設 計 自 動 化 77－19
（1995. 10. 20）

動作記述からのデータフローグラフ生成手法

川田 容子　戸川 望　佐藤 政生　大附 辰夫

早稲田大学理工学部電子通信学科
〒169 東京都新宿区大久保 3-4-1
E-mail: kawata@sato.comm.waseda.ac.jp

あらまし

本稿では，DSP を対象とした高位合成の第 1 段階として，制御構造を含まない演算式から成る動作記述からデータフローグラフ（DFG）を生成する手法を提案する．DFG はその後の合成結果に大きな影響を与えるため，時間，面積といった設計要求を考慮したグラフを生成することが重要である．提案手法は，与えられた時間制約を満たすように，DFG の構造変形によって演算を並列化する．時間制約を満たす DFG から，面積の評価値と共にいくつかの DFG の候補を生成する．複数生成された DFG を，それぞれその後の合成の入力とすることで，より設計要求を満足する合成結果を得ることができると考える．計算機上に提案手法を実装し，評価実験を行った結果について報告する．

キーワード　高位合成，DSP，データフローグラフ，構造変形

An Algorithm for Generating Data Flow Graphs from Behavioral Descriptions

Yoko KAWATA　Nozomu TOGAWA　Masao SATO　Tatsuo OHTSUKI

Dept. of Electronics and Communication Engineering
Waseda University
3-4-1 Okubo, Shinjuku, Tokyo 169, Japan
E-mail: kawata@sato.comm.waseda.ac.jp

Abstract

In this paper, we propose an algorithm for generating data flow graphs (DFGs) from a behavioral description which consists of algebraic expressions without control structures. DFG generation is the first task of the high level synthesis for designing DSP. Since the results of the synthesis much depend on a DFG structure, it is important to consider design requirements such as time and area during generating DFGs. In the proposed technique, we transform a DFG structure and make operations in parallel without increasing a resource cost so that the DFG can satisfy a given time constraint. Among generated DFGs satisfying a given time constraint, multiple DFGs are chosen based on the estimated resource costs. We can obtain better results by preparing multiple DFGs and synthesizing each of them. Experimental results show that we obtain multiple DFGs with low resource costs from a practical behavioral description.

Key Words　*high level synthesis, DSP, data flow graph, transformation*

1 まえがき

本稿では，DSP を対象とした高位合成システムにおけるデータフローグラフ (DFG) 生成手法を提案する．高位合成では，入力された動作記述からデータおよび制御の流れを表すデータフローグラフ (DFG) と呼ばれる中間グラフを生成する．1 つの動作記述を表現する DFG は 1 通りではなく，同じ動作記述から複数の DFG を生成できる (図 1参照)．DFG は，スケジューリングやアロケーション等その後の合成結果に大きな影響を与えるため，時間や面積といった設計要求を十分に考慮した DFG を生成する必要がある．

DFG 生成で，以下の 2 点を考慮する必要があると考える．

(1) DSP を対象とする場合，高速なシステムの設計が要求されるため，時間制約を与える必要がある．時間制約を満足する DFG を生成するため，最大限に演算を並列化できるようにデータ依存関係を分析し，DFG の構造を改良することが重要である．

(2) 一般に与えられた時間制約を満たす DFG は多く存在する．時間制約を満たす DFG の中では，得られる設計ができるだけ小面積である DFG を生成することが重要である．

DFG 生成手法は従来，演算法則に基づく構造変形やリタイミングにより，面積 [5],[6],[8],[9]，時間 [2],[3],[4],[6],[7]，消費電力 [12]，テスト容易化 [10],[11] 等を最適化する方法が提案されてきた．いずれの手法も DFG を改良することにより，より適切な合成結果を得られることが報告されている．しかし，従来手法の多くは DFG のデータ依存解析が局所的な部分に限られている [2],[3],[4],[5]．そのため適用可能な構造変形が限定され，クリティカルパス長の短い DFG を得ることができない．また，最適化の対象となる 1 つの評価値によってのみ DFG を変形し，生成する DFG を 1 つに特定している．一般に合成に対する評価値は 1 つではなく，上述のようないくつかの目標が存在する．これらの目標は互いに依存しあうため，設計に適したトレードオフをとることが重要である．DFG 生成の段階で生成する DFG を 1 つに限定するのは適当でないと考える．

本稿では，時間と面積の 2 点を考慮した DFG 生成手法を提案する．時間と面積はそれぞれ，DFG のクリティカルパス長と必要資源コストで与える．資源とは，演算器，レジスタ，内部接続のことを指し，各資源の面積に比例する値をコストとする．高位合成の結果，必要となる資源のコストの総和を必要資源コストという．

提案手法は，次の特徴を持つ．

- 演算法則に基づく式変形によって DFG のグラフ構造を変形し，演算の並列化を行って，時間制約を満たす DFG を生成する．構造変形は，1 つの節点から調べられる範囲でできる限り大域的に行うことにより，演算の並列性を抽出できる可能性を高める．
- 一般に並列性が小さいと必要資源コストが小さいため，並列性の小さい DFG を初期 DFG とする．構造変形を行う際，できるだけ必要演算器コストを増加させないような処理を行う．
- 与えられた時間制約を満たす DFG を複数生成する．時間制約を満たす DFG に対し必要演算器コストの上下界値を求め，この評価値を基準に DFG を複数選び出す．

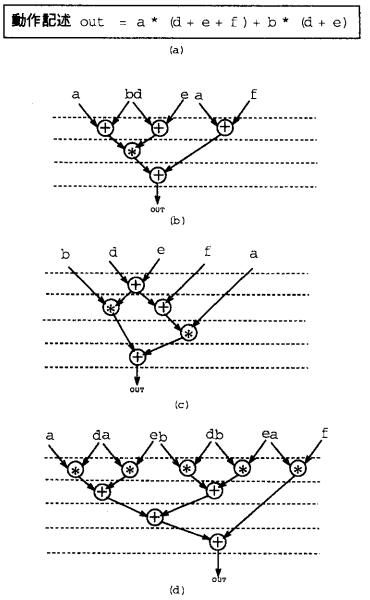

図 1: 同じ動作記述から生成される DFG の例 (入力された動作記述 (a)，動作記述から生成される DFG(b),(c),(d))

以下，本稿で扱う DFG 生成問題を定式化し，DFG 生成手法を提案する．提案手法は，初期 DFG 生成，DFG の構造改良，必要演算器コストの上下界値の算出の 3 段階から成る．続いて，計算機上での実験結果を示し，提案手法の有効性を検証する．

2 DFG 生成問題

DFG は，データの依存関係から生じる演算の順序関係を表現する有向グラフで，節点と枝はそれぞれ演算とデータの流れを表す．有向径路の中で，最も節点数が多い径路をクリティカルパス，クリティカルパス上にある節点数をクリティカルパス長と呼ぶ．

動作記述は制御構造を含まない加算，減算，乗算の演算式に限定する．時間制約は，グラフのクリティカルパス長の最大許容値として与える．演算の実行時間はすべて等しいものとする．パイプラインは考慮しない．DFG 生成の段階では，レジスタや内部接続のコストを見積もることが困難なため，必要演算器コストのみを考慮して DFG 生成を行う．

このとき DFG 生成問題とは，入力として，

- 動作記述
- 時間制約 (クリティカルパス長の最大許容値)

が与えられたとき，

- 時間制約を満たす DFG
- DFG のクリティカルパス長
- DFG の必要演算器コストの上下界値

を出力することをいう．

例えば図 1において，動作記述 (a) に対して時間制約 4 を満足する DFG の例として (b),(c),(d) がある．これらの DFG の中から，それぞれの必要演算器コストの上下界値を評価値として DFG を選出する．図 1では，(b),(c) のコストが (d) のコストに比べて小さいため，(b),(c) が選択される．

3 DFG 生成手法

DFG 生成手法として，次の 3 段階から成る戦略をとる．

Step 1. 動作記述の演算順序を保持した DFG を生成する (初期 DFG 生成)．

Step 2. DFG を走査してクリティカルパス長を削減する構造改良を行う．DFG が構造改良される毎に時間制約を満たすかを調べ，時間制約を満たすならば Step 3 へ．時間制約を満たさない場合は，そのまま構造改良を続ける (構造改良)．

Step 3. 必要演算器コストの上下界値を算出する．あらかじめ保存されている DFG と比べて，上界値と下界値がともに増加していなければ保存する．保存されている DFG のうち資源コストの大きいものを破棄する．保存されている DFG を入力として Step 2 へ戻る (評価，選択)．

Step 1(初期 DFG 生成) では，入力された動作記述の演算式の演算順序を調べ，逐次的に節点と枝を生成する．これは，中置記法 (infix) から後置記法 (postfix) への変換アルゴリズム [16] を適用することで実現できる．動作記述から逐次的に生成された DFG は，並列性が小さくクリティカルパス長が長い構造になっている．この構造から，できるだけ演算器コストを増加させずに並列性を引き出すことで順次構造を変形していくことにより，時間制約を満足する並列性の小さい DFG を生成する．

Step 2(構造改良) は，演算法則に基づいて構造を変形し，演算の並列化をはかる．1 つの節点から調べられる範囲でできるだけ大域的に構造を変形することによって並列性を抽出できる可能性を高め，その結果，よりクリティカルパス長の短い DFG が得ることが期待できる．演算数が増加する構造変形は資源コストを増大させる可能性が高いと考え，演算数が増加しない変形を優先する．さらに，演算数が増加する変形については，変形の過程を組み合わせて複数の DFG を生成し，より資源コストの小さい DFG を得ることを考える (4 章参照)．

Step 3(評価，選択) では，Step 2 の構造変形によって新たに生成された DFG で，クリティカルパス長が時間制約以下であるものに対し，必要演算器コストの上下界値を求める．この評価値を基準に保存すべき DFG を選択する．上界値は，リストスケジューリング [1] を行って算出し，下界値はスケジューリングにおける順序制約を演算のスケジューリング可能範囲 (ASAP 値と ALAP 値の範囲) に緩和して求める方法 [14],[15] を用いる．

以上の DFG 生成手法の中で提案の中心となるのは，Step 2(構造改良) である．次章において構造改良手法を提案する．

4 DFG の構造変形

本章では，3 章で述べた DFG 生成手法のうち，DFG 構造改良手法 (Step 2) を中心に述べる．Step 2 の構造改良は，以下の 3 点を目標とする．

- 与えられた時間制約を満たすようにクリティカルパス長を減らす．
- 必要演算器コストを小さく抑える．
- 複数の DFG を生成する．

4.1 定義

以下に本章で用いる用語および記号を定義する．

演算のタイプ: ある演算が演算を行う演算器ユニットの種類．加算，減算，乗算に限定する．

コントロールステップ: 演算の実行時間の単位でクロックに同期した時間 [1]．

$G_{DF} = (V, E)$**:** 演算の順序関係を表す有向グラフで，DFG を表す．節点 V と枝 E はそれぞれ演算とデータの流れを表す．DFG の節点は，常に ASAP スケジューリングによってコントロールステップに割り当てられていることとする．

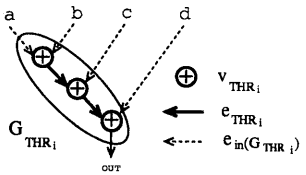

図 2: THR グラフ

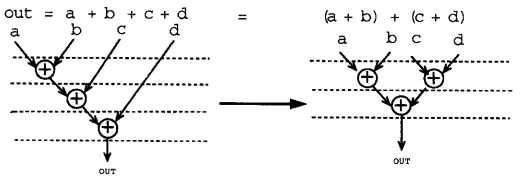

図 3: Tree Height Reduction

ready time: ASAP スケジューリングによって決まる，DFG の各枝の値が使用可能になる最も早いコントロールステップ．

$V_{i,j} \subset V$: 演算のタイプが i，割り当てられているコントロールステップが jである節点の集合．

$E_{in}(v)$: 節点 $v \in V$の入枝の集合で，その要素である枝 $e_{in}(v)$ をファンインと呼ぶ．

$E_{out}(v)$: 節点 $v \in V$の出枝の集合で，その要素である枝 $e_{out}(v)$ をファンアウトと呼ぶ．

$E_{i,j} \subset E$: タイプ i の演算のファンイン $e_{in}(v_i)$ で ready time が jである枝の集合．

$G_{THR_i} = (V_{THR_i}, E_{THR_i})$: G_{DF}の連結した部分グラフで，以下の性質を満たすものを **THR グラフ**と呼び，G_{THR_i}と書く (図 2参照)．

- G_{THR_i}の節点 $v \in V_{THR_i}$は，演算のタイプが i で，枝 $e \in E_{THR_i}$の始点と終点は必ず $v \in V_{THR_i}$なる節点 vに接続している．
- コントロールステップが最も遅い節点を除くすべての節点のファンアウトは，G_{THR_i}に含まれている．

$E_{in}(G_{THR_i})$: THR グラフの節点 $v \in V_{THR}$のファンイン $e_{in}(v)$ のうち，THR グラフに含まれないファンイン ($e_{in}(v) \notin E_{THR_i}$) の集合．この集合を，**THR グラフのファンイン集合**と呼ぶ．

4.2 DFG 構造改良手法

与えられた時間制約を満たす DFG を得るためには，DFG を探索しクリティカルパス長を減らす構造変形をできるだけ多く発見するアルゴリズムを構築する必要がある．

提案手法は，DFG の節点を ASAP スケジューリングによってコントロールステップに割り当て，コントロールステップの早い順にデータ依存関係を調べていく．構造変形によってある演算 (節点) のコントロールステップが早くなると，この演算より後の演算は，コントロールステップが早くなる，あるいは演算のファンインの ready time が早くなるため，さらに構造改良を行うことでコントロールステップが早くなる可能性がある．従って，DFG のクリティカルパス長の削減には，コントロールステップの早い順に演算を調べて構造変形を行っていくことが有効であると考える．また，データ依存関係を大域的に調べるほど，すなわちデータ依存関係を調べる節点数が多いほど，演算の並列化を行う構造変形を抽出できる．提案手法では，1 つの節点から両方向にデータ依存関係を調べ，構造変形を試みる．

DFG の構造は，演算法則 (結合則，交換則，分配則) に基づいて変形することができる．クリティカルパス長を減らす構造変形としては，結合則，交換則に基づいて木の高さを低くする手法 (Tree Height Reduction) [2],[3],[4],[13]，分配則の適用 [3],[6],[8],[13]，因数分解 [7] の 3 つの方法がある．提案手法では，次の 2 つの方法を適用する．

方法 1(Tree Height Reduction). THR グラフ G_{THR_i}において，G_{THR_i}のファンイン集合 $F_{in}(G_{THR_i})$ の中で ready time の最も早いファンインを 2 つ選んで新たな節点を生成する．この節点のファンアウトと残りのファンインの中から同様に，ready time の最も早い枝を 2 つ選んでこれをファンインとして新たな節点を生成する．このような操作を繰り返し，THR グラフを再構築する (図 3参照)．この方法は，THR グラフのクリティカルパス長を最短にできる．

方法 2(演算の分配). 演算の分配則を利用する．例えば，

$$a * (b + c) = a * b + a * c$$

のように演算を分配した構造に変形すると，加算 (減算) と乗算の演算順序が入れ替わる．このことにより THR グラフが形成され，方法 1. の構造改良が可能になる (図 4参照)．

いずれの方法も，実際に構造変形を行う前に，構造変形を行う THR グラフ G_{THR_i}のクリティカルパス長が削減されるかを否かを調べ，パス長が削減される場合のみ構造変形を行う．方法 1 は、THR グラフ G_{THR_i}のファンイン

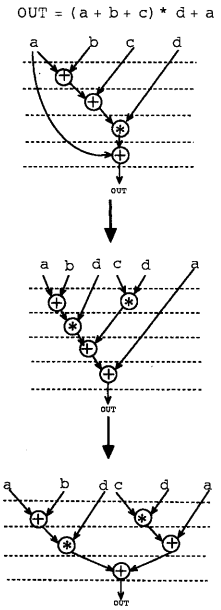

図 4: 演算の分配

集合 $E_{in}(G_{THR_i})$ に属するファンイン $e_{in}(v_i \in V_{THR_i})$ の ready time から，Tree Height Reduction を行って G_{THR_i} のクリティカルパス長が削減できるか否かを判断できる．方法 2 は，演算を分配した結果形成される THR グラフを仮定し，仮定された THR グラフにおいて方法 1 と同様のアルゴリズムを適用することにより，パス長が削減されないあるいは増加する構造変形を除くことができる．

4.3 構造変形と必要演算器コスト

4.2 節で述べた Tree Height Reduction(方法 1) と演算の分配 (方法 2) の 2 種類の構造改良手法を適用して生成される時間制約を満たす DFG の中で，演算器コストが最小である DFG を得ることを目標とする．必要演算器コストは，

- DFG の節点数 (演算数)
- 各節点のスケジューリング可能範囲

に依存する．節点数が多いほど，あるいは各節点のスケジューリング範囲が狭いほど必要演算器コストは大きくなる可能性が高い．方法 1 による構造変形では演算数が変化せず，方法 2 では演算の分配によって乗算が増加する．節点のスケジューリング可能範囲は，その節点が置かれているパスの長さに依存する．すなわち，クリティカルとなるパスの数が少なく，かつクリティカルなパス以外のパスの長さが短い DFG ほど，各節点のスケジューリング範囲が広く必要演算器コストが小さくなる．方法 2 は演算数を増加させる構造変形であるため，方法 1 と比べて長さの長いパス数を増加させる可能性が高い．従って必要演算器コストを小さい DFG を生成するために，方法 1(Tree Height Reduction) を方法 2(演算の分配) より優先することが妥当であると考える．

また，構造改良を適用する際，必要演算器コストができるだけ最小になるような構造変形の組み合わせを選び出すことを考える．与えられた時間制約を満たす DFG を得る構造変形の組み合わせは，DFG の規模の増大に伴って急激に増加する．そこで，方法 1 では必要演算器コストを増加させる構造変形は少ないと考え，適用手順を固定する．方法 2 では，必要演算器コストを増加させる可能性が大きいと考え，構造変形を適用する場所を組み合わせる．ここでは，反復回数に上限を設定した上で，一時的に生成される DFG を保存し，それぞれの DFG に方法 2 を適用する．

4.4 DFG 構造改良アルゴリズム

まず，アルゴリズムで用いる用語，記号を定義する．

Q_r: 出力となる DFG を保存するキュー．

Q_i: 構造変形の過程で一時的に生成される DFG を保存するキュー．

$Iter_{max}$: アルゴリズム中のループ (Step 4) の反復回数の上限値．

以下に，DFG 構造改良アルゴリズムを示す．

Step 1. ASAP スケジューリングを行って，DFG の節点をコントロールステップに割り当てる．

Step 2. コントロールステップの早い順に，コントロールステップ jにある節点 $v_{i,j} \in V_j$それぞれについて (2.1)～(2.3) までの処理を行う．全ての節点を調べ終わった場合は，Step 3 へ．

(2.1) 節点 $v_{i,j}$について，入力および出力の両方向を走査することにより，THR グラフ $G_{THR_i} = (V_{THR_i}, E_{THR_i})$ の抽出を試みる．G_{THR_i}が抽出できた場合は (2.2) へ．

(2.2) Tree Height Reduction の適用を試み，G_{THR_i}のクリティカルパス長の削減を図る．Tree Height Reduction が適用されない場合は Step 2 に戻る．

(2.3) DFG が時間制約を満たすならば，必要演算器コストの上下界値を求める．この上下界値を基準に Q_rに挿入するかを決定する．Step 2 へ．

Step 3. Step 2 によって最終的に生成された DFG を Q_i に保存する．

Step 4. 反復回数が上限値以下の間，Q_iから 1 つ DFG G_{DF}を取り出し，(4.1)～(4.4) までの処理を行う．

(4.1) G_{DF}について，コントロールステップの早い順に，コントロールステップ jにある各節点 $v_{i,j} \in V_j$に対し (4.2)～(4.6) までの処理を行う．

(4.2) 節点 $v_{i,j}$について，分配則適用の可能性を判定する．$i =$ '$*$' であり，かつ $v_{i,j}$のファンインを生成する節点 $v_{i',j'}$において $i' =$ '$+$' であれば，分配則が適用できる．分配則が適用できる場合は，(4.3) へ．できない場合は (4.1) へ．

(4.3) 節点 $v_{i,j}$を $v_{i',j'}$に分配した構造を仮定する．$v_{i,j}$を含む THR グラフ $G_{THR_i} = (V_{THR_i}, E_{THR_i})$，あるいは $v_{i',j'}$を含む THR グラフ $G_{THR_{i'}} = (V_{THR_{i'}}, E_{THR_{i'}})$ の抽出を試みる．THR グラフが抽出できない場合は，(4.1) に戻る．

(4.4) 抽出された THR グラフにおいて，Tree Height Reduction の適用により THR グラフのクリティカルパス長の削減を試みる．クリティカルパス長が削減できる場合は，節点 $v_{i,j}$を $v_{i',j'}$に分配した構造に変形し，Tree Height Reduction を適用する．Tree Height Reduction が適用されない場合は，(4.1) に戻る．

(4.5) 構造変形された DFG を Q_iに挿入する．

(4.6) DFG が時間制約を満たすならば，必要演算器コストの上下界値を求める．この上下界値を基準に Q_rに挿入するかを決定する．(4.1) へ．

生成された DFG を Q_rに挿入する際 (Step (2.3)，Step (4.6))，Q_r中に保存されている DFG と比較して上下界値が共に増加していなければ挿入する．また，Q_r中に保存されている DFG が，挿入した DFG と比較して上下界値が共に大きければ，その DFG を削除する．

5 計算機実験結果

提案手法を，SUN SPARC Station 5 上に C 言語を用いて実装した．2 つの例について DFG を生成した結果を示す．

1 つ目の例は，簡単な動作記述の例 (図 5参照) で，時間制約 (最大クリティカルパス長) = 5 を与えた場合，6 個の DFG が生成された．CPU time は，0.23[sec] かかった．初期 DFG および出力された DFG の必要演算器コストの上下界値，節点数等の値を表 1に示す．表中，DFG 欄は初期 DFG を *init*，出力された DFG を便宜上出力順の番号で記している．L_{cp}，*#nodes*，Lower bound，Upper bound はそれぞれ，DFG のクリティカルパス長，節点数，必要演算器数の上界値，下界値を表す．*#add*，*#mul*はそれぞれ加算器および乗算器を示す．表 1より，必要演算器コストがほぼ同じ値の DFG が生成されたことが分かる．また，初期 DFG を図 6に，生成された DFG のうち DFG1，DFG2，DFG6 をそれぞれ図 7に示す．DFG1，DFG2 は節点数が最も少なく，DFG6 は必要演算器コストの下界値が最も小さい．

```
primary output:  nout;
a = a * (b * c * d + e);
out = (a + b + c) * d + e;
```

図 5: 動作記述の例

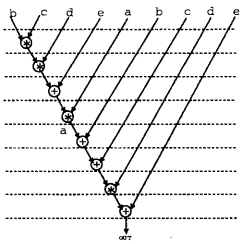

図 6: 初期 DFG

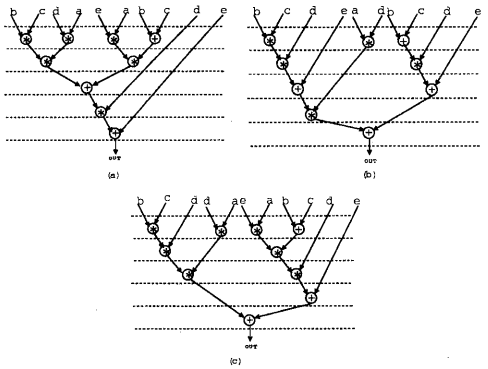

図 7: 実験結果—生成された DFG (時間制約 = 5) (a)DFG1, (b)DFG2, (c)DFG6

2 つ目の例は，DFG 生成やスケジューリング問題のベンチマークとしてよく用いられている 5th-elliptical wave filter[8] の動作記述 (図 8参照) を入力として与えた．CPU time は，0.61[sec] であった．表 2に結果を示す．5th-elliptical wave filter は節点数が 34 個で比較的規模が大きいため，時間制約 11 を満たす DFG は多く存在するが，表 2より必要演算器コストの小さい DFG が生成できたことが分かる．

表 1: 実験結果—生成された DFG (時間制約 = 5)

DFG	L_{cp}	#nodes	Lower bound #add	Lower bound #mul	Upper bound #add	Upper bound #mul
init	8	8	-	-	-	-
1	5	9	1	2	1	2
2	5	9	1	2	1	2
3	5	10	1	2	1	2
4	5	10	1	2	1	2
5	5	10	1	2	1	2
6	5	10	1	1	1	2

init: 初期 DFG, L_{cp} : クリティカルパス長

表 2: 実験結果—5th elliptical wave filter (時間制約 = 11)

n DFG	L_{cp}	#nodes	Lower bound #add	Lower bound #mul	Upper bound #add	Upper bound #mul
init	13	34	-	-	-	-
1	11	34	3	2	4	2
2	11	34	3	2	3	2
3	11	34	2	1	3	2
4	11	35	2	1	3	2
5	11	36	2	1	3	2
6	11	37	2	1	3	2
7	11	38	2	1	3	2
8	11	39	2	1	3	2
9	10	40	2	1	3	2

init: 初期 DFG, L_{cp}: クリティカルパス長

6 むすび

本稿では，時間と面積の 2 点を考慮した DFG 生成手法を提案した．提案手法は，与えられた時間制約を満たす DFG をできるだけ必要演算器コストが小さくなるように複数生成する．今後，提案手法に対し，マルチサイクル演算やパイプラインを考慮できるように拡張する必要がある．

謝辞

本研究の一部は，文部省科学研究費補助金 (特別研究員奨励費) の援助を受けた．また，第 3 著者は倉田記念科学技術振興会 (倉田奨励金) の助成に感謝いたします．

参考文献

[1] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High Level Synthesis*, Kluwer Academic Publishers, pp. 232–237, 1992.

[2] R. Hartray and A. Casvant, "Tree Height Minimization in Pipelined Architectures," *Proc. International Conference Computer-Aided Design,* pp. 112–115, 1989.


```
primary output:  in1, out2, out3, out4,
                 out5, out6, out7, out8, out;
m1 = in1 + in2;
m2 = in2 + in3;
m3 = in7 + in8;
m4 = m2 + in4 + m3;
m5 = m2 + m4 * c1;
m6 = m4 * c2 + m3;
m7 = (m5 + m2) * c3 + m1;
m8 = (m6 + m3) * c4 + in8;
m9 = m5 + m7 + in5;
m10 = m6 + m8 + in6;
out2 = (in2 + m7) * c5 + in1 + m7;
out5 = m9 * c6 + in5;
out3 = out5 + m9;
out4 = m5 + m4 + m6;
out6 = m10 * c7 + in6;
out7 = m10 + out6;
out = (m8 + in8) * c8;
out8 = out + m8;
```

図 8: 動作記述 5th elliptical wave filter

[3] A. Nicolau and R. Potasman, "Incremental Tree Height Reduction for High Level Synthesis," *Proc. 28th ACM/IEEE Design Automation Conference,* pp. 770–774, 1991.

[4] D. A. Lobo and B. M. Pangrle, "Redundant Operator Creation: A Scheduling Optimization Technique," *Proc. 28th ACM/IEEE Design Automation Conference,* pp. 775–778, 1991.

[5] M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations," *Proc. International Conference Computer-Aided Design,* pp. 88–91, 1991.

[6] M. Potkonjak and J. Rabaey, "Maximally Fast and Arbitarily Fast Implementation of Linear Computations," *Proc. International Conference Computer-Aided Design,* pp. 304–308, 1992.

[7] Z. Iqual, M. Potkonjak, S. Dey and A. Parker, "Critical Path Minimization Using Retiming and Algebraic Speed-up," *Proc. 30th ACM/IEEE Design Automation Conference,* pp. 573–577, 1993.

[8] M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations," *IEEE Trans. on Computer-Aided Design,* vol. 13, no. 3, pp. 277–292, 1994

[9] D. J. Kolson, A. Nicolau and N. Dutt, "Integerating Program Transformations in the Memory-Based Synthesis of Image and Video Algorithms," *Proc. 31th ACM/IEEE Design Automation Conference,* pp. 27–30, 1994.

[10] M. Potkonjak and S. Dey, "Optimizing Resource Utilization and Testability Using Hot Potato Techniques," *Proc. 31th ACM/IEEE Design Automation Conference,* pp. 201–205, 1994.

[11] M. Potkonjak, S. Dey, and R. K. Roy, "Considering Testability at Behavioral Level: Use of Transformations for Partial Scan Cost Minimization Under Timing and Area Constraints," *IEEE Trans. on Computer-Aided Design,* vol. 14, no. 5, pp. 531–544, 1995.

[12] A. P. Chandrakasan, R. Mehra, J. Rabaey, M. Potkonjak and R. W. Brodersen, "Optimizing Power Using Transformations," *IEEE Trans. on Computer-Aided Design,* vol. 14, no. 1, pp. 12–31, 1995.

[13] 村岡洋一, 超並列コンパイラ, コンピュータ数学シリーズ 20, pp. 28–64, コロナ社, 1990.

[14] R. Jain, A. Parker and N. Park, "Predicting System-Level Area and Delay for Pipelined and Nonpipelined Designs," *IEEE Trans. on Computer-Aided Design,* vol. 11, no. 8, pp. 955–965, 1992.

[15] S .Y. Ohm, F. J. Kurdasi and N. Dutt, "Comprehensive Lower Bound Estimation from Behaviorical Descriptions," *Proc. International Conference Computer-Aided Design,* pp. 182–187, 1994.

[16] A. M. Tenenbaum, Y. Langsam, and M. J. Augenstein, Data Structuers Using C, *Prince Hall*, pp. 73–97, 1990.