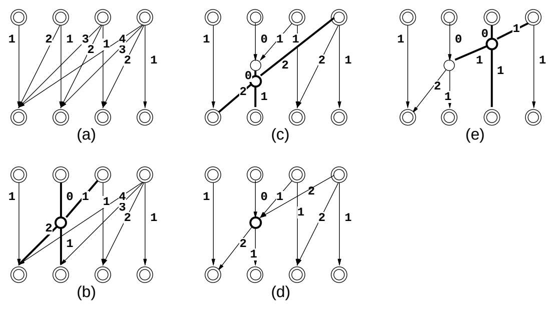
Fig. 7. Delay networks by applying two transformation rules in series. (a) as an initial delay network, we start from the bipartite delay graphs of adder-like circuits, (a) =TR.B-X⇒ (b) =TR.B-X⇒ (c) =TR.Y-V⇒ (d) =TR.B-X⇒ (e), (e) is the resultant delay network.

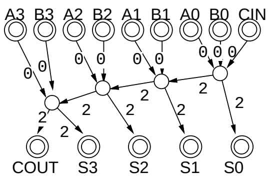

Fig. 8. The resultant concise delay network for the 4-bit ripple adder.

Figure 8 shows the resultant delay network of the 4-bit ripple adder, which is transformed from the bipartite delay graph in Figure 2.

## IV. Extended Algorithms

We have left several questions unanswered for Algorithm A.

1. Which type of initial delay network should be selected, a delay bipartite graph or an original delay network?
2. Do we need to extend B-X transformation rules which transform many-by-many vertices?
3. Does the order of applying two transformation rules affect the size of the resultant delay network? If it does, how do we determine the order of application?

### A. Initial Delay Networks

Both delay bipartite graphs and original delay network can be given as initial delay networks for the algorithm. Note that we assume that original delay networks have been transformed into delay networks without vertex weight. We apply our algorithm twice; once using a delay bipartite graph and once using an original delay network, then we can select the better of the two results.

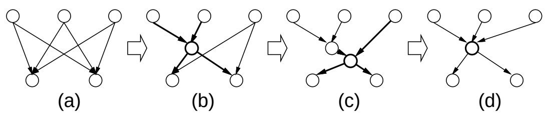
Fig. 9. 3-by-2 transformation emulated by 2-by-2 transformations.

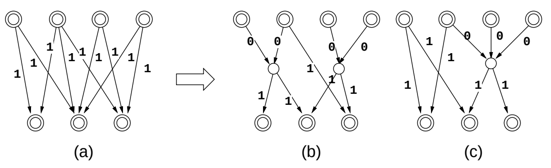
Fig. 10. Two resultant delay networks in different applying orders of transformation rules.

When we start with an original delay network, we will need another transformation rule to obtain better results. Since combinatorial circuits may have reconvergence, there may exist more than one path between two vertices in original delay networks. Consequently, delay networks may have parallel edges between two vertices while we run Algorithm A. On the other hand, parallel edges never appear in delay networks transformed from delay bipartite graphs, since the path between any two vertices is unique, if one exists. Once parallel edges appear in delay networks, they are never removed by TR.B-X and TR.Y-V. We introduce another transformation rule, *TR.PD*, which deletes parallel edges between two vertices except for one edge with maximum weight among them. We extend our algorithm to add TR.PD, and we call it Algorithm A'.

### B. Extended B-X Transformation Rules

TR.B-X is applied to 2-by-2 vertices. A similar transformation rule could be defined for 2-by-3, 3-by-3 or many-by-many vertices. Such extended B-X transformation rules can be realized as a series of (original) TR.B-X and TR.Y-V. (See Figure 9.) Therefore, we use only 2-by-2 TR.B-X and not many-by-many.

### C. Order of Applying the Transformation Rules

The order of applying the transformation rules affects the topology of the resultant delay network. Different orders result in different numbers of edges in the resultant delay network. (See Figure 10.) Unfortunately, we have not found the any consistent order that gives the best result in all cases.

To escape the local optimum, we extend our algorithm to introduce the inverse transformation of TR.B-X.

**Transformation Rule: TR.X-B** (See Figure 11) For vertex $v$ whose in-degree is 2 and out-degree is

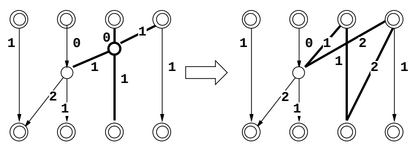

Fig. 11. TR.X-B. (the inverse transformation of TR.B-X).

2, and the four edges around $v$ are: $s1(v1 \to v)$, $s2(v2 \to v)$, $t1(v \to v3)$, $t2(v \to v4)$. Then delete edges $s1, s2, t1, t2$ and vertex $v$, and create four edges $p1(v1 \to v3)$, $p2(v2 \to v4)$, $x1(v1 \to v4)$, $x2(v2 \to v3)$. The weight of created edges are defined follows:

$$\begin{array}{rcl} W(p1) & := & W(s1) + W(t1) \\ W(p2) & := & W(s2) + W(t2) \\ W(x1) & := & W(s1) + W(t2) \\ W(x2) & := & W(s2) + W(t1) \end{array}$$

**Algorithm A+** :

1. Given a delay network $N_0$. Let $i = 0$.
2. Apply TR.B-X, TR.Y-V and TR.PD to $N_i$ repeatedly in any order until no more application of transformation rules is possible. Then, we obtain new delay network $N_{i+1}$.
3. Apply TR.X-B to $N_{i+1}$, repeatedly in any order until no more application of TR.X-B is possible. Then, we obtain new delay network $N'_{i+1}$.
4. If the size of $N'_{i+1}$ is smaller than that of $N'_i$, then let $i = i + 1$ and goto Step 2. Otherwise, we take $N'_i$ as the resultant delay network.

TR.B-X keeps the number of edges same, which means the number of edges in the delay network never increase while Algorithm A+ is going on. The result of Algorithm A+ may be smaller than that of Algorithm A'. However, the result is still only a local optimum.

## V. Experimental Results

In this section we show the results of applying our algorithms to some examples from the ISCAS combinational logic benchmark suite. The experimental results show that our algorithms reduce a large number of edges from the initial delay networks. The number of remaining edges is nearly proportional to the sum of source vertices and sink vertices with a small proportionality factor.

First we demonstrate the characteristics of our algorithms on simple delay models, and then we show the result on real technology delay information, which demonstrates our algorithm is still effective for real world.

As simple delay models, we consider two types of delay models, 'unit' and 'unit-fanout' ('u-f' for short). In delay model 'unit', delay is computed as 1 per node in the circuit, and 'unit-fanout' adds an additional delay of 0.2 per fanout. We use the original circuits of ISCAS benchmark, which means that we do not apply any logic optimization to the circuits. We make both bipartite delay graphs and original delay networks as initial delay networks, and in each case considers two types of delay models. Bipartite delay graphs are generated by computing delays of all input-output pairs and original delay networks are transformed into delay networks without vertex weight.

Table I shows how many edges are reduced by Algorithm A' from each type of initial delay networks, in each delay model. We describe only the number of edges in the tables, since the number of vertices is less than that of edges. There are no significant differences with different delay models except for the bipartite delay graph of C1908. We could not find from our experiment which type of initial delay networks derives smaller resultant networks in general. The conclusion thus is that we should apply our algorithms for both types of initial delay networks and pick the smaller result.

Table II shows how many edges are reduced by iterations of Algorithm A' and inverse transformations. We have not found the optimum order of applying the transformation rules to derive the minimum networks, however Algorithm A+ makes the resultant delay networks smaller.

Table III shows the size of our best resultant delay networks compared with the number of primary inputs and primary outputs. The average number of remaining edges is 2.3 times of the sum of the numbers of primary inputs and outputs, attesting to the quality of the results.

Algorithm A+ works also very effectively for real technology delay factor. We executed technology mapping of the ISCAS benchmark circuits on an NEC gate-array library, and then, we generated delay matrices base on the static delay analysis. The average number of remaining edges is 3.5 times of the sum of the numbers of primary inputs and outputs for actual library.

## VI. Conclusion and future work

This paper presents an efficient data structure, called the concise delay network, for delay abstraction in combinational logic circuits; as well as simple and powerful algorithms for deriving these concise delay networks. The transformation rules defined in this paper are very simple and are easy to implement. The experimental results show that even these simple rules can derive very small sized concise delay networks. We show that concise delay networks which have number of edges that are a small multiple of the the sum of the number of inputs and outputs, are obtained in practice. Therefore, our approach reduces the delay computational cost to $O(|V|)$, while the bipartite graph representation requires $O(|V|^2)$ computational cost, where $|V|$ is the number of terminals.

TABLE I
The numbers of edges in delay networks for applying Algorithm A'

| Circuit | | | #of edges for bipartite delay graphs | | | | #of edges for original delay networks | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | $m$ | $n$ | $E_g$ | $E_1$ $(-\Delta E,\ E_1/(m+n))$ | | | $E_g$ | $E_1$ $(-\Delta E,\ E_1/(m+n))$ | |
| | | | | unit delay | u-f delay | | | unit delay | u-f delay |
| C432 | 36 | 7 | 225 | 69 (69%, 1.60) | 69 (69%, 1.60) | $<$ | 343 | 126 (64%, 2.90) | 126 (64%, 2.90) |
| C499 | 41 | 32 | 1312 | 126 (90%, 1.73) | 126 (90%, 1.73) | $<$ | 440 | 234 (47%, 3.21) | 234 (57%, 3.21) |
| C880 | 60 | 26 | 419 | 267 (36%, 3.10) | 252 (40%, 2.93) | $>$ | 729 | 209 (71%, 2.43) | 218 (70%, 2.53) |
| C1355 | 41 | 35 | 1312 | 213 (84%, 2.92) | 213 (84%, 2.92) | $<$ | 1064 | 234 (78%, 3.21) | 234 (78%, 3.21) |
| C1908 | 33 | 25 | 807 | 62 (12%, 1.07) | 366 (55%, 6.31) | | 1522 | 263 (93%, 4.53) | 300 (80%, 5.17) |
| C2670 | 233 | 140 | 1143 | 590 (48%, 1.58) | 508 (56%, 1.36) | $>$ | 2183 | 424 (91%, 1.14) | 430 (80%, 1.15) |
| C3540 | 50 | 20 | 724 | 467 (35%, 6.67) | 473 (35%, 6.76) | $<$ | 2956 | 621 (79%, 8.89) | 651 (78%, 9.30) |
| C5315 | 178 | 123 | 2978 | 1569 (47%, 5.21) | 1559 (48%, 5.18) | $>$ | 4492 | 869 (81%, 2.89) | 882 (80%, 2.93) |
| C6288 | 32 | 32 | 784 | 146 (81%, 2.28) | 146 (81%, 2.28) | $<$ | 4382 | 1470 (66%, 23.0) | 1470 (66% 23.0) |
| C7552 | 207 | 108 | 3544 | 1814 (49%, 5.76) | 2049 (42%, 6.50) | $>$ | 6206 | 1149 (81%, 3.65) | 1261 (80%, 4.00) |

| | | |
|---|---|---|
| $m$ | : | the number of input terminals. |
| $n$ | : | the number of output terminals. |
| $E_g$ | : | the number of edges in the given delay network. |
| $E_1$ | : | the number of edges in the resulting delay network. |
| $-\Delta E$ | : | $= (E_g - E_1)/E_g$, the ratio how many edges are reduced. |
| $E_1/(m+n)$ | : | the ratio resulting edges per terminals. |

TABLE II
The numbers of edges in delay networks (in unit delay model) on iterations in Algorithm A+.

| Circuit | | | Bipartite delay graphs | | | | Original delay networks | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | $m$ | $n$ | $E_1$ | $f$ | $E_f$ $(-\Delta E,\ \frac{E_f}{m+n})$ | | $E_1$ | $f$ | $E_f$ $(-\Delta E,\ \frac{E_f}{m+n})$ |
| C432 | 36 | 7 | 69 | 2 | 45 (35%, 1.05) | $<$ | 126 | 1 | 126 (0%, 2.93) |
| C499 | 41 | 32 | 126 | 1 | 126 ( 0%, 1.73) | $<$ | 234 | 2 | 232 (1%, 3.18) |
| C880 | 60 | 26 | 267 | 8 | 216 (19%, 2.51) | $>$ | 209 | 2 | 206 (1%, 2.40) |
| C1355 | 41 | 35 | 213 | 5 | 176 (17%, 2.41) | $<$ | 234 | 2 | 232 (1%, 3.18) |
| C1908 | 33 | 25 | 62 | 1 | 62 ( 0%, 1.07) | $<$ | 263 | 7 | 240 (9%, 4.14) |
| C2670 | 233 | 140 | 590 | 5 | 442 (25%, 1.18) | $>$ | 424 | 3 | 413 (3%, 1.11) |
| C3540 | 50 | 20 | 467 | 8 | 366 (22%, 5.23) | $<$ | 621 | 7 | 582 (6%, 8.31) |
| C5315 | 178 | 123 | 1569 | 8 | 1285 (18%, 4.27) | $>$ | 869 | 4 | 827 (5%, 2.75) |
| C6288 | 32 | 32 | 146 | 3 | 122 (16%, 1.91) | $<$ | 1470 | 2 | 1457 (1%, 22.8) |
| C7552 | 207 | 108 | 1814 | 8 | 1554 (14%, 4.93) | $>$ | 1149 | 8 | 1061 (8%, 3.37) |

| | | |
|---|---|---|
| $E_1$ | : | the number of edges just after the first iteration. |
| $E_f$ | : | the number of edges in the final resulting delay network. |
| $f$ | : | the iteration number at the final result. |
| $-\Delta E$ | : | $= (E_1 - E_f)/E_0$, the ratio how many edges are reduced by iterations. |
| $E_f/(m+n)$ | : | the ratio resulting edges per terminals. |

We have an idea for rise/fall delays; we make two vertices for each terminal, one for 'rise' and the other for 'fall', then we make directed edges whose weight means the delay from the corresponding input transition to output transition. Empirical study for such representation is future work. The representation of unbounded delay of boundary gates and the complexity analysis of the minimum network problem are also future work.

## References

[1] S. Note, F. Catthoor, G. Goossens, and H. J. D. Man, "Combined hardware selection and pipelining in high-performance data-path design," *IEEE Trans. on CAD*, vol. 11, pp. 413–423, Apr. 1992.

[2] A. Kuehlmann and R. A. Bergamaschi, "Timing analysis in high-level synthesis," in *1992 IEEE Int. Conf. on CAD*, pp. 349–354, Nov. 1992.

[3] S. Even, *Graph Algorithms*. Computer Science Press, Rockville, MD, 1979.

TABLE III
The size of the resultant delay networks (in unit delay model).

| Circuit | | | Resulting networks | | |
|---|---|---|---|---|---|
| Name | $m$ | $n$ | $E_r$ | $\frac{E_r}{m \times n}$ | $\frac{E_r}{m+n}$ |
| C432 | 36 | 7 | 45 | 0.18 | 1.05 |
| C499 | 41 | 32 | 126 | 0.10 | 1.73 |
| C880 | 60 | 26 | 206 | 0.13 | 2.40 |
| C1355 | 41 | 32 | 176 | 0.13 | 2.41 |
| C1908 | 33 | 25 | 62 | 0.08 | 1.07 |
| C2670 | 233 | 140 | 413 | 0.01 | 1.11 |
| C3540 | 50 | 20 | 366 | 0.47 | 5.23 |
| C5315 | 178 | 123 | 827 | 0.04 | 2.75 |
| C6288 | 32 | 32 | 122 | 0.12 | 1.91 |
| C7552 | 207 | 108 | 1061 | 0.05 | 3.37 |
| | | Average | | 0.13 | 2.30 |

$E_r$ : the number of edges in the final resulting delay network; smaller one among networks starting bipartite delay graphs and original delay networks.