設 計 自 動 化 77－20
(1995. 10. 20)

リソースアロケーションを考慮したデータパス・スケジューリング手法

西田 浩一　　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: nishida@ohtsuki.comm.waseda.ac.jp

あらまし

DSP のデータパスを対象とした高位合成において，データフローグラフのスケジューリングが中心的な役割を果たす．スケジューリングでは，アロケーション時に割り当てられるリソースのハードウェア量をできるだけ正確に見積もり，最小化するように各演算をコントロールステップに割り当てることが望ましい．しかも画像処理をはじめとした高速な DSP においては，演算をオーバーラップして実行するパイプライン処理が重要である．本稿では，パイプライン処理に対応し，演算器とレジスタの双方のコストの最小化を目的とする時間制約スケジューリング手法を提案する．本手法では，フォースディレクティッド法のように各演算を最も良いと思われるコントロールステップへ一度に割り当てるのではなく，各演算に対して最も悪いと思われるコントロールステップを徐々に除いていくことにより最終的な解を得る．計算機実験結果により，提案手法は現実的な例題に対し，1秒以下でほぼ最適解に近い解が得られることを示す．

キーワード　DSP，高位合成，スケジューリング，リソースアロケーション，パイプライン処理

A Data Path Scheduling Algorithm with Resource Allocation for DSP Synthesis

Koichi NISHIDA　　Nozomu TOGAWA　　Masao SATO　　Tatsuo OHTSUKI

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

Abstract

In high-level synthesis for DSP data paths, scheduling of data flow graphs plays a primary role. In scheduling, it is required that a hardware resource amount is estimated as precisely as possible, and that operations are assigned to control steps so that a resource amount is minimized. In addition, pipelining that overlaps operations is necessary in high-speed DSP application such as image processing. In this paper, we propose a time constraint scheduling algorithm that deals with pipelining, and minimizes both functional unit and register costs. In our algorithm, the control step regarded as the worst in terms of a resource cost is gradually eliminated for each operation in each iteration. Finally, each operation is assigned to one control step. Experimental results for practical DSP data flow graphs show that our algorithm obtains near optimal solutions in less than one second.

Key Words　*DSP, high-level synthesis, scheduling, resource allocation, pipelining*

1 まえがき

ディジタル信号処理（DSP: Digital Signal Processing）用LSIの設計において，チップコストの削減，設計期間の短縮が重要な課題になっている．動作レベルの記述からレジスタトランスファレベルの記述を合成する高位合成は，LSIのトップダウン設計における最上位に位置し，短期間で大規模な回路を設計するために有効である．

高位合成では，スケジューリングおよび，それに続くリソースアロケーションが中心的な役割を果たす．スケジューリングは，動作記述中の演算の実行順序を決定し，演算をコントロールステップと呼ばれるクロックに同期した時間に割り当てる処理であり，リソースアロケーションは，動作記述中の演算を演算器に，データをレジスタに，データ転送をマルチプレクサやバスに割り当てる処理である [1]．スケジューリングの結果はLSIの性能とチップ面積に大きく影響するため，スケジューリング時にはリソースアロケーションを考慮し，チップ面積等のハードウェアコストを正確に見積り，それに関する制約，および実行時間制約を満たす必要がある．しかも，実行速度とハードウェアコストのトレードオフを対話的に効率良く行えることが望ましい．

スケジューリングは，動作の実行時間を制約とし，必要なハードウェアコストを最小化する時間制約スケジューリングと，ハードウェアコストを制約とし，動作の実行時間を最小化する資源制約スケジューリングの2つに大別される [1]．これらは，いずれもNP完全問題であると考えられる．ごく小規模な問題は0-1整数線形計画法で解くことが可能であるが [2],[3],[4],[5]，実際の高位合成では合成時間の点から実用的ではないため，種々の発見的算法が提案されている [6],[7],[8],[9],[10],[11]．代表的な時間制約スケジューリングにフォースディレクティッド法 [6],[7] が，資源制約スケジューリングにリストスケジューリング [1],[7] がある．従来の多くのスケジューリング手法では，ハードウェアコストの考慮として演算器を中心としており，また合成速度も対話処理には十分であるとは言えなかった．

本稿では，演算器とレジスタ双方のハードウェアコストの最小化を目的とし，マルチサイクル演算器の考慮，パイプライン化DSPの合成 [12] を可能とした時間制約スケジューリング手法を提案する．提案手法を現実的な例題に適用した計算機実験結果を報告する．提案手法はフォースディレクティッド法と同様，演算がコントロールステップに置かれ得る確率からハードウェアのコストを見積る．その際，演算の割当てを一度に1つのコントロールステップに決定してしまうのではなく，割当てにふさわしくないコントロールステップを1つずつ除いていくことにより，最終的な解を得る．計算機実験により，本手法は1秒以下でほぼ最適解に近い解が得られることを確認した．

以下，第2章でスケジューリング問題の定式化，第3章で提案手法について述べ，第4章で計算機実験結果を報告し，第5章で全体のまとめを述べる．

2 問題の定式化

データフローグラフ $G_{df}(V,E)$ を考える．Vは節点の集合であり，外部入力節点の集合 V_I，演算節点の集合 V_{OP}，外部出力節点の集合 V_O に分けられる．Eはデータ依存制約を表す枝の集合である．任意の演算節点 $v \in V_{OP}$は，演算タイプ $type(v)$，演算に必要なクロックサイクル数 $clk(v)$ を持つ．演算タイプの集合を OTとする．演算器タイプは演算タイプと1対1に対応し，演算器タイプの集合も OT で表す．すなわち，データフローグラフにおける任意の演算節点 $v \in V_{OP}$は，その演算を実行可能な演算器タイプを1つだけ持つ．任意の演算器タイプ $ot \in OT$は，ハードウェアコスト $cost(ot)$，演算の実行に必要なクロックサイクル数 $clk(ot)$ を持つ．データを一時記憶するレジスタのハードウェアコストを $cost_{reg}$とする．

演算節点 $v \in V_{OP}$は，コントロールステップと呼ばれるクロックに同期したタイムステップに割り当てられる．この割当てをスケジューリングという．

$delay$を，あるサンプリングデータが外部入力 V_Iに入力されてから，その入力データに対する演算結果が外部出力 V_Oに出力されるまでのクロックサイクル数とする．コントロールステップの集合を，$CS = \{cs_i \mid 1 \leq i \leq delay\}$ とする．全ての外部入力はコントロールステップ cs_1で入力され，全ての外部出力はコントロールステップ cs_{delay} で出力されるものとする．

$latency$を，連続した2つのサンプリングデータ間のクロックサイクル数とする．$latency$ は，DSPを実現するステートマシンの状態数と一致する．ステートマシンの状態の集合を，$ST = \{st_i \mid 1 \leq i \leq latency\}$ とする．パイプライン化されていないDSPにおいては，$latency = delay$ となる．図1に，コントロールステップ $cs \in CS$と状態 $st \in ST$の関係を示す．

演算節点 $v \in V_{OP}$は，スケジューリングの結果，コントロールステップ $cs_{\sigma(v)+i}\,(0 \leq i \leq clk(v)-1)$ に割り当てられる．ここで $cs_{\sigma(v)}$は，vの先頭がスケジューリングされるコントロールステップを表す．

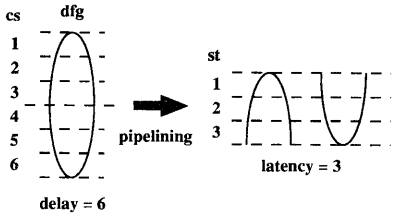

図 1: コントロールステップと状態の関係

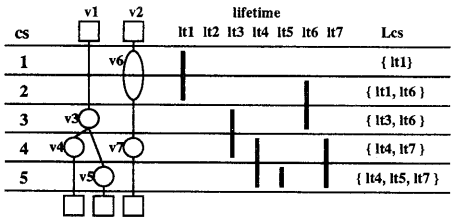

図 2: ライフタイムと L_{cs}

コントロールステップ $cs \in CS$にスケジューリングされるタイプ $ot \in OT$の演算節点の集合を $V_{cs,ot}$とする．

各節点 $v \in V_I \cup V_{OP}$は，ライフタイムを持つ．ライフタイムとは，データが出力されてから，そのデータが最後に用いられるまでの時間区間である．コントロールステップ $cs \in CS$を横切るライフタイムの集合を L_{cs}で表す．図2に，ライフタイムと L_{cs}の例を示す．

DSP を実現するために必要なタイプ $ot \in OT$の演算器の個数を

$$fn(ot) = \max_{st \in ST} \sum_{cs \in CS_{st}} |V_{cs,ot}|$$

とする．ここでCS_{st}は，状態が $st \in ST$となるコントロールステップの集合であり，

$$CS_{st_i} = \{cs_j \in CS \mid j \equiv i \pmod{latency}\}$$

となる．

DSP を実装するために必要なレジスタの個数を

$$rn = \max_{st \in ST} \sum_{cs \in CS_{st}} |L_{cs}|$$

とする．

リソースアロケーションを考慮した時間制約スケジューリング問題とは，

- データフローグラフ $G_{df}(V, E)$
- 演算器タイプ集合 OT
- レジスタ・コスト $cost_{reg}$
- $delay$
- $latency$

が与えられたとき，演算器とレジスタの総コスト

$$\sum_{ot \in OT} cost(ot) \cdot fn(ot) + cost_{reg} \cdot rn$$

が最小になるように，任意の演算節点 $v \in V_{OP}$のスケジューリング$\sigma(v)$ を決定することをいう．

3 リソースアロケーションを考慮した時間制約スケジューリング手法

提案するスケジューリング手法の概略を以下に示す．

(1) データフローグラフに対して ASAP，ALAP スケジューリング [1] を行い，各演算節点 $v \in V_{OP}$の初期移動範囲（initial mobiliy）を求める．

(2) 必要なハードウェア資源ができるだけ小さくなるように，各演算節点の mobility を徐々に減少させる．

(3) 全演算節点の mobility が 1 になるまで (2) を繰り返す．

ここで演算節点 $v \in V_{OP}$の mobility とは，vの先頭（1 クロック目）の割当てが可能なコントロールステップの総数である．本手法ではアルゴリズムの各イタレーションにおいて，各演算節点の割当てが可能なコントロールステップから，必要な演算器，およびレジスタのコストを見積る．イタレーションの初期の段階では，必要な演算器，レジスタのコストを正確に見積ることができないため，各演算節点のスケジューリングを 1 つのコントロールステップに決定することは難しい．よって本手法では，各演算節点に対し，最も割当てにふさわしくないコントロールステップを除いていくことにより，徐々に各演算のスケジューリングを決定し，より良い解を目指す．

3.1 演算節点の確率

本手法ではフォースディレクティッド法と同様に，各演算節点がそれぞれのコントロールステップに置かれ得る確率から必要な演算器のコストを見積る．

演算節点 $v \in V_{OP}$がコントロールステップ $cs \in CS$にスケジューリングされる確率 $p_{op}(v, cs)$ を，以下の式で定義する．

$$p_{op}(v, cs_i) = \frac{1}{mob(v)} \sum_{j=i-clk(v)+1}^{i} cns(v, cs_j)$$

ここで $cns(v, cs)$ は，演算節点 vの先頭に対して，コントロールステップ cs への割当てが可能ならば 1，そうでなければ 0 をとる 0-1 変数であり，$mob(v)$ は，演算節点 v の mobility を表す．

図 3に，演算節点の確率 $p_{op}(v, cs)$ の算出例を示す．ここで演算節点 vは，実行に 2 クロックサイクルを要し，v の先頭に対してコントロールステップ 2，3，5 への割当てが可能であるとする．

同じタイプ $ot \in OT$の演算節点の確率を，状態 $st \in ST$ ごとに合計したものを $ps_{op}(ot, st)$ とし，以下の式で表す．

$$ps_{op}(ot, st) = \sum_{cs \in CS_{st}, v \in V_{ot}} p_{op}(v, cs)$$

cs	cns(v,cs)		p_{op} (v,cs)
1	0		0
2	1		1/3
3	1	v	2/3
4	0		1/3
5	1		1/3
6	0		1/3
7	0		0

図 3: 演算節点の確率

ここで，V_{ot}は演算タイプが $ot \in OT$である演算節点の集合である．

アルゴリズムにおいて，DSP を実装するために必要なタイプ ot の演算器の個数は，$ps_{op}(ot, st)$ の最大値 $ps_{op}^{max}(ot)$ で見積られる．

3.2 ライフタイムの確率

本手法では演算器のコストだけではなく，データの一時保存に用いるレジスタのコストも積極的に最適化する．レジスタのコストも演算器のコストと同様に，各演算節点がそれぞれのコントロールステップに置かれ得る確率から見積る．

コントロールステップ $cs \in CS$が，節点 $v \in V_I \cup V_{OP}$ が出力するデータのライフタイムになる確率 $p_l(v, cs)$ は，$p_l^s(v, cs)$ と $p_l^d(v, cs)$ の積で表される．

$$p_l(v, cs) = p_l^s(v, cs) \cdot p_l^d(v, cs)$$

$p_l^s(v, cs)$ は，データを出力する演算節点から算出される確率であり，$p_l^d(v, cs)$ は，そのデータを用いる演算節点から算出される確率である．

確率 $p_l^s(v, cs)$ の算出

まず，$p_l^s(v, cs)$ を算出する．各外部入力節点 $v \in V_I$に対し，

$$p_l^s(v, cs) = 1 \quad \text{for} \quad \forall cs \in CS$$

とする．各演算節点 $v \in V_{OP}$に対し，

$$p_l^s(v, cs_i) = \begin{cases} 0 & (\text{if} \quad i < cns_t(v) + clk(v) - 1) \\ 1 & (\text{if} \quad i \geq cns_b(v) + clk(v) - 1) \\ \frac{1}{cns_{lt}(v, i - clk(v) + 1)} & (\text{otherwize}) \end{cases}$$

とする．$cs_{cns_t(v)}$は，演算節点 vの先頭の割当てが可能なコントロールステップのうち，最も時間の早いステップを表し，$cs_{cns_b(v)}$は，演算節点 vの先頭の割当てが可能なコントロールステップのうち，最も時間の遅いステップを表す．また $cns_{lt}(v, i)$ は，cs_iと等しいか，より時間の早いコントロールステップのうち，演算節点 vの先頭の割当てが可能なステップの総数であり，以下の式で表される．

$$cns_{lt}(v, i) = \sum_{j=cns_t(v)}^{i} cns(v, cs_j)$$

cs	p_l^s(v, cs)	$p_l^{d'}$(vc1, cs)	$p_l^{d'}$(vc2, cs)	p_l^d(v, cs)	p_l(v, cs)
1	0	1	1	1	0
2	0	1	1	1	0
3	1/3	1	1	1	1/3
4	2/3	1	1	1	2/3
5	2/3	1	1	1	2/3
6	1	1	1	1	1
7	1	1	2/3	1	1
8	1	2/3	1/3	7/9	7/9
9	1	1/3	0	1/3	1/3
10	1	0	0	0	0
11	1	0	0	0	0

図 4: ライフタイムの確率

図 4に，$p_l^s(v, cs)$ の算出例を示す．ここで演算節点 vは，実行に 2 クロックサイクルを要し，vの先頭に対してコントロールステップ 2，3，5 への割当てが可能であるとする．

確率 $p_l^d(v, cs)$ の算出

つぎに，$p_l^d(v, cs)$ を算出する．外部出力節点の子を持つ節点 $v \in V_I \cup V_{OP}$ に対し，

$$p_l^d(v, cs) = 1 \quad \text{for} \quad \forall cs \in CS$$

とする．外部出力節点の子を持たない節点 $v \in V_I \cup V_{OP}$に対し，$p_l^d(v, cs)$ は，関数 $p_l^{d'}(v, cs, n)$ の漸化式で表される．

$$p_l^d(v, cs) = p_l^{d'}(v, cs, |CHLD(v)|)$$

ここで，

$$\begin{aligned} p_l^{d'}(v, cs, n) &= p_l^{d'}(v, cs, n-1) + p_l^{d''}(chld(v, n), cs) \\ &\quad - p_l^{d'}(v, cs, n-1) \cdot p_l^{d''}(chld(v, n), cs) \\ p_l^{d'}(v, cs, 0) &= 0 \end{aligned}$$

$CHLD(v)$ は節点 vの子の集合を表し，$chld(v, n)$ は，節点 vの n 番目の子を表す．

$p_l^{d''}(v, cs)$ は，以下の式で定義される．

$$p_l^{d''}(v, cs_i) = \begin{cases} 1 & (\text{if} \quad i < cns_t(v)) \\ 0 & (\text{if} \quad i \geq cns_b(v)) \\ \frac{1}{cns_{gt}(v, i+1)} & (\text{otherwize}) \end{cases}$$

ここで $cns_{gt}(v, i)$ は，cs_iと等しいか，より時間の遅いコントロールステップのうち，演算節点 vの先頭の割当てが可能なステップの総数であり，以下の式で表される．

$$cns_{gt}(v, i) = \sum_{j=i}^{cns_b(v)} cns(v, cs_j)$$

```
main()
{
    データフローグラフ G_df(V,E), 演算器タイプ集合 OT,
    cost_reg, delay, latency を入力;
    ASAP スケジューリング;
    ALAP スケジューリング;
    ps_op(ot,st), ps_l(st) 初期化;
    schedule();
}
```

図 5: メイン・アルゴリズム

図 4に，$p_l^d(v,cs)$ の算出例を示す．ここで演算節点 vは，2 つの子 $vc1$，$vc2$ を持ち，$vc1$ の先頭に対してコントロールステップ 8，9，10 への割当てが可能であり，$vc2$ に対してコントロールステップ 7，8，9 への割当てが可能であるとする．$p_l^d(v,cs)$ は，$p_l^{d''}(vc1,cs)$ と $p_l^{d''}(vc2,cs)$ の和から積を引いたものとなる．

最終的なライフタイムの確率 $p_l(v,cs)$ は，$p_l^s(v,cs)$ と $p_l^d(v,cs)$ の積から計算される．

ライフタイムの確率を，状態 $st \in ST$ごとに合計したものを $ps_l(st)$ で表す．

$$ps_l(st) = \sum_{cs \in CS_{st}, v \in V_l \cup V_{OP}} p_l(v,cs)$$

アルゴリズムにおいて，DSP を実装するために必要なレジスタの個数は，$ps_l(st)$ の最大値 ps_l^{max}で見積られる．

3.3 アルゴリズム

図5に，本手法のメイン・アルゴリズムを示す．まず与えられたデータフローグラフに対して ASAP，ALAP スケジューリングを行い，演算節点の確率の合計 $ps_{op}(ot,st)$，およびライフタイムの確率の合計 $ps_l(st)$ の初期値を算出する．次にスケジューリング・アルゴリズム本体 $schedule()$ が呼び出される．

手続き *schedule*

図 6に，スケジューリング・アルゴリズム本体を示す．スケジューリング・アルゴリズムのイタレーションでは，フォースディレクティッド法のように，各演算を最も良いと思われるコントロールステップに一度に決めてしまうのではなく，最も悪いと思われるコントロールステップを 1 つずつ除いていくことにより，より良い解を目指す．演算節点の置かれ得るコントロールステップの範囲を狭める処理を，割当て考慮除外，あるいは mobility reduction という．

アルゴリズムにおいて，DSP を実現するために必要なタイプ ot の演算器の個数は，演算節点の確率の合計 $ps_{op}(ot,st)$ の最大値 $ps_{op}^{max}(ot)$ で見積られ，レジスタの個数は，ライフタイムの確率の合計 $ps_l(st)$ の最大値 ps_l^{max} で見積られる．よって，$ps_{op}^{max}(ot)$ for $\forall ot \in OT$，および

```
schedule()
{
    while (! 全ての演算がスケジューリングされた) {
        for (∀ot ∈ OT) {
            mobility reduction によって減少する見込みが
            ある ps_op(ot,st) の最大値 ps_op^max(ot) を算出;
            ps_op(ot,st) が最大となる時の st ∈ ST を
            st_op^max(ot) とする;
            評価関数 cri(ot) を算出;
        }
        cri_op^max = max_{ot∈OT} cri(ot);
        ot^max を cri(ot) が最大になる時の ot とする;
        ps_l^max = max_{st∈ST} ps_l(st);
        st_l^max を ps_l(st) が最大となる st とする;
        評価関数 cri_l を算出;
        if (cri_op^max < cri_l)
            reg_schedule(st_l^max);
        else
            fu_schedule(ot^max, st_op^max(ot^max));
    }
}
```

図 6: スケジューリング・アルゴリズム

ps_l^{max}ができるだけ小さくなるように mobility reduction を行う．すなわち $ps_{op}(ot,st)$ for $\forall ot \in OT$，$ps_l(st)$ ができるだけ平均化されるようにする．

アルゴリズムの各イタレーションでは，どのタイプの演算器，あるいはレジスタの総コストの最適化を目指して mobility reduction を行うかを決定する必要がある．そのため，$ps_{op}(ot,st)$，$ps_l(st)$ の平均化の程度を表す評価関数 $cri(ot)$，cri_lを導入し，それらのうち最も値の大きいものに対し，総コストが小さくなるように mobility reduction を行う．ここでは $cri(ot)$，cri_lとして以下のものを採用する．

$$cri(ot) = \begin{cases} cost(ot) \cdot \{ps_{op}^{max}(ot) - ps_{op}^{ave}(ot)\} \\ \qquad (\text{if} \quad ps_{op}^{max}(ot) - \lceil ps_{op}^{ave}(ot) \rceil > 0) \\ cost(ot) \cdot \{ps_{op}^{max}(ot) - ps_{op}^{ave}(ot)\} - Z \\ \qquad (\text{otherwize}) \end{cases}$$

$$cri_l = \begin{cases} cost_{reg} \cdot (ps_l^{max} - ps_l^{ave}) \\ \qquad (\text{if} \quad ps_l^{max} - \lceil ps_l^{ave} \rceil > 0) \\ cost_{reg} \cdot (ps_l^{max} - ps_l^{ave}) - Z \\ \qquad (\text{otherwize}) \end{cases}$$

ここで $ps_{op}^{ave}(ot)$ は，演算確率の合計 $ps_{op}(ot,st)$ の平均値であり，ps_l^{ave}は，ライフタイムの確率の合計 $ps_l(st)$ の平均値である．Zは十分に大きな整数[1]とする．

これらの評価関数は，$ps_{op}(ot,st)$ for $\forall ot \in OT$，および $ps_l(st)$ について，それぞれどの程度平均化する価値があるかを表している．もともと平均化されているものに対しては値が小さくなり，ハードウェアのコストが大きいほど値が大きくなる．また，平均化しても効果がないも

[1] 4 章の計算機実験では $Z = 1000$ とした．

```
fu_schedule(ot, st)
{
    ∀cs ∈ CS_st, ∀v ∈ V_ot において，p_op(v, cs) の 0 でない
    最小値を探索し，そのときの v を v_min，cs を cs_min とす
    る; /* ただし，cs_min は，演算節点の先頭に対応するコ
    ントロールステップとする */
    mobility_reduction(v_min, cs_min);
}
```

図 7: 手続き *fu_schedule*

のに関しては小さな値になる．例えば $ps_{op}^{max}(ot)$ が 3.9 で $ps_{op}^{ave}(ot)$ が 3.1 である場合，タイプ ot の演算器は必ず 4 個必要になるので，これ以上平均化しても効果がない．

手続き *schedule* ではこれらの評価関数の値によって，手続き *fu_schedule* または *reg_schedule* を呼び出し，演算器，あるいはレジスタのコスト最適化をめざして mobility reduction を行う．

手続き *fu_schedule*

fu_schedule(ot, st)（図 7）は，タイプ ot の演算器数の最適化を目指すために，タイプ ot，状態 st における演算節点の確率の合計 $ps_{op}(ot, st)$ が小さくなるように mobility reduction を行う手続きである．この手続きでは，タイプが ot となる任意の演算節点 $v \in V_{ot}$，状態が st となる任意のコントロールステップ $cs \in CS_{st}$ において，v が cs に割り当てられる確率が最も小さいもの（0 は除く）を探索し，手続き *mobility_reduction* により割当て考慮を除外する．確率の最も小さいものを除外することにより，状態 st における確率の合計 $ps_{op}(ot, st)$ が減少すると同時に，他の状態における確率の合計に与える影響を最小限に抑えることができる．

手続き *reg_schedule*

reg_schedule(st)（図 8）は，レジスタ数の最適化を目指すために，状態 st におけるライフタイムの確率の合計 $ps_l(st)$ が小さくなるように mobility reduction を行う手続きである．図において，$idx(cs_i) = i$ であるとする．この手続きは，状態が st となる任意のコントロールステップ $cs \in CS_{st}$，データを出力する任意の節点 $v \in V_I \cup V_{OP}$ において，cs が v のライフタイムになる確率が最も小さいもの（0 は除く）を探索し，手続き *mobility_reduction* により，v のライフタイムを上下どちらか適当な方向から縮める．確率がもっとも小さいものを探索することにより，状態 st における確率の合計 $ps_l(st)$ が減少すると同時に，他の状態における確率の合計に与える影響を最小限に抑えることができる．

図 9に例を示す．今，ライフタイムの確率の合計 $ps_l(st)$ が最大となる状態が 9 であり，$delay = latency$（非パイプライン）とする．コントロールステップ 9 における確率がもっとも小さい（0 は除く）ライフタイムを探索し，そ

```
reg_schedule(st)
{
    ∀cs ∈ CS_st, ∀v ∈ V_I ∪ V_OP において，p_l(v, cs) の 0 で
    ない最小値を探索し，そのときの v を v_min，cs を cs_min
    とする;
    v_min の子 ∀vc ∈ CHLD(v_min) のなかで，cns_b(vc) が最
    も大きいものを探索し，vc_b とする;
    if (idx(cs_min) − cns_t(v_min) < cns_b(vc_b) − idx(cs_min))
        mobility_reduction(v_min, cs_{cns_t(v_min)});
    else
        mobility_reduction(vc_b, cs_{cns_b(vc_b)});
}
```

図 8: 手続き *reg_schedule*

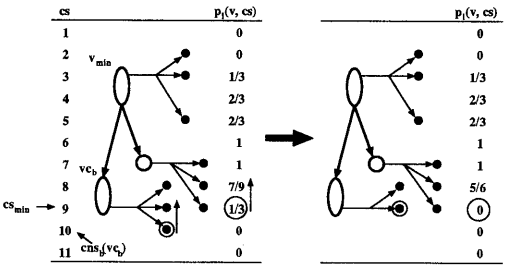

図 9: *reg_schedule* の例

のデータを出力するノードを v_{min} とする．v_{min} のライフタイムを下から縮めるために，v_{min} の子 vc_b に対して，コントロールステップ 10 への割当て考慮を除外する．その結果，コントロールステップ 9 におけるライフタイムの確率は 1/3 から 0 に減少する．

手続き *mobility_reduction*

mobility_reduction(v, cs)（図 10）は，演算節点 v のコントロールステップ cs への割当て考慮を除外する手続きである．割当て考慮除外に際して，演算節点の確率の合計 $ps_{op}(ot, st)$，ライフタイムの確率の合計 $ps_l(st)$ を更新する．割当て考慮を除外した結果，v の依存関係にある演算節点の mobility に影響を与える場合，それらの演算節点に対しても mobility reduction 操作を行う．

図 11に，依存関係にある演算節点の mobility への影響の例を示す．図の DFG において，節点 v_1 はコントロールステップ 1,3,4，節点 v_2 はコントロールステップ 2,3,5，節

```
mobility_reduction(v, cs)
{
    演算節点 v の cs への割当て考慮を除外;
    ps_op(ot, st)，ps_l(st) を更新;
    if (v と依存関係にある演算節点の mobility に影響) {
        v と依存関係にある演算節点に対し，mobility reduc-
        tion 操作を行う;
    }
}
```

図 10: 手続き *mobility_reduction*

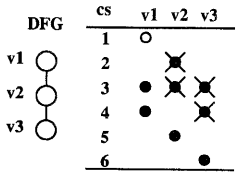

図 11: 依存関係にある演算節点への影響

表 1: ハードウェアリソース・ライブラリ

library	resource	cost	#cycle
1	add	5	1
	multiply	10	1
2	add	5	1
	multiply	15	2
3	add	5	1
	multiply	10	2
	register	5	–

点 v_3はコントロールステップ 3,4,6 への割当てが可能であるとする．今，節点 $v1$ のコントロールステップ 1 への割当て考慮を除外する場合，データ依存制約により，節点 $v2$ のコントロールステップ 2，3 への割当て，節点 $v3$ のコントロールステップ 3，4 への割当てが不可能となり，それらについても割当て考慮を除外する必要がある．

4 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，5th order Digital Wave Filter (DWF) [13]，および Inverse Discrete Cosine Transformation (IDCT) [14] のデータフローグラフに適用した結果を報告する．

表 1に，実験に使用したハードウェアリソース・ライブラリを示す．cost はハードウェアコストを表し，#cycle は演算に必要なクロックサイクル数を示す．マルチサイクルの乗算器は，パイプライン化されていないものとする．

表 2に，提案手法によるスケジューリング結果と，BEM II[15] を用いたスケジューリング結果の比較を示す．表において delay は与えた実行ステップ数，#add，#multiply は，それぞれ必要な加算器，乗算器の数を表す．BEM II は，0-1 整数線形計画問題を 2 分決定グラフ（BDD[16]）を用いて解くことができるソフトウェアツールであり，スケジューリングは最適解を保証する．本結果は，レジスタを考慮せず，パイプライン化を行っていない．ライブラリは 1 を使用した．提案手法では，BEM II による結果と等しい結果が得られており，DWF の合成は 30 倍以上高速であることがわかる．なお，BEM II による IDCT のスケジューリングでは，最適解が多く，結果の列挙に時間がかかるため，CPU 時間の測定は不可能であった．

表 2: 実験結果 1（lib=1, レジスタ考慮なし）

system	Ours		BEM II	
dfg	DWF	IDCT	DWF	IDCT
delay	14	6	14	6
#add	3	8	3	8
#multiply	2	5	2	5
CPU time[s]	0.09	0.07	3.19	–

表 3: 実験結果 2（IDCT, lib=3, レジスタ考慮あり）

latency	delay	#FU add	#FU mult	#reg	total cost	CPU time [s]
2	8	14	15	39	415	0.08
3	9	10	11	31	315	0.07
4	8	8	8	20	220	0.04
5	10	8	6	21	205	0.12
6	12	4	5	22	180	0.12
7	7	8	8	15	195	0.08
8	8	8	7	15	185	0.08
9	9	8	5	13	155	0.08
10	10	6	4	14	140	0.13
11	11	8	4	14	155	0.07
12	12	4	4	14	130	0.10
13	13	4	3	13	115	0.16
14	14	3	3	14	115	0.18

表 4: 実験結果 3（IDCT, lib=3, レジスタ考慮なし）

latency	delay	#FU add	#FU mult	#reg	total cost	CPU time[s]
2	8	14	15	48	460	0.08
3	9	9	11	33	320	0.12
4	8	8	8	21	225	0.11
5	10	6	6	24	210	0.13
6	12	4	6	26	210	0.12
7	7	8	8	16	200	0.07
8	8	8	7	15	185	0.05
9	9	8	5	13	155	0.07
10	10	6	4	14	140	0.09
11	11	8	4	15	155	0.07
12	12	6	4	16	150	0.09
13	13	4	4	14	130	0.14
14	14	5	3	15	130	0.15

表 3，表 4に，提案手法による，latency を変化させた場合の IDCT のスケジューリング結果を示す．表 3にはレジスタを考慮した場合，表 4にはレジスタを考慮しない場合の結果を示す．表において，latency，delay はそれぞれ与えたレイテンシ，実行ステップ数を表し，#FU，#reg は，それぞれ必要な演算器（加算，乗算），レジスタの数を表し，total cost は演算器，レジスタの総コストを表す．IDCT のデータフローグラフにおけるクリティカルパス長は 7 であるため，latency が 6 以下のものは，パイプライン化されている．ライブラリは 3 を使用した．レジスタを考慮した場合は，考慮しない場合に比べて，必要なレジスタ数が平均 10%程度減少しており，total cost も平均 5%程度減少していることがわかる．実行時間が高速なた

表 5: 実験結果 4（DWF, ours: lib=2, レジスタ考慮なし）

system	Ours			FDS				FDLS			ALPS		
delay	17	18	21	17	18	19	21	17	18	21	17	18	21
#add	3	2	2	3	3	2	2	3	2	2	3	2	2
#multiply	3	2	1	3	2	2	1	3	2	1	3	2	1
CPU time	0.05	0.08	0.12	between				between			within tens		
	sec			2 & 6 min				1 & 2 min			of sec		

め，本手法により latency と total cost とのトレードオフを効率良く行うことができる．

表 5に，提案手法による DWF のスケジューリング結果（Ours）と，フォースディレクティッド法（FDS）[7]，フォースディレクティッド・リストスケジューリング（FDLS）[7]，および ILP 法に基づく ALPS system[5] によるスケジューリング結果との比較を示す．パイプライン化は行っていない．提案手法はレジスタを考慮せず，またライブラリは 2 を使用した．フォースディレクティッド・リストスケジューリングは資源制約スケジューリング手法である．提案手法は，他手法と同程度の結果を 1 秒以下で実現しており，他手法に比較して，優位性が実験的に明らかになった．

5 むすび

本稿では，演算器とレジスタ双方のハードウェアコストの最小化を目的とし，マルチサイクル演算器の考慮，パイプライン化 DSP の合成を可能とした時間制約スケジューリング手法を提案した．提案手法を 5th Order DWF，および IDCT の合成に適用した結果，1 秒以下でほぼ最適解に近い解を得ることができることを確認した．

今後の課題は，マルチプレクサやバスのコストの考慮，イタレーション間データ依存のあるデータフローグラフのパイプライン化 [9],[10] への適用が考えられる．

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: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[2] J. H. Lee, Y. C. Hsu, and Y. L. Lin, "A New Integer Programming Formulation for The Scheduling Problem in Data Path Synthesis," in *Proc. ICCAD-89*, pp. 20–23, Nov. 1989.

[3] C. T. Hwang, Y. C. Hsu, and Y. L. Lin, "Optimum and Heuristic Data Path Scheduling Under Resource Constraints," in *Proc. 27th DAC*, pp. 65–70, June 1990.

[4] H. Achatz, "Extended 0/1 Formulation for the Scheduling Problem in High-Level Synthesis," in *Proc. EURO-DAC '93*, pp. 226–231, Sept. 1993.

[5] C. T. Hwang, J. H. Lee, and Y. C. Hsu, "A Formal Approach to the Scheduling Problem in High-Level Synthesis," *IEEE Trans. CAD*, Vol. 10, pp. 464-475, Apr. 1991.

[6] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. CAD*, Vol. 8, No. 6, pp. 661–679, June 1989.

[7] P. G. Paulin and J. P. Knight, "Algorithms for High-Level Synthesis," *IEEE Design & Test*, pp. 18–31, Oct. 1989.

[8] R. J. Cloutier and D. E. Thomas, "The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm," in *Proc. 27th DAC*, pp. 71–76, June 1990.

[9] C. T. Hwang, Y. C. Hsu, "PLS: A Scheduler for Pipeline Synthesis," *IEEE Trans. CAD*, Vol. 12, No. 9, pp. 1279–1286, Sept. 1993.

[10] C. Y. Wang, K. K. Parhi, "High-Level DSP Synthesis Using Concurrent Transformations, Scheduling, and Allocation," *IEEE Trans. CAD*, Vol. 14, No. 3, pp. 274–295, March 1995.

[11] S. Amellal and B. Kaminska, "Functional Synthesis of Digital Systems with TASS," *IEEE Trans. CAD*, Vol. 13, No. 5, pp. 537–552, May 1994.

[12] E. F. Girczyc, "Loop Winding - A Data Flow Approach to Functional Pipelining," in *Proc. ISCAS '87*, Vol. 2, pp. 382–385, May 1987.

[13] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms," *VLSI and Modern Signal Processing Algo.*, S. Y. Kung, H. J. Whitehouse, and T. Kailath, ed. pp. 257–264, 1985.

[14] P. H. N. de With, "Motion-Adaptive Intraframe Transform Coding of Video Signals," *Philips J. Res. 44*, pp. 345–364, 1989.

[15] S. Minato, "BEM II: An Arithmetic Boolean Expression Manipulator Using BDDs," *IEICE Trans. Funadamentals*, Vol. E76-A, No. 10, pp. 1721–1729, Oct. 1993.

[16] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comp.*, Vol. C-35, No. 8, pp. 677–691, Aug. 1986.