九州大学学術情報リポジトリ
Kyushu University Institutional Repository

イニシエーション・インターバルとアロケーションの制約下における総面積最小を目的としたパイプライン・スケジューリング手法

小玉, 翔
九州大学システム情報科学府情報理学専攻

松永, 裕介
九州大学システム情報科学研究院情報工学部門

https://doi.org/10.15017/13844

出版情報：電子情報通信学会技術研究報告. 108 (478), pp.29-34, 2009-03-11. 電子情報通信学会
バージョン：
権利関係：

KYUSHU UNIVERSITY

イニシエーション・インターバルとアロケーションの制約下における総面積最小を目的としたパイプライン・スケジューリング手法

小玉　翔†　　松永　裕介††

† 九州大学　システム情報科学府　情報理学専攻

†† 九州大学　システム情報科学研究院　情報工学部門

E-mail: †{kodama,matsunaga}@c.csce.kyushu-u.ac.jp

あらまし　本稿では，パイプライン回路を対象とした動作合成における，スループット制約下での回路の総面積最小化を目的としたパイプラインスケジューリング手法を提案する．既存手法の多くがリソース制約とイニシエーション・インターバルの制約下でパイプラインスケジューリングを行うのに対して，提案手法ではアロケーションとイニシエーション・インターバルの制約下でパイプラインスケジューリングを行う．そのため，リソース制約が原因となって実行可能解が得られないという問題が発生しない．また，各演算を割り当て可能なクロックサイクルがリソース制約によって制限されないため，各種の演算器の面積とレジスタの面積を考慮し，回路の総面積が小さくなるようにパイプラインスケジューリングを行うことが可能である．実験の結果，提案手法は既存手法に対して回路の総面積を平均で約 30%削減し，実行時間も実用の範囲内であった．

キーワード　動作合成 , パイプライン合成，パイプラインスケジューリング

Area Optimized Pipeline Scheduling with Initiation Interval and Allocation Constraints

Sho KODAMA† and Yusuke MATSUNAGA††

† Graduate Schoool of Infomation Science and Electrical Engineering , Kyushu University

†† Faculty Schoool of Infomation Science and Electrical Engineering , Kyushu University

E-mail: †{kodama,matsunaga}@c.csce.kyushu-u.ac.jp

Abstract　In this paper, a pipeline scheduling algorithm for minimizing total circuit area under throughput constraint is presented for behavioral synthesis targeted for pipelined circuit. While most previous works perform pipeline scheduling with constraints of initiation interval and resource, proposed method performs pipeline scheduling with constraints of allocation and initiation interval. Therefore, the cituation in which a feasible solution cannot be achieved because of resource constraint is not occured in proposed method. Additionally, pipeline scheduling for minimizing total circuit area can be performed with considering area of each type of functional unit and register because clock cycles to which each operation can be bound is not limited by resource constraint. Experimental results show that proposed method can reduce total circuit area by about 30% on an average compared to previous methods. Also, computation time of proposed method is well with in utility.

Key words　Behavioral Synthesis , Pipeline Synthesis, Pipeline Scheduling

1. はじめに

ASIC(Application Specific Integrated Circuit) や FPGA(Field Programmable Gate Array) などの特定用途向けの LSI(Large Scale Integrated Circuit) において，スループットは性能を測る上での重要な指標の一つである．スループットとは，単位時間当たりに出力可能なデータ数であり，回路の動作周波数を，回路に対してデータを入力可能なクロックサイクル間隔 (イニシエーション・インターバル) で割った値として定義される．回路のスループットを増大させるための手法の一つはパイプライン化であり，イニシエーション・インターバルを当該回路のレイテンシよりも短くすることで回路のスループットを増大させ

る．しかしながら，回路のパイプライン化によるスループットの増大と回路の総面積，消費電力はトレードオフの関係にある．そのため，パイプライン回路を対象とした動作合成 (以下，パイプライン合成) においては，スループットの制約下における回路の総面積や消費電力の最小化，あるいは回路の総面積や消費電力の制約下におけるスループットの最大化が行われる．本稿では，スループットの制約下における回路の総面積最小化を目的としたパイプライン合成手法に着目する．スループットはイニシエーション・インターバルと動作周波数の対として与えるものとし，回路の総面積を演算器の総面積とレジスタの総面積の和として定義する．

動作合成とは，従来の LSI 設計において用いられてきた RTL(Register Transfer Level) 記述を，より抽象度が高い動作記述から自動生成する技術であり，合成対象の回路においてパイプライン化を行う場合にパイプライン合成と呼ばれる．一般に，パイプライン合成では以下の処理が逐次的に行われる．まず，モジュール選択を行って使用する演算器の種類と数を決定する．次にアロケーションにおいて各演算を実行する演算器の種類を決定した上で，パイプラインスケジューリングを行って各演算を実行するクロックサイクルを決定する．最後にバインディングを行なって各演算を実行する演算器のインスタンスと各変数を保持するレジスタのインスタンスを決定する．

スループットの制約下において回路の総面積最小化を行う場合，多くの既存手法においてはモジュール選択，アロケーション，パイプラインスケジューリングの一連の処理をモジュール選択，アロケーションを変更しながら反復して行う．これは，回路の総面積が小さいモジュール選択，アロケーションを一回で決定することが難しいからである．この際，モジュール選択によって使用する各演算器の数は決定しているため，パイプラインスケジューリングはリソース制約とイニシエーション・インターバルの制約下で行われる．リソース制約下におけるパイプラインスケジューリング手法として，パイプライン合成の分野では List Scheduling [1] や Iterative Modulo Scheduling [2] を用いたものが提案されている．これらの手法の問題点はレジスタ数について考慮していないことであり，レジスタ数について考慮した手法としては RAPS [3] が提案されている．また，ソフトウェアパイプライニングの分野においてもレジスタ数を考慮した手法 [4] [5] [6] が提案されており，パイプライン合成に対して適用することが可能である．

回路の総面積が小さいモジュール選択，アロケーションを探索する過程でリソース制約下におけるパイプラインスケジューリングを行う手法は以下の二点で問題がある．第一の問題点は，リソース制約とイニシエーション・インターバルの制約下におけるパイプラインスケジューリングでは実行可能解が保証されないことである．もう一つの問題点は，使用する各演算器の個数を回路の総面積が小さくなるように決定するのが困難なことである．なぜならば，レジスタの総面積がパイプラインスケジューリングの結果として決定するからである．また，レジスタの総面積は，使用する演算器の個数を少なくすることによって必ずしも削減されない．よって，本稿ではアロケーションとイニシエーション・インターバルの制約下における回路の総面積最小を目的としたパイプラインスケジューリング手法を提案する．提案手法においては，アロケーションは入力として与えられるが，使用する各演算器の数については未決定である．モジュール選択はパイプラインスケジューリングの結果として決定される．そのため，リソース制約に起因して実行可能解が得られない問題は発生しない．また，各演算器およびレジスタの面積を同時に考慮してパイプラインスケジューリングを行うことが出来る．類似した仮定の下でのパイプラインスケジューリング手法として Force-Directed Scheduling(FDS) [7] が提案されているが，制約として回路のレイテンシを与える必要がある．パイプラインスケジューリングを行う前に回路の総面積を小さくすることが可能なレイテンシを求めることは困難である．

提案手法は，パイプラインスケジューリングの初期解を決定した上で，初期解を反復して改善することで回路の総面積を削減する．各反復において，演算器およびレジスタの総面積に対して与える影響が大きい演算および変数を選択し，これらに対して局所改善を適用することで質の高い解を短時間で得ることができる．実験の結果，提案手法は回路の総面積を HRMS と比較して平均で約 30%，RAPS と比較して平均で約 25%削減した．また処理時間は実用の範囲内であった．

本稿の構成は以下の通りである．第二章において用語の定義と問題の定義を行う．第三章において提案手法について述べる．第四章において実験結果を示し，第五章で本稿をまとめる．

2. 準　　備

本稿において，パイプラインスケジューリングの入力は Data Dependence Graph(DDG) およびアロケーション g である．DDG $G = (V, E)$ は各ノード $v \in V$ が演算，各エッジ $e_{ij} = (v_i, v_j) \in E$ が演算間の依存関係を表す有向グラフである．各エッジ $e_{ij} \in E$ には Dependence Distance と呼ばれる値 $dist(e_{ij})$ がラベルとして付加される．$dist(e_{ij})$ は，演算 v_i において生成される変数が演算 v_j において消費されるまでに経過するループの回数を表す．即ち，パイプライン実行における k 回目の繰り返しにおいて演算 v_i によって生成された変数が $k + dist(e_{ij})$ 回目の繰り返しにおいて演算 v_j によって消費されることを表す．アロケーションは，演算器ライブラリを $\mathcal{ML}$ として $g : V \rightarrow \mathcal{ML}$ で表される．パイプラインスケジューリングは，イニシエーション・インターバル Δ の制約下で，各演算 $v \in V$ に対して以下の式 (1) を満たすように実行を開始するクロックサイクル (実行開始サイクル) $c(v) (\geqq 0)$ を決定する．

$$\forall v_i \in V, c(v_i) \geqq \max_{v_j \in FI(v_i)} [c(v_j) + l(v_j) - \Delta \times dist(e_{ji})] \quad (1)$$

ここで，$FI(v)$ は演算 v のファンインノードの集合を表し，$l(v_i)$ は演算 v_i のレイテンシを表す．即ち，各演算 $v \in V$ について，そのファンインノードとの依存関係が満たされなければならない．DDG 上の変数の集合を D で表し，変数 $d \in D$ を生成する演算を $v_{gen}(d)$ で表す．また，変数 d を消費する演算の集合を $V_{cons}(d)$ で表す．また，変数 d が生成されるクロックサイクルを $start(d)$，消費されるクロックサイクルを $end(d)$

で表す．変数 d の生存区間は，$start(d)$ と $end(d)$ の順序対として定義される．なお，本稿において DDG は DAG(非循環有向グラフ) であり，各演算および変数のビット幅は全て等しいとする．

パイプラインスケジューリングが終了すると各演算器およびレジスタ数について下界が決定され，回路の総面積の下界を求めることが出来る．本稿では，この下界を回路の総面積と定義する．真値についてはバインディング終了後に決定する．各演算器 $m \in \mathcal{ML}$ の数 $Num(m)$ は，演算器 m に割り当てられている演算の集合 $V_{allocate}(m)$ について，同時に実行状態にある演算の数の最大値として求めることが出来る．ただし，各演算がパイプライン実行されていることを考慮する必要があるため，各演算 v について式 (2) を満たす自然数 ($\geqq 0$) の集合 $X(v)$ を定義する．

$$X(v) = \cup_{i=0}^{\tau(v)-1}[(c(v)+i) \bmod \Delta] \quad (2)$$

ここで，$\tau(v)$ は演算 v が割り当てられている演算器のイニシエーション・インターバルである．また mod は剰余算を表す．即ち，$X(v)$ は演算 v が実行状態にあるクロックサイクルについて Δ で剰余を取ったものの集合である．二つの演算 v_i, v_j は $X(v_i) \cap X(v_j) \neq \phi$ が成り立つ時，同一の演算器インスタンスを共有できない．なぜならば，演算 v はクロックサイクル $\cup_{x \in X(v)}(x + \Delta \times n)(n \geqq 0)$ において各々実行状態にあるからである．なお，$c_{class}(v) = c(v) \bmod \Delta$ を演算 v のサイクル・クラスと呼ぶ．また，クロックサイクルに対して Δ で剰余を取った値を単にサイクル・クラスと呼ぶ．サイクル・クラスは $[0, \Delta-1]$ のいずれかの値を取る．演算器 m の個数 $Num(m)$ は式 (3) で求められる．

$$Num(m) = \max_{i \in [0,\Delta-1]}[\sum_{v_j \in V_{allocate}(m)} |\{i\} \cap X(v_j)|] \quad (3)$$

また，パイプラインスケジューリングの結果に関わらず，演算 m において共有可能な演算の数は上界 $share_{max}(m)$ に制限される．$share_{max}(m) = \lfloor \frac{\Delta}{\delta(m)} \rfloor$ で与えられる．さらに，各演算器 $m \in \mathcal{ML}$ において，アロケーション結果および上界 $share_{max}(m)$ を用いて，演算器インスタンスの個数の下界 $Num_{lower}(m)$ を次のように求めることが出来る．即ち，$Num_{lower}(m) = \left\lceil \frac{|V_{allocate}(m)|}{share_{max}(m)} \right\rceil$ で得られる．

レジスタの数は，同時に生存している変数の数の最大値として求められる．演算器の個数を求める場合と同様に，各変数 $d \in D$ もクロックサイクル $start(d) + \Delta \times n$ において各々生成されることを考慮する必要がある．レジスタ数 Num_{reg} は式 (4) で求められる．

$$Num_{reg} = \max_{i \in [0,\Delta-1]}[\sum_{d \in D} live(d)[i]\ ] \quad (4)$$

ここで，$live(d)[i]$ は，サイクル・クラス i とサイクル・クラス $i+1$ の境界において変数 d が生存している回数を表す要素数 Δ の配列であり，Algorithm 1 で求めることが出来る．なお，$live(d)[\Delta-1]$ は，サイクル・クラス $\Delta-1$ とサイクル・クラス 0 の境界において変数 d が生存している回数を表す．式 (4) から明らかなように，パイプライン合成においては，一つの変数を保持するために複数のレジスタを要する場合がある．

Algorithm 1

```
1: ∀i ∈ [0, Δ − 1] , live(d)[i] = 0
2: for i = start(d) to end(d) − 1 do
3:     tmp = i mod Δ
4:     live(d)[tmp] := live(d)[tmp] + 1
5: end for
```

以上より，回路の総面積 A_{total} は式 (5) で得られる．

$$A_{total} = \sum_{m \in \mathcal{ML}} [A(m) \times Num(m)] + A_{reg} \times Num_{reg} \quad (5)$$

ここで，$A(m)$，A_{reg} はそれぞれ演算器 m，レジスタの面積である．故に，本論文で扱う問題は以下のように定義できる．

- 入力：G，Δ，g
- 目的：式 (1) の制約下で，式 (5) で得られる回路の総面積が小さいパイプラインスケジューリング c を求める

3. 提案手法

提案手法は，パイプラインスケジューリングの初期解を生成した上で，初期解を反復的に改善して回路の総面積を削減する．

3.1 初期解の生成

初期解は，各変数 $d \in D$ の生存区間が短くなるように生成する．これは，個々の変数を保持するために必要となるレジスタの数を削減するためである．まず，ASAP スケジューリング [1] を行い，各演算 $v \in V$ のクロックサイクル $c_{ASAP}(v)$ を保持する．次に，ASAP スケジューリングを行った場合のレイテンシを制約として ALAP スケジューリング [1] を行って各演算 v のクロックサイクル $c_{ALAP}(v)$ を保持する．その上で，各演算 v の実行開始サイクルを $c(v)$ を c_{ASAP} で初期化した後，各演算の実行開始サイクルを優先度に基づいて一回ずつ変更する．優先度としては高さ [2] を使用し，高さが小さい演算から順に変更する．変更対象の演算 v における実行開始サイクルの変更は以下のようにして行う．

- $c(v) < c_{ALAP}(v)$ ならば $c(v) = c_{max}(v)$ に割り当てる．ただし，他の演算において行われた実行開始サイクルの変更を考慮する．また，PO ノードの場合は変更を行わない．
- $c(v) = c_{ALAP}(v))$ ならば変更しない．

ここで，$c_{max}(v)$ は演算間の依存を満たした上で演算 v が取りうる最大のサイクルであり，式 (6) で定義される．

$$c_{max}(v_i) = \max_{v_j \in FO(v_i)}[c(v_j) - l(v_i) + \Delta \times dist(e_{ij})] \quad (6)$$

また，演算 v の実行開始サイクルを変更する際，ファンアウトノードの集合 $FO(v)$ が PO ノードを含み，且つファンアウトノードの全てが PO ノードであるわけではない場合は PO ノードとの依存関係を無視する．これは，PO ノードとの依存関係によって各変数の生存区間の短縮が抑止されることを防ぐためである．尚，PO ノードとの依存関係に違反が生じた場合には

PO ノードのクロックサイクルを違反が解消されるまで増加させる．図 1 に初期解生成の例を示す．この例において，各演算 $v_i (i \in [1, 9])$ のレイテンシは 1 である．ASAP スケジューリングの結果と ALAP スケジューリングの結果を基に各演算の実行開始サイクルを変更することで，各変数の生存区間を短くすることが出来る．

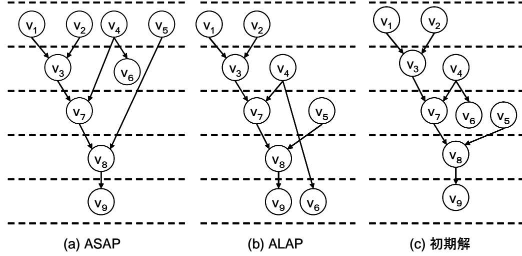

図 1　初期解生成の例

3.2　パイプラインスケジューリングの改善

初期解の生成によって得られたパイプラインスケジューリング結果に対して反復改善法を適用して回路の総面積を削減する．反復改善のフローを図 2 に示す．各反復においては，まず，演算器の総面積削減とレジスタの総面積削減が各々試みられる．これらの処理によって回路の総面積が削減されない場合は SlideSucc を行って回路の総面積削減を試みる．SlideSucc を行っても回路の総面積が削減されない場合に反復は終了する．

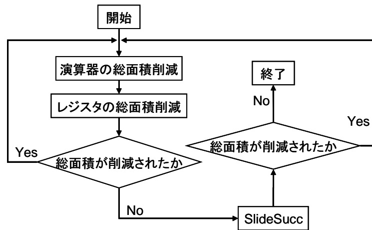

図 2　反復改善のフロー

3.2.1　演算器の総面積削減

まず，実行開始サイクル変更の候補となる演算の集合 V_{cand} を以下のように生成する．

（1） $V_{allocate}(m) \neq \phi$ かつ，現在の演算器数 $Num(m)$ について $Num(m) > Num_{lower}(m)$ が成り立つ演算器の内，$m \notin M_{ignore}$ なる各演算器 $m \in \mathcal{ML}$ から面積 $A(m)$ が最も大きい演算器 m_{sel} を選択する．

（2） 演算器 m_{sel} に割り当てられている演算の集合 $V_{allocate}(m_{sel})$ の内，演算器数の決定に関わっている演算の集合を V_{cand} とする．

即ち，面積が大きい演算器 m を優先的に選択し，当該演算器の個数を削減することを試みる．また，M_{ignore} は演算器の集合であり，M_{ignore} に含まれる演算器は演算器の総面積削減を試みる際に無視される．M_{ignore} は反復改善の開始時において空集合に初期化され，各反復において更新される．また，演算器数の決定に関わっている演算の集合 $V_{det}(m)$ は，実行状態にある演算の数が $Num(m)$ に等しいサイクル・クラス k について，当該サイクル・クラスで実行状態にある演算の集合である．即ち，$\{k\} \cap X(v) \neq \phi$ を満たす演算 v の集合である．演算の集合 V_{cand} を生成した後は，各演算 $v \in V_{cand}$ について実行開始サイクルを変更した場合のゲインを求める．ゲインは回路の総面積における削減とする．実行開始サイクルの変更は，クロックサイクル $c_{min}(v)$ から $c_{max}(v)$ の範囲で試行する．これらは式 (6)，式 (7) で定義されるが，現在の実行開始サイクルからの変更が $\pm\Delta/2$ の範囲に納まるように調整される．これは，演算器の総面積削減を行う場合，演算 v が現在割り当てられているクロックサイクルも含めて Δ よりも大きい区間を探索する必要が無いためである．

$$c_{min}(v) = \max_{v_i \in FI(v)} [c(v_i) + l(v_i) - dist(v_i, v) \times \Delta] \qquad (7)$$

以上の操作を各演算 $v \in V_{cand}$ に対して行ってゲインを求め，最もゲインが大きい変更において回路の総面積が削減される場合にその変更を現在の解に適用する．回路の総面積が削減されない場合には，$v \in V_{cand}$ が割り当てられている演算器 m を演算器の集合 M_{ignore} に追加する．即ち，演算器の総面積削減に際して選択された演算器について，当該演算器の数を削減できなかった場合，その演算器は次回以降の反復における演算器の総面積削減の際に無視される．

3.2.2　レジスタの総面積削減

まず，レジスタ数の決定に関わっている変数の集合 D_{det} を求める．D_{det} は，生存している変数の数がレジスタ数と等しいサイクル・クラスの境界について，その境界で生存している変数の集合である．変数の集合 D_{det} を求めた後，演算の集合 V'_{cand} を式 (8) を用いて求める．

$$V'_{cand} = \bigcup_{d \in D_{det}} [\{v_{gen}(d)\} \cup V_{cons}(d)] \qquad (8)$$

V'_{cand} は，変数 $d \in D_{det}$ を生成，あるいは消費している演算の集合である．レジスタの総面積削減においては，演算の集合 V'_{cand} に含まれる演算を実行開始サイクル変更の候補とする．

まず，演算 $v \in V'_{cand}$ について，実行開始サイクルの変更を試行する．演算器の総面積削減を行う場合と同様に，ゲインは回路の総面積における削減とする．各演算における実行開始サイクルの変更方法は演算器の総面積削減を行う場合と同様である．各演算に対して行った実行開始サイクルの変更のうち，最もゲインが大きい変更について，回路の総面積が削減される場合にその変更を現在のパイプラインスケジューリング結果に対して適用する．

次に，以下の条件を満たす演算の集合 $V_{use} \subseteqq V'_{cand}$ を求める．

- 消費する変数の数が生成する変数の数よりも多い
- 消費しているすべての変数 d について，$d \in D_{det}$ が成り立つ．

各演算 $v \in V_{use}$ については，演算 v が消費する各変数の生存区間の長さを保ったまま変数の生存区間を変更する．即ち，演算 v が消費する変数の集合を $D_{cons}(v)$ とした時，変数 $d \in D_{cons}(v)$ を生成している演算 $v_{gen}(d)$ も演算 v と共に実行開始サイクルを変更する．また，実行開始サイクルの変化量も演算 v と

演算 $v_{gen}(d)$ で等しくする．変数 $d \in D_{cons}$ を生成している演算 $v_{gen}(d)$ は演算 v のファンインノードであるので，演算 $v \in V_{use}$ およびファンインノードの集合 $FI(v)$ について実行開始サイクルを同時に変更する．図 3 に例を示す．この例において，イニシエーション・インターバルは 3 であり，パイプラインスケジューリング変更前のレジスタ数は 4 である．この例では，集合 V_{use} に含まれる演算 v_9 および，演算 v_9 のファンインノード v_7, v_8 の実行開始サイクルを各々1 ずつ小さくすることでレジスタ数を削減している．演算 v_7, v_8, v_9 のいずれか一つの演算について実行開始サイクルの変更を行った場合，レジスタ数を削減することはできない．

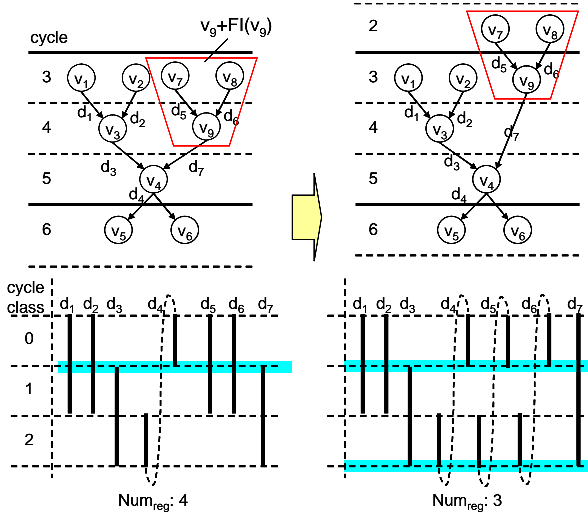

図 3　演算およびそのファンインノードの実行開始サイクル変更

演算 v の実行開始サイクルを現在よりも小さいクロックサイクルに変更する場合，各演算 $v_i \in FI(v)$ と演算 v_i のファンインノード $v_j \in FI(v_i)$ の間の依存関係について考慮する必要があり，この依存関係から実行開始サイクルの変更における変化量の最大値が求まる．実行開始サイクルを現在よりも大きいクロックサイクルに変更する場合は，演算 v と演算 $v_i \in FI(v)$ について，ファンアウト側の依存関係を満たすように実行開始サイクルの変化量の最大値を求める．各演算 $v \in V_{use}$ およびそのファンインノードについて，実行開始サイクルをそれぞれ可能な変化量まで変更する．なお，提案手法においては，まず，現在の実行開始サイクルよりも小さいクロックサイクルへの変更を演算 $v \in V_{use}$ について試行する．その中で，ゲインが最大となる変更を適用した場合に回路の総面積が削減するのであればその変更を適用する．次に，現在の実行開始サイクルよりも大きいクロックサイクルへの変更を各演算 $v \in V_{use}$ について試行する．この場合も，ゲインが最大となる変更を保持しておき，その変更によって回路の総面積が削減する時には現在の解に対して適用する．

3.2.3　SlideSucc

提案手法においては，初期解の性質上，各演算 v において可能な実行開始サイクルの選択肢が少ない場合がある．即ち，$c_{min}(v)$ と $c_{max}(v)$ の差が小さい．そのため，演算器数の削減が十分に進まず，特に面積の多い演算器において問題となり得る．よって，十分にインスタンスの数が削減されていない演算器について以下の処理を行う．まず，演算器数 $Num(m)$ が下界 $Num_{lower}(m)$ に達していない演算器の中で最も面積が大きい演算器を選択する．次に，選択した演算器 m に割り当てられている演算の内，演算器数の決定に関わっている演算 $v \in V_{det}(m)$ に対して以下の処理を行う．まず，集合 $V_{det}(m)$ の中からコスト $cost(v)$ が最も小さい演算 v_{sel} を選択し，演算 v_{sel} のサクセッサ $Succ(v_{sel})$ について実行開始サイクルを一様に Δ 増加させる．その上で，演算 v_{sel} の実行開始サイクルの変更を行う．ここで，演算 v のサクセッサとは，演算 v から PI ノードまでエッジをファンイン側に辿ることで得られるノードの集合である．コスト $cost(v)$ は以下の式で与えられる．

$$
\begin{aligned}
&cost(v) = |D_{str}(v)| \qquad (9)\\
&D_{str}(v) = \{d | d \in D\ ,\ v_{gen}(d) \notin Succ(v)\ \wedge \\
&\qquad\qquad V_{cons}(d) \cap Succ(v) \neq \phi\}
\end{aligned}
$$

更に，コスト $cost(v)'$ に基づいて演算 $v'_{sel} \in V_{det}(m)$ を選択し，演算 v'_{sel} およびそのサクセッサ $Succ(v'_{sel})$ について実行開始サイクルを Δ 増加させる．その上で，演算 v'_{sel} の実行開始サイクルの変更を行う．コスト，$cost(v)'$ は以下の式で与えられる．

$$
\begin{aligned}
&cost(v)' = |D_{str}(v)'| \qquad (10)\\
&D_{str}(v)' = \{d | d \in D\ ,\ v_{gen}(d) \notin Y(v)\ \wedge \\
&\qquad\qquad V_{cons}(d) \cap Y(v) \neq \phi\}
\end{aligned}
$$

ここで，$Y(v) = \{v\} \cup Succ(v)$ である．以上の変更の内，ゲインが最大のものについて，回路の総面積が削減可能であればその変更を適用し，次回の反復に進む．また，この際，演算器の集合 M_{ignore} を空集合とする．削減できない場合は，次に面積が大きい演算器に対して順次同様の処理を行う．

この手法の利点は，変更対象として選択された演算 v_{sel}，v'_{sel} における実行開始サイクルの変更を行うまでは演算器の総面積が変化しないことである．なぜならば，式 (3) より，各演算器の数は各演算 $v \in V$ の実行開始サイクルが Δ の整数倍増加，あるいは減少しても変化しないからである．加えて，レジスタの総面積に関しても，生存区間が長くなる可能性がある変数は，演算の集合 $Succ(v_{sel})$，あるいは $Succ(v'_{sel}) \cup \{v'_{sel}\}$ に含まれる演算に消費されている変数で，且つ当該変数を生成している演算がこれらの演算の集合に含まれない変数のみである．

4.　実　　験

提案手法と既存手法について回路の総面積と処理時間の評価を行った．既存手法としては RAPS [3] および HRMS [4] を用いた．これらの手法は，いずれもリソース制約とイニシエーション・インターバルの制約下でパイプラインスケジューリングを行う．なお，本実験において，同一の種類の演算は同じ種類の演算器に割り当てる．また，アロケーションは各種類の演算について，$Area_{lower}(m) = Num_{lower}(m) \times A(m)$ で得

られる値 $Area_{lower}(m)$ が最小となる演算器を割り当てた．また，既存手法におけるリソース制約は，使用する各演算器 m ごとに $Num_{lower}(m)$ で与えた．また，スループットの制約は 1.0×10^6[sample/s] および 2.0×10^6[sample/s] で与えた．演算器ライブラリは，Synopsys 社の Module Compiler を用いてパイプライン演算器および非パイプライン演算器をそれぞれ合成した．遅延制約は，$300ps$ から $1ns$ まで $100ps$ 刻み，$1ns$ から $20ns$ まで $1ns$ 刻みで変化させて合成を行った．レジスタは Synopsys 社の Design Compiler を用いて生成した．使用したセルライブラリは HITACHI $0.18\mu m$ ライブラリである(注1)．提案手法および既存手法は C++言語にて実装し，Intel Xeon 3.00GHz，メインメモリ 16GB の計算機で実験を行った．

表 1 にスループットの制約が 2.0×10^6[sample/s] の場合における回路の総面積の評価結果を示す．なお，表 1 において，二列目はイニシエーション・インターバルと動作周波数の制約 [MHz] を示している．また，六列目と七列目は，提案手法における回路の総面積を，HRMS および RAPS における回路の総面積を基準として正規化した値を示している．提案手法は回路の総面積を HRMS と比較して平均で約 30%，RAPS と比較して平均で約 27%削減している．スループットの制約が 1.0×10^6[sample/s] の場合における詳細な評価結果は紙面の制約上省略し，概略を述べる．スループットの制約が 1.0×10^6[sample/s] の場合において，提案手法は回路の総面積を HRMS と比較して平均で約 28%，RAPS と比較して平均で約 26%削減している．また，最

表 1　回路の総面積の評価結果 (スループット制約：2.0×10^6)

ベンチマーク		回路の総面積 [μm^2]			Norm.	
	(Δ,Freq.)	HRMS	RAPS	Proposed	HRMS	RAPS
five_tap_fir	(2, 400)	557009	557009	477290	0.857	0.857
	(3, 600)	682638	762357	602920	0.883	0.791
	(4, 800)	599874	623789	552043	0.920	0.885
elliptic_filter	(2, 400)	462162	589712	432712	0.936	0.734
	(3, 600)	359323	518760	351351	0.978	0.677
	(4, 800)	462162	589712	432712	0.936	0.734
integer_transform_horizontal	(2, 400)	2.46E+06	1.85E+06	1.23E+06	0.500	0.666
	(3, 600)	1.82E+06	1.44E+06	1.04E+06	0.569	0.720
	(4, 800)	1.94E+06	1.96E+06	1.02E+06	0.527	0.523
16point_fir_filter	(2, 400)	1.42E+06	1.08E+06	875254	0.618	0.809
	(3, 600)	1.37E+06	1.11E+06	1.01E+06	0.738	0.908
	(4, 800)	1.24E+06	1.16E+06	1.11E+06	0.893	0.960
jpeg_dct_unroll	(2, 400)	4.13E+07	1.93E+07	1.37E+07	0.332	0.711
	(3, 600)	3.12E+07	1.96E+07	1.46E+07	0.466	0.744
	(4, 800)	3.84E+07	5.52E+07	1.39E+07	0.362	0.252
				Ave.	0.701	0.731

も規模が大きいベンチマークである jpeg_dct_unroll における各手法の処理時間を表 2 に示す．jpeg_dct_unroll の DDG は，演算数が 1296，エッジ数が 1839，変数の数が 752 である．提案手法の処理時間は，HRMS と比較して数倍長くなっているが，RAPS と比較して高速であり，実用の範囲内であると考えられる．

5. まとめ

本稿では，アロケーションとイニシエーション・インターバルの制約下における回路の総面積最小を目的としたパイプラインスケジューリング手法を提案した．既存手法がリソース制約下でのパイプラインスケジューリングを行うのに対して，提案手法はアロケーションを入力として与え，各演算器の個数はパイプラインスケジューリングの結果として決定する．そのため，既存手法における，実行可能解が得られる保証がないという問題点を解決できる．また，演算器とレジスタの面積を同時に考慮して回路の総面積を削減することができる．提案手法は初期解を生成した上で反復改善を適用することで回路の総面積を削減する．演算器の総面積やレジスタの総面積に対して与える影響が大きい演算や変数を選択し，これらについて実行開始サイクルの変更や生存区間の変更を行うことで効率的に回路の総面積を削減している．実験の結果，提案手法は回路の総面積を HRMS [4] と比較して平均で約 30%，RAPS [3] と比較して平均で約 25%削減した．また，処理時間も実用の範囲内であった．今後の課題として，解の質の改善および閉路を含んだ DDG への対応が挙げられる．

表 2　jpeg_dct_unroll における処理時間

	runtime[s]		
(Δ, Frequency)	HRMS	RAPS	Proposed
(2, 400)	0.26	15.6	0.99
(3, 600)	0.21	14.27	0.85
(4, 800)	0.24	15.89	0.77
(2,200)	0.25	14.54	0.71
(3,300)	0.21	14.11	0.66
(4,400)	0.22	13.25	1.04
(5,500)	0.21	12.86	0.75
(6,600)	0.22	12.34	0.8
(7,700)	0.21	14.29	0.92
(8,800)	-	-	1.09

謝辞

本研究は，平成 19 年度受託研究費「システムレベル合成アルゴリズムの研究」によるものである．

文献

[1] Giovanni De Micheli, “SYNTHESIS AND OPTIMIZATION OF DIGITAL CIRCUITS”, McGraw-Hill, 1994.
[2] B. R. Rau, “Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops”, Proc. of 27th International Symposium on MICRO, pp. 63-74, 1994.
[3] Negendran Rangan, Karam S. Chatha, “A technique for throughput and register optimization during resource constrained pipelined scheduling”, Proceedings of the 18th VLSI Design, pp. 564-569, Jan, 2005.
[4] Josep Llosa, Mateo Valero, and Antonio Gonzalez, “Modulo Scheduling with Reduced Register Pressure”, IEEE Transactions on Computers, vol. 47, No. 6, pp. 625-638, Jun. 1998.
[5] Richard A. Huff, “Lifetime-Sensitive Modulo Scheduling”, Notices of ACM SIGPLAN vol. 28, issue 6, pp. 258-267, Jun. 1993.
[6] A. E. Eichenberger, E. S. Davidson, “Stage Scheduling: A Technique to Reduce the Register Requirements of a Module Schedule”, Proceedings of the 28th International Synposium of Microarchitecture, pp. 338-349, Nov. 1995.
[7] Pierre G. Paulin, John P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s”, IEEE Trans. on Computer-Aided Design, vol. 8, pp. 661-679, Jun. 1989

(注1)：本研究は東京大学大規模集積システム設計教育研究センターを通し、Synopsys ツールを用いて行われたものである。