設 計 自 動 化 67－3
（1993．6．25）

Minimization of Delay Buffers in Pipelined Circuit Synthesis

Xing-jian XU　　Mitsuru ISHIZUKA

University of Tokyo

In the design of pipelined digital systems, delay buffers implemented by shift registers are usually introduced into the systems to insure that all the inputs to a processing element arrive at precisely the same time. Automatic techniques for finding the lengths of such buffers, and their proper points of insertion in the system, have been proposed. They are usually based on graph-theoretic approach[2] or linear programming[3]. The fastest approach was developed by Hu with the order of $O(n^4*\log(n))$. In this paper, we introduce a heuristic approach based on some priority functions indicates the charactoristics of the pipelined network to balance the network with a computational complexity of $O(n^2)$ at most.

パイプライン回路自動合成における遅延バファの最小化

徐　行倹、石塚　満

東京大学

パイプライン回路自動合成のときに、各入力データが同時に演算器に入力されるため、シフト・レジスタによる遅延バファーがよく使われる。遅延バファの挿入場所と遅延バファ数を自動的に決めるいくつかの手法がこれまでに提案されている。代表的な手法としては、グラフィクス理論によるアプローチ「2」と線形計画法「3」であり、最も効率的な改良した線形計画法は計算速度が$O(n^4\log(n))$というオーダになっている。本論文ではパイプライン・ネットワークの特徴を表す優先度関数により、最も遅くても$O(n^2)$の速度のヒューリスティクなアプローチを提案する。

1. INTRODUCTION

In digital systems requiring high throughput rate, array processors and/or pipelined designs are often used. The array processor has a simple structure than pipeline; however, it requires more devices to implement and is limited to the case that input data is independent to each other. It is quite widely used in image processing, pattern recognition[1] and so on. Comparing with the array processing, the pipeline has a complex structure, but the advantages of requirement of fewer devices and can be applied to dependent input data. The pipeline is a quite useful technology in design automation of Digital Signal Processors (DSP). To insure that a pipelined structure operates correctly, it is obviously necessary that each multi-input processor in the pipeline receive all its input data at the same time. It is referred as data synchronization[2], or pipeline balancing[3]. A usual method to balance a pipeline is inserting delay buffers, shift registers, onto the path with shorter delay. However, the solution to balancing pipeline is not unique. Obviously, it would be beneficial to determine the minimum number of total delay buffers necessary to balance the pipelined network to reduce the hardware cost.

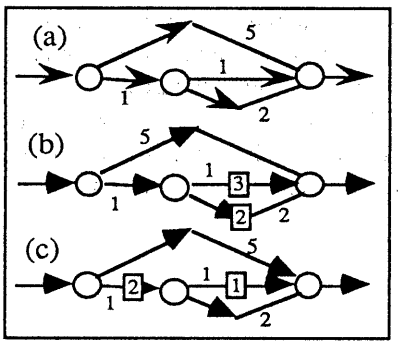

Fig. 1 An example of balancing

Figure 1 illustrates an example of pipeline balancing. Fig. 1(a) shows an unbalanced pipelined network. Fig. 1(b) is a balanced pipeline network, while Fig. 1(c) is an optimal one. A straightforward method to minimize the number of delay buffers was introduced by Gao[3] based on Integer Linear Programming with the exponential computational complexity. Hu[2] proposed an approach to the same problem based on improved Linear Programming basically with the polynomial order of $O(n^4*\log(n))$. In this paper, we introduce our approach based on some priority functions to balance the pipelined network. Our algorithm is able to find the optimal solutions of all the pipelined network we used to test our algorithm at a high speed of $O(n^2)$ at most. Unfortunately, we are not able to prove that our algorithm warrantees that the optimal results can be found for all kinds of pipelined network. But we are sure that the algorithm can find the satisfied solutions at least.

In chapter 2, some definitions and concepts are introduced. Chapter 3 gives a detail explainations of our algorithm, and some experiments are showed in chapter 4. A brief conclusion is given at chapter 5 as the end of this paper.

2. DEFINITIONS

2.1 Constructing a SSFG for a pipelined system

A pipelined network can be represented by a so-called Signal Flow Graph (SFG)[2]. The SFG expression consists of processing nodes (processors), communicating edges (data linkages), and delays assigned to the edges. Delays are associated with SFG edges in the way of that the delay assigned to an edge which is incident our of node u, corresponding to processor p_u, and into node v, corresponding to processor p_v, equals to the processing time of processor p_u. In SFG, all primary input signals to the system originate at node o, and all primary outputs from the system terminate at node t. Both these nodes are assumed to consume zero processing time.

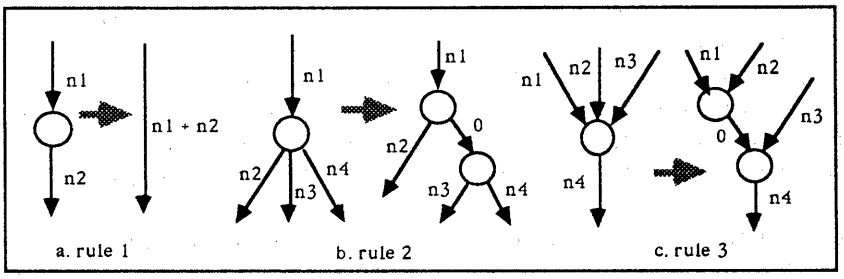

Fig. 2 Rules to construct a SSFG

To realize the speedup of our algorithm, a Simplified Signal Flow Graph (SSFG) is employed to represent a pipelined network, which can be delivered from a SFG based on the transformation rules illustrated in Fig. 2, so that a SSFG consists of fork nodes or joint nodes. A fork is a node with one edge goes to it and two edges go out of it. A joint is a node with two edges go to it and one edge goes out of it. A fork or joint stands for a processor or a dummy processor with zero delay and an edge in a SSFG stands for a serial of processors and communication linkages. A SSFG has two special nodes, one is origin node, *start*, and another is terminus node, *end*.

One feature of such SSFG is that the number of fork is equal to the number of joint. If there are n fork in a SSFG, the number of edges go out of fork and go to joint will be n+1, as one fork breaks one edge into two. In order to joint these n+1 edges into one, n joint is needed, since a joint merges two edges into one. This result will be used in the rest of this paper.

A sub-SSFG is a sub-graph of SSFG, and has an origin node o, a terminus node t, and two different paths. It can be defined as $\{o, t, Lp, Rp\}$, o and t are distinct. *Lp* is the left path and *Rp* is the right path respectively, both of them start at *o* and end at *t*. The SSFG of example1 is illustrated in Fig. 3:

Figure 3 is a SSFG of a simple pipelined network. It has an node set of {e1, e2, e3, e4, e5} and a node set of {n1, n2, n3, n4}. The delays of edges are indicated by {d1, d2, d3, d4, d5}. There are three sub-SSFG in Fig. 3:

a. {n1, n3, e1e3, e2};
b. {n2, n4, e4, e3e5};
c. {n1, n4, e1e4, e2e5};

In order to balance this network, the following three equations have to be satisfied:

for sub-SSFG a: d1+d3 = d2 (1)
for sub-SSFG b: d4 = d3+d5 (2)
for sub-SSFG c: d1+d4 = d2+d5 (3)

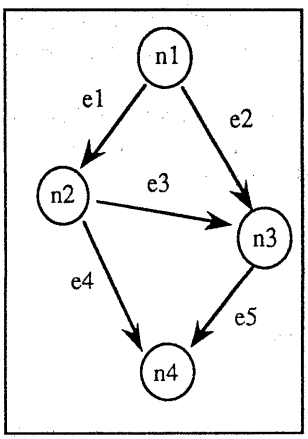

Fig. 3 The primitive set of SSFGs

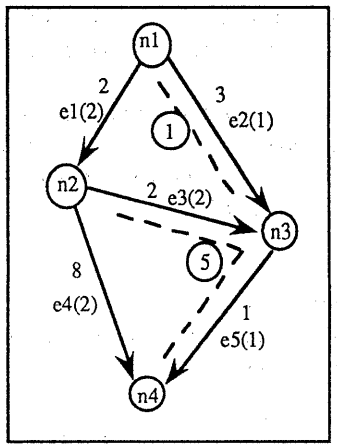

Fig.4 depth from left to right

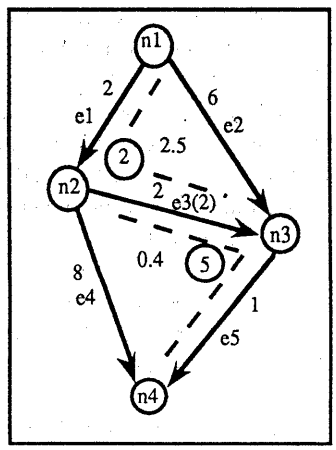

Fig.5 Ratio of sharing delay buffers

Equation (3) can easily be delivered by adding equation (2) to equation (1); that is the rank of the sub-SSFGs is two instead of three. A set of sub-SSFG is called primitive set of sub-SSFGs if the number of sub-SSFGs is the rank of the pipelined network and of each sub-SSFGs in the set is independent to each other.

For a SSFG with 2n nodes of n forks and n joints (the number of fork equals to the number of joint), the rank of the primitive set of sub-SSFGs is n, which equals to the number of forks or joints. It tells us a fact that the primitive set sub-SSFGs can be found between their origin nodes, forks, and their terminus nodes, joints near to their forks.

The problem of balancing the pipelined network is to make two paths of all the sub-SSFGs in the primitive set of sub-SSFG have the same delay by inserting some shift registers onto the path with shorter delay. The purpose of minimizing delay buffer is not only balancing the pipelined network, but also find an approach which uses the minimum number of shift registers.

3. SOLVING THE MINIMIZATION PROBLEM

The discussion in the rest of this paper is based on a SSFG with 2n nodes and an origin node noted by start and a terminus node indicated by end, respectively, without notification.

3.1 Level number nodes

A level number is assigned to a node according to the topological position in vertical direction. It is quite useful to speed up the algorithm of finding the primitive set of sub-SSFGs introduced in next section. A level number of a node can be calculated by:

for origin node of SSFG: level(start) = 1;
for the nodes except origin: level(node) = max(level(parents of node)) + 1;

The procedure of setting node level starts from setting start to one, and continues setting the child nodes of set nodes until all nodes are set. The computational complexity is O(2n).

3.2 Finding the primitive set of sub-SSFGs

The left path can be found by visiting right child nodes from the left child node of a fork, which is the origin fork of the sub-SSFG until a joint node is found. If a right path from the origin to that joint does not exist, then the left path continues visiting SSFG from the child of that joint until another joint is found. This procedure is repeated until two paths are found, a left path and a right path, which start from the origin fork node and end at a joint. As the example of fork node n1 in Fig. 3, in order to find the left path, n2 is visited as it is the left child of origin node n1, then n3 is visited since it is a right child of n3. Because n4 is a joint, e1e3 is supposed to be a left path and n3 is a terminus node of the sub-SSFG. Then, all the joint nodes between level(n1) and level(n3) are searched for finding a right path from n1 to n3. If there is no such path, the left path tries to find a next joint from current joint and then checks if these is a right path. As edge e2 is a path from n1 to n3, then a sub-SSFG of {n1, n3, e1e3, e2} is found.

Finding a right path is straightforward. The procedure searches the nodes between level(origin) and level(joint), which can be visited from origin, for finding if there is a right path from origin to the joint. As a summary of this section, the algorithm of find the primitive set of sub-SSFGs can be written as:

Algorithm1, finding the primitive set of sub-SSFGs

```
for ( all forks )
{
    find a left path from current fork to a joint;
    while ( a right path from current fork to current joint is not found )
    find a left path from current fork to next joint;
}
```

If the maximum length of left and right is m, the computational complexity of algorithm1 will be O(n*m) since finding a right path is at O(m). In the worst case of that the lengths of left and right paths is almost n, the order of algorithm1 becomes $O(n^2)$.

The primitive set of three sub-SSFGs in Fig. 3 can be found by algorithm1 as follows:

a. {n1, n3, e1e3, e2};

b. {n2, n4, e4, e3e5};

3.3 Depth of edges

After the primitive set of sub-SSFGs is found, balancing pipelined network can be started and it becomes the problem of making two paths of all sub-SSFG have the same delay by inserting delay buffers onto the path with shorter delay.

Figure 4 is obtained by assigning delays to each edges. The primitive set of two sub-SSFGs are, s1:{n1, n3, e1e3, e2} and s2: {n2, n4, e4, e3e5}. Two dashed lines indicate the paths with shorter delays of the primitive set of two sub-SSFGs, and two numbers of 1 and 5 in cycles are the number of delay buffers which have to be inserted onto the paths to make the balances to two sub-SSFGs. In sub-SSFG of s1, it is clear that one delay buffer has to be inserted onto edge e2 since there is only choice to insert the delay buffer. But in sub-SSFG s2, five delay buffers can insert onto edge e3, or edge e5. If a delay buffers are inserted onto edge e3, the two paths of sub-SSFG s1 will be unbalanced. As the result of it, another five delay buffers have to be inserted onto edge e2 to balance s1 again. Totally, it costs eleven delay buffers to balance the whole pipelined network while the optimal approach costs only six delay buffers, if five delay buffers are inserted onto edge e5.

We employ so-called depths for all the nodes to solve this problem, which are shown following edge numbers in brackets in Fig. 4. For example, the depth of edge e3 is two, it means if one delay buffer is inserted onto edge e3, it will cost two delay buffers totally as usual. Although there are two places to insert five delay buffers to balance sub-SSFG s2, the delay buffers have to be inserted onto e5 instead of edge e3 since e5 has a smaller depth than edge e5. Figure 4 shows the depth from left to right, which is useful to insert delay buffers onto the shorter path on the right side. The calculation of the depth from left to right is explained as follows:

a. The depth of each edge on the most right side equals to one;

Such edges can be found by visiting SSFG from origin node based on the following two rules:

rule1: visit the right child node if current node is a fork;

rule2: visit child node if current node is a joint.

b. For all the sub-SSFG, the depths of all the edges on the left path equal to:

1 + min(the depths of all the edges on the right path);

Step b has the order of O(n*m), m is the length of sub-SSFG, since it is applied to all the sub-SSFGs, and for a sub-SSFG, the order of calculating the minimum depth is m. In the worst case, the order is $O(n^2)$ when the length of sub-SSFG is n.

Of course another depth from right to left is also needed in the case of inserting delay buffers onto the left path of sub-SSFG. It can be calculated in the same way as the one from left to right.

3.4 Ratio of sharing delay buffers with neighbor sub-SSFG

Figure 5 shows another case of Fig. 4 by changing the delays of some edges. Two delay buffers have to be inserted onto the left path of sub-SSFG s1, while five delay buffers have to be inserted onto the right path of sub-SSFG s2. Edge e3 is shared by both sub-SSFGs. Although the depth of e3 is two and depth of e5 is one, it is easy to understand that the best way is to insert some delay buffers onto e3, so that these delay buffers can be shared by both sub-SSFGs to save delay buffers. The optimal answer is inserting two delay buffers onto edge e3 to balance sub-SSFG s1, and then inserting three delay buffers onto edge e5 to balance sub-SSFG s2. The total number delay buffers needed to balance the pipelined network is five. If five delay buffers are inserted onto edge e3 to balance sub-SSFG first, the left path of sub-SSFG s1 will be changed from a shorter path to a longer path. Another three delay buffers have to be inserted onto edge e2 and totally, it costs eight delay buffers to balance the network.

Sharing delay buffers in these two sub-SSFGs looks the same, but they cause different results. For this purpose, a so-called sharing ratio of sharing delay buffers with neighbor is introduced in this paper. It is calculated by the expression of:

sharing ratio = inserted buffers of all neighbor sub-SSFGs of shorter path / inserted buffers of own sub-SSFG.

The sharing ratio of sub-SSFG s1 is 5/2 = 2.5 and the one of s2 is: 2/5 = 0.4. To deal with sharing delay buffers between two neighboring sub-SSFGs, the sub-SSFG with a smaller sharing ratio has a higher priority to be balanced. Like calculation of depth, the computational complexity of calculation of sharing ratio is log(n*m), where m is the length of sub-SSFG.

3.5 Algorithm

Before the algorithm of minimizing the number of delay buffers is introduced, we would like to explain the physical meanings of depth, sharing ratio and choice by example2 shown in Fig. 6.

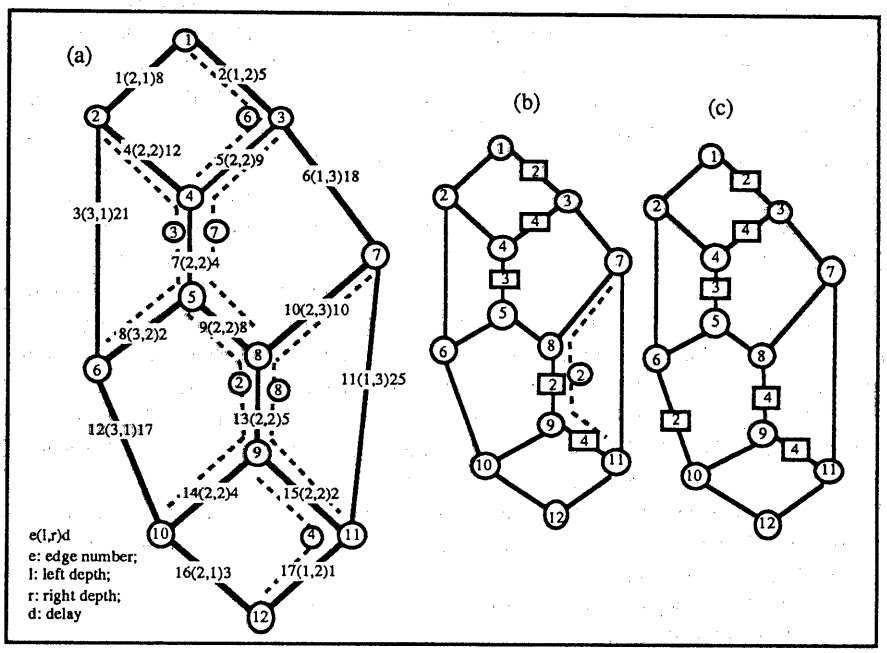

Fig. 6 Example1

Table 1 The primitive set of six sub-SSFGs and their evaluations

sub SSFG	origin node	end node	left path	delay	right path	delay	share ratio	choice
s1	n1	n4	e1 e4	20	e2 e5	14	1.667	infinit
s2	n2	n6	e3	21	e4 e7 e8	18	2.33	1
s3	n3	n8	e5 e7 e9	21	e6 e10	28	1.57	3
s4	n5	n10	e8 e12	19	e9 e13 e14	17	7.5	2
s5	n7	n11	e10 e13 e15	17	e11	25	0.75	2
s6	n9	n12	e14 e16	7	e15 e17	3	2	infinit

There is another priority function of sub-SSFG, which is called choice of sub-SSFG in this paper. The shorter path of sub-SSFG s1 has two edges of e2 and e5. The depth of edge e2 is one, it means that inserting delay buffers to balance s1 does not effects the delay of neighbor sub-SSFG. So the choice of sub-SSFG s1 is infinite. The shorter path of sub-SSFG s2 has three edges e4, e7 and e8. It means that the delay buffers can be inserted onto three paths to balance sub-SSFG s2. But edge s4 is shared by sub-SSFGs s2 and s1, and edge e4 is on the longer path of s1; thus, inserting delay buffers onto edge e4 causes the delay of sub-SSFG s1 longer and it costs more delay buffers to balance s1. Like e4, edge e8 is also not good to insert delay. Edge e7 is shared by sub-SSFGs s2 and s3, and it is on the shorter paths of both sub-SSFGs. It is desirable that nserting delay buffers does not make the delay of neighbor sub-SSFG longer.

There is one edge onto which the delay buffers can be inserted. If there is an edge with a depth of one, set the choice of sub-SSFG to infinite; otherwise, set the choice to the number of edges on shorter path, which are on the shorter path of neighbor sub-SSFG. Table 1 shows all the sub-SSFGs and their evaluations.

Balancing the pipelined network of exmaple1 can be divided into the following four steps:

step1: Because s5 has a sharing ratio of 0.75, which is less than one, it is impossible to share all of its buffers with neighboring sub-SSFGs. Accordingly, s5 has to be balanced before its neighbors. Insert two buffers onto edge e13, and four buffers onto edge edge e15, since two and four buffers are needed to balance s4 and s6. Sub-SSFGs s4 and s6 are balanced, but sub-SSFG s5 is still unbalanced, it will be alanced at later step.

step2: Insert three buffers onto edge e7, since s2 has only one choice of edge e7;

step3: Remained sub-SSFGs are s1 and s3. S3 is balanced before s1 as s3 has fewer choices than s1. After s3 is balanced by inserting 4 buffers onto edge e5, 2 buffers are inserted onto edge e2 to balance s1.

step4: The result after step1 to step3 is illustrated in Fig. 6(b). Only sub-SSFG s5 is unbalanced. To balance s5, two buffers can be inserted onto edges e10, e13, and e15. Because e13 has a right depth of three and e13 and e15 have the same right depth of two, two buffers are inserted onto e13. As sub-SSFG s4 is unbalanced, another two buffers is inserted onto edge e12 to balance sub-SSFG again. The final minimized result is shown in Fig. 6(c).

As a summary of this section, the algorithm of minimizing the number of delay buffers can be written as follows:

step1: Balance all the sub-SSFGs with the sharing ratio smaller than one in the sequence from small sharing ratio to large one;

step2: Balance all the sub-SSFGs in the sequence from small choice to large one. Among the sub-SSFGs, with the same choice, the sub-SSFG with a smaller sharing ratio has a higher priority to be balanced.

step3: After step1 and step2, only the sub-SSFGs with a sharing ratio smaller than one and the one with a zero choice are remained. Such sub-SSFGs are balanced according to the depths of its edges. The edge with a smaller depth has a higher priority.

4. EXPERIMENTS

Figure 8 shows an example introduced in [3]. In finding sub-SSFG s1, the left path can be found easily, which is e1 and the sub-SSFG s1 is supposed to be {n1, n6, e1, ?}. As we introduced in section 3.2, finding a right path has a order of O(m), m is the length of sub-SSFG. In this example, every node is searched for the right path between n1 and n6. It takes O(n) to find the right path of e2e3e6e8. The optimal solution is shown in Fig. 8(b) which is the same as the one in [2].

So far, all the examples introduced in previous sections are plane graphs. Figure 9 illustrates a non-plane graph. Our approach is able to find the optimal solution even in the case of the non-plane graph, which is illustrated in Fig. 8(b). After sorting the primitive set of sub-SSFGs according the their choices and sharing ratios, s1 is balanced first by inserting five buffers onto edge e2; and then, s3 is balanced first by inserting seven buffers onto edge e7 and two onto e5.

Comparing with a plane graph, finding the primitive set of SSFGs and sharing buffers with neighbors are more complex. Because of this, the algorithm has an order of $O(n^2)$. The algorithm, which can only be applied to plane graphs, has the order of O(n*log(n)).

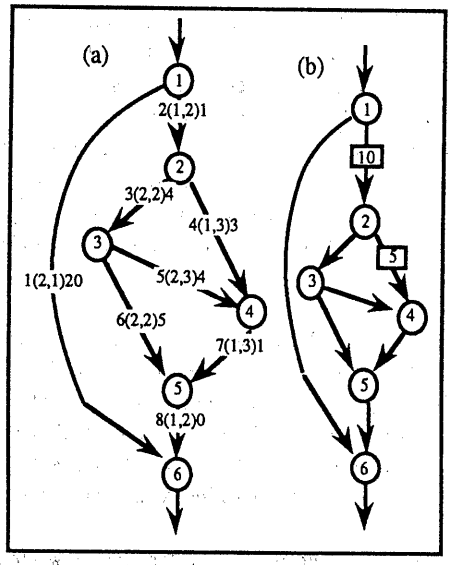

Fig. 7 Example2

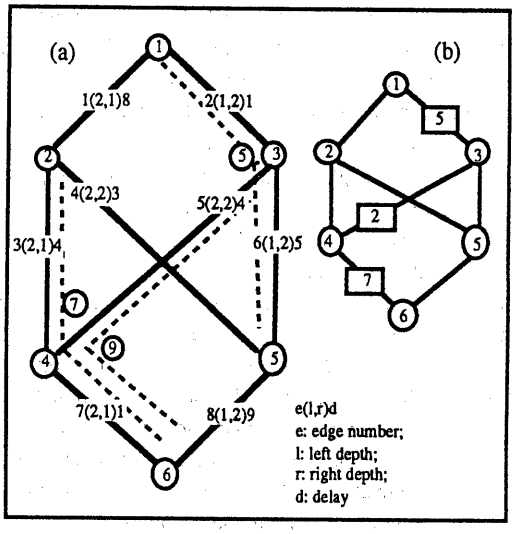

Fig. 8 An example of a non-plane graph

Table2 The primitive set of sub-SSFGs of example2

sub SSFG	origin node	end node	left path	delay	right path	delay	share ratio	choice
s1	n1	n6	e1	20	e2 e3 e6 e8	10	0	infinit
s2	n2	n4	e3 e5	8	e4	3	0	infinit
s3	n3	n5	e6	5	e5 e7	5	0	0

Table3 The primitive set of sub-SSFGs of Fig. 8

sub SSFG	origin node	end node	left path	delay	right path	delay	share ratio	choice
s1	n1	n5	e1 e4	11	e2 e6	6	0	infinit
s2	n2	n6	e3 e7	5	e4 e8	12	1.29	infinit
s3	n3	n6	e5 e7	5	e6 e8	14	0.78	infinit

5. CONCLUSION

We have considered the balancing of pipelined digital systems. With the aid of priority functions of sharing ratio, choice and depth, a pipelined network can be balanced at the order of $O(n^2)$. As we mentioned in chapter 1, we are not able to prove that our algorithm warrantees that the optimal results can be found for all kinds of pipelined network. However at least our algorithm has found the optimal solutions for all the examples used to test our approach. We are sure that the automatic design of pipelined digital systems will become a more and more important technology in DSP design automation. As our future research, we are interested in pipelined digital design automation including evaluations of pipelined digital systems, which will be used to determine if a pipelined system is selected to a behavioral specification. We are also interested in the design automation of pipelined digital systems under the constraints of processing speed and/or chip area.

REFERENCES

[1] O. Hasegawa, W. Wongwarawiwat, C. W. Lee, M. Ishizuka: Real-time Moving Human Face Synthesis Using a Parallel Computer Network, IEEE IECON'91, pp. 1380-1385, Nov. 1991

[2] X. Hu: Minimizing the number of delay buffers in the synchronization of pipelined systems. 28th ACM/IEEE Design Automation Conf. pp. 758-763, 1991.

[3] Guang R. Gao: A code mapping Scheme for Dataflow Software Pipeline, Kluwer Academic Publishers, 1991