設計自動化 68－2
(1993. 10. 28)

条件分岐制御のあるデータパスのスケジューリング手法

山田 晃久† 山崎 年樹†† 石浦 菜岐佐†† 神戸 尚志† 白川 功††

† シャープ株式会社生産技術研究所
〒632 奈良県天理市櫟本町 2613-1

†† 大阪大学工学部情報システム工学科
〒565 大阪府吹田市山田丘 2-1

あらまし 本稿では，高位合成におけるデータパスのスケジューリング問題に対して，条件分岐を含む動作記述の最適な時間制約スケジューリングを求める手法を提案する．動作記述中に条件分岐を含む場合には，条件判断の演算と条件分岐中の演算のどちらを先に行なうかにより，必要な制御ステップ数や資源数が異なってくるが，従来の手法では，これらを考慮して最適な解を求めることはできなかった．本手法はスケジューリング問題を 0-1 整数非線形計画問題として定式化することにより，最適解を求める．この制約式をブール式で表現し，二分決定グラフを用いて充足問題を解くことにより，効率よく解を求める．実験の結果いくつかの動作記述に対して，従来手法よりも優れた結果を得ることができた．

和文キーワード 高位合成，データパス・スケジューリング，0-1 整数非線形計画問題，二分決定グラフ

A Data-Path Scheduling Method For Conditional Resource Sharing

Akihisa YAMADA† Toshiki YAMAZAKI†† Nagisa ISHIURA†† Takashi KAMBE† Isao SHIRAKAWA††

† Production Technology Laboratories, SHARP Corporation
2613-1 Ichinomoto, Tenri-shi, Nara 632, Japan

†† Department of Information Systems Engineering, Faculty of Engineering, Osaka University
2-1 Yamadaoka, Suita-shi, Osaka 565, Japan

Abstract This paper presents a new data-path scheduling method for behavior descriptions with nested conditional branches. The main difficulty in this problem is that the total clock cycles and the number of functional units may vary depending on the priority order between the operations to compute branch conditions and those within the branches. We formulate the problem as a 0-1 integer nonlinear programming, whose objective function is to minimize the total cost of functional units under a given timing constraint. We obtain optimal solutions efficiently by representing the constraints in terms of Boolean formulas and by solving the satisfiability problem using a binary decision diagram based Boolean function manipulator. Experimental results show that our method achieves better solutions than other existing methods.

英文 key words high-level synthesis, data-path scheduling, 0-1 integer nonlinear programming, binary decision diagram

1 はじめに

近年の LSI の大規模化にともない, 設計コスト削減, 設計期間短縮が重要な課題となっている. 短期間で大規模な回路を設計する際の有効な手法としてトップダウン設計手法が注目を浴びている. トップダウン設計手法とは, まずハードウェアの動作記述を行ない, その段階で動作の検証を行なった後, 下位の設計を進めていくものである. 高位合成はこのトップダウン設計における重要な技術の一つであり, 動作レベルの記述からレジスタトランスファレベルの回路を合成する技術である. 本文では, 高位合成において動作記述中の演算を実行する順序を決定し, 制御ステップに割り当てるスケジューリング問題に対して, 動作記述中に条件分岐がある場合にも最適な解を与えるような手法について考察する.

動作記述中に条件分岐がある場合のスケジューリングでは

(1) まず分岐条件を判断する演算を行なった後, その結果に従って分岐中の演算を実行する

(2) すべての分岐中の演算を実行後, 分岐条件の結果に基づいて演算結果を選択する

のいずれかで行なわれていた. (1) では異なる分岐条件で実行される演算間で演算器の共有ができ, 演算器の個数を少なくすることがるできるが, 分岐条件を判断する演算が終了するまで分岐中の演算が実行できず, 制御ステップ数が多くなる場合がある. 逆に (2) では分岐中の演算は分岐条件を判断する演算の実行を待たずに実行できるため制御ステップ数を少なくすることができるが, 分岐中の演算間で演算器を共有することができず, 演算器の個数が多くなる場合がある. Lee[1] らはスケジューリング問題を整数線形計画問題として定式化し, 与えられた制御ステップ数のもとで演算器のコストを最小にする解を求める手法を提案した. この手法では, すべての解空間を探索することになるので, 条件分岐がない場合には最適解を得ることができるが, 条件分岐がある場合に (1) を前提としているため最適解が得られるとは限らない. Kim[2] らは, 演算器を共有できる演算を 1 つの演算にまとめるという手続きを, 条件分岐に対して階層的に適用し, 条件分岐のない形に変形し, それをスケジューリングした後, 逆順に戻すという手法を提案した. しかし, これも (1) を前提としており, 最適解が得られるとは限らない. これらの手法の問題点は, 分岐条件を判断する演算と分岐中の演算の実行順序を予め決定し, スケジューリングを行なっていることに起因する. 実際は, 分岐条件が決定された後であれば, 分岐中の異なる分岐条件で実行される演算は演算器を共有でき, 分岐条件が決定されていない時にはたとえ異なる分岐条件で実行される演算でも演算器を共有できないので, これをスケジューリング中に扱うことができれば上の問題を解決することができる. Wakabayashi 等 [3] は, 条件ベクタという概念を導入し, (1), (2) の前提を置かずに, 演算間で演算器を共有できるかどうかという関係を処理中に更新しながらスケジューリングする手法を提案した. この手法は他の手法に比べて, 良い解を得ることができるが, 発見的手法であるため必ず最適解が求まるとは限らず, また条件分岐の構造によってはそのままでは適用できないものもある.

我々は動作記述中に条件分岐がある場合にも, 与えられた制御ステップ数内で, 最小の演算器コストでスケジューリングを行なう, 時間制約スケジューリング手法を提案する. 本手法は [3] と同様, (1), (2) の前提を置かずにスケジューリングを行なうもので, 0-1 整数非線形計画問題として定式化することにより最適な解を得ようとするものである. 本手法ではこの制約式をブール式で表現し, 二分決定グラフ (BDD)[4] を用いて充足問題を解くことにより, 効率よく解を求める. 本手法は, 解空間を全探索できるので, [3] よりも良い解を見つけることができる. またどのような条件分岐の構造に対しても適用できる.

本文では, まず 2 章で高位合成におけるスケジューリング問題と, 条件分岐がある場合の演算間の演算器の共有について述べる. 3 章では本手法で用いるフローグラフについて述べた後, 条件分岐がある場合のスケジューリング問題をブール式を用いて定式化する. 4 章では実験結果により本手法の有効性を示す.

2 高位合成におけるスケジューリング問題

2.1 スケジューリング問題

高位合成におけるスケジューリング問題とは, コントロール・データフローグラフ等で与えられたハードウェアの動作に対し, これを実行するための制御ステップ数や演算器の個数等を考慮して, 各演算が実行される順序を決定するものである.

本文では, ハードウェアの動作を表現するものとして以下のようなコントロール・データフローグラフ [5] (以後 CDFG と呼ぶ) $G = (V, E)$ を考える. 節点の集合 V は互いに素な 4 つの部分集合 O, B, M, L ($O \cup B \cup M \cup L = V$) からなる. $o_i \in O$は, 動作記述中の演算に対応し, これを演算節点と呼ぶ. $b_i \in B$は, 動作記述中の条件分岐の始まりを表し, 分岐節点と呼ぶ. $m_i \in M$は, 条件分岐内で生成される値に対応し, マージ節点と呼ぶ. 動作記述の条件分岐は 1 つの分岐節点と 0 個以上のマージ節点によって表される. $l_i \in L$ は CDFG の入出力にある変数や定数を表し, 変数節点と呼ぶ. 枝の集合 E は互いに素な 3 つの部分集合 $S, P, W (S \cup P \cup W = E)$ からなる. 節点 $v_i \in V$ から 節点 $v_j \in V$ に値が渡される時, かつその時に限り有向枝 $(v_i, v_j) \in S$ を設ける. これを S-枝と呼ぶ. 条件分岐中の演算が, 条件が真の時に実行されるパス (以後, 真のパスと呼ぶ) 中にあるか, 偽の時に実行されるパス (以後, 偽のパスと呼ぶ) 中にあるかを示すために, 分岐節点 b_i と, 条件分岐中のパスの最初の演算節点 o_j の間に有向枝 $(b_i, o_j) \in W$ を設ける. この枝を W-枝と呼ぶ. W-枝にはその枝が真・偽どちらのパスを表すかを示すために, 真のパスを表す時は T, 偽のパスを表す時は F の属性をつける. 分岐節点 b_i と対応するマージ節点 m_jの間に有向枝 $(b_i, m_j) \in P$ を設ける. この枝を P-枝と呼ぶ. 図 1(a) の動作記述に対する CDFG は同図 (b) のようになる. 図中 S-枝を細実線で, W-枝を破線で, P-枝を太実線で示している. また見やすくするために変数節点は描かずに, 枝の始点, あるいは終点にその変数や定数を記している.

この CDFG に対して $+, -, >$ を ALU で実行することにし, スケジューリングした結果は例えば図 1(c) のようになる. 図 1(c) の各列は CDFG 中の演算を表し, 各行が制御ステップを表す. 制御ステップとは, データパスを制御する回路の状態が遷移する時間の最小単位で, 1 クロックを 1 制御ステップとすることが多い. 図 1(c) では, o_5 と o_9, o_6 と o_{10} は条件分岐 b_1 の異なる実行条件で実行されるので 1 つの ALU を共有できる. また o_5 と o_7 も 条件分岐 b_2 の異なる実行条件で実行されるので 1 つの ALU を共有できる. さらに o_2 の結果は o_3 の結果が真である時のみ必要となるので, o_3 の結果が偽である時には実行する必要がないため o_8 と ALU を共有することができる. したがって, このスケジューリングで必要となる ALU は 2 つとなる. 制御ス

筆者の一人山田は現在大阪大学大学院後期課程に在学中.

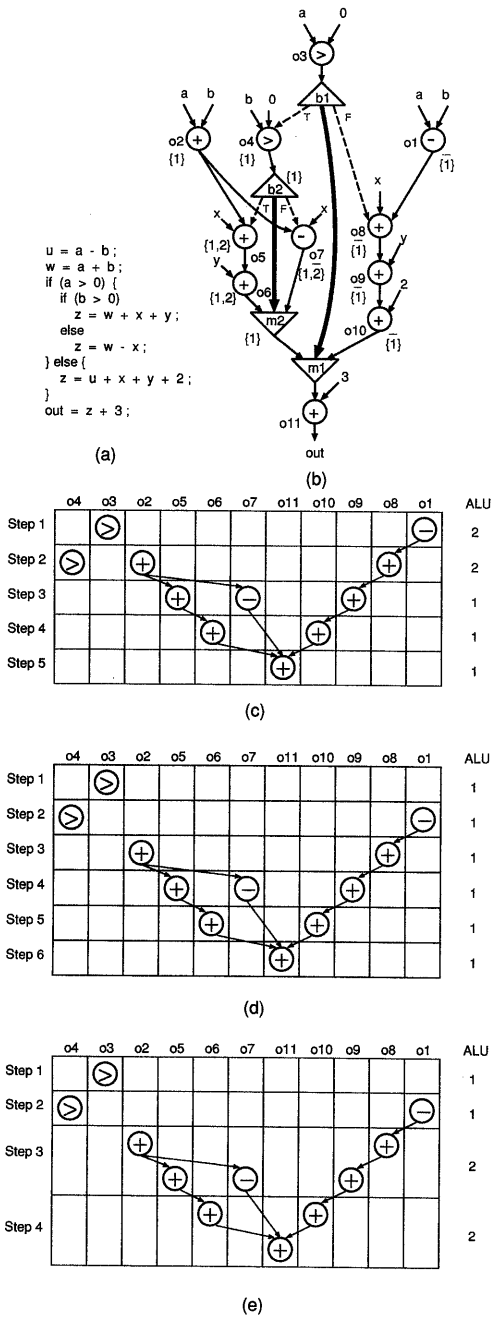

図 1: CDFG とスケジューリングの例 (a) 動作記述 (b) CDFG, (c) スケジューリング結果 (制御ステップ数 5, ALU 数 2), (d) スケジューリング結果 (制御ステップ数 6, ALU 数 1), (e) スケジューリング結果 (演算のチェインを行なう場合).

テップ数を 6 にすると, 図 1(d) のように ALU 1 つ で実現することができる. このようにスケジューリングの問題を解く際には資源数と制御ステップ数の 2 つを考慮する必要がある. この 2 つのどちらを制約にして, どちらを最小化するかによってスケジューリング手法を 2 つに分けることができ, 1 つは資源数を制約として与えて, 制御ステップ数を最小化するもの (資源制約スケジューリング) であり, もう 1 つは制御ステップ数を制約として与えて資源数を最小化するもの (時間制約スケジューリング) である.

本文では後者の時間制約スケジューリングを考える. ここでは, 資源として演算器のみを考え, レジスタ等の記憶コストや接続コスト等は考えないことにする. さらに, 各演算に対して次のような前提を置く:

- 各演算を実行する演算器の種類は既に決定されているものする.
- 複数の制御ステップにまたがる演算は考えない.

1 制御ステップ内で複数の演算を続けて実行する演算のチェインは扱うことにする. 例えば 1 制御ステップの長さが 100 ns である時, 実行 (遅延) 時間が 40 ns の ALU を使えば, 1 制御ステップ内で最大 2 つの演算を実行できる. この場合, チェイン数が 2 であると書くことにする. 例えば図 1(b) の CDFG に対してチェイン数を 2 としてスケジューリングを行なうと, 図 1(e) のように制御ステップ数を 4 とすることができる.

2.2 相互排他演算間の演算器の共有

2 つ以上の演算が同時に実行されない時, これらの演算は互いに排他であるといい, これらの演算を相互排他演算と呼ぶ. 同じ型の演算器で実行される相互排他演算は 1 つの演算器を共有することができ, 必要な演算器数を減らすことができる. 演算間の排他性には,

1. 異なる制御ステップに割り当てられた演算間の排他性
2. 同じ制御ステップに割り当てられたが, 実行条件が排他な演算間の排他性

の 2 つがある [6]. 条件分岐がある場合に問題となるのは 2 である.

動作記述中で実行条件が排他である演算が必ずスケジューリング後に互いに排他になるわけでない. 例えば, 図 1 の o_5 と o_9 は 動作記述中では実行条件が排他であるが, スケジューリングで条件分岐 $b1$ の分岐条件を判定する演算 (以後単に条件演算と呼ぶ) o_3 が実行される前の制御ステップに割り当てられると, 互いに排他ではなくなる. 例えば 図 1 (c) で o_3 が Step 3 に割り当てられると o_5 と o_9 は互いに排他ではなくなる. このように, 演算間の排他性は条件演算と条件分岐中の演算の先行関係に依存する. ここでは, 動作記述中で実行条件が排他的であることを静的に排他と呼び, 条件演算の実行の前後関係によって排他的となることを動的に排他と呼んで区別することにする. 静的に互いに排他な演算間で演算器を共有するためにはこれらの演算を条件演算よりも後の制御ステップに割り付ければよい. しかし, これでは, 条件分岐中の演算が実行できるにもかかわらず, 条件演算を待たなければならない状況では制御ステップ数が増える可能性がある. したがって, スケジューリングを効率良く行なうためには演算間の動的な排他性を考慮してスケジューリングを行なう必要がある.

本節では動的な排他性を考える準備として静的な排他性について考える. 静的な排他性には図 1(b) の o_1 と o_7 のように, ある演算の実行結果が, ある分岐中で使用されないために現れる排他性 (データ依存関係に基づく排他性) と,

o_5 と o_9 のように動作記述中に陽に異なる実行条件で実行するように記述されたものとがある [6]. これらは, [5] で提案されているように CDFG 上で節点に実行条件をタグ付けすることにより見い出すことができる. CDFG 中の条件分岐を $b_1, b_2, ..., b_q$ とするとき, 節点 v_i のタグ $tag(v_i)$ とは $\{1, \overline{1}, 2, \overline{2}, ..., q, \overline{q}\}$ の部分集合であり, v_i が b_l において, 真のパスに含まれるときには $l \in tag(v_i)$, 偽のパスに含まれるときには $\overline{l} \in tag(v_i)$ を満たすものである. このとき, $l \in tag(v_i), \overline{l} \in tag(v_j)$ であれば, v_i と v_j は静的に互いに排他である. 例えば, 図 1 では, o_5 のタグは $\{1, 2\}$ であり, o_8 のタグは $\{\overline{1}\}$ であるから o_5 と o_8 は静的に互いに排他であることがわかる.

このタグを付けた後, S-枝を逆向きに探索することによりデータ依存関係に基づく排他性についても見い出すことができる. すなわち各演算節点について, その演算節点を始点とする S-枝 のすべての終点のタグの積集合をその演算のタグとすることにより, 上と同じように排他性を見い出せる. 例えば, 図 1(b) の o_2 の子は $o_5(tag(o_5) = \{1, 2\})$ と $o_7(tag(o_7) = \{1, \overline{2}\})$ なので, そのタグは $\{1, 2\} \cap \{1, \overline{2}\} = \{1\}$ となり, タグが $\{\overline{1}\}$ である o_1 とは静的に互いに排他であることがわかる. このようにしてタグ付けされた演算はタグが表す条件分岐のパス中に演算があるものとして考えることができる.

3 スケジューリング問題のブール式による定式化

[1] で定式化された整数線形計画問題は, 演算が制御ステップに割り当てられたかどうかを 0-1 の値を取る変数で, 必要となる各演算器の総数を整数変数で表し,

1. フローグラフ中の全演算が割り当てられる
2. フローグラフ中の全演算が実行可能な順序に保たれる
3. 各制御ステップで必要な各演算器の個数がその演算器の総数を越えない

という 3 つの制約の元で必要となる演算器のコストを最小にする変数の値の組を求めるものである. この手法は解空間を全探索できるので, 条件分岐がない場合には最適な解を求めることができる. しかし条件分岐がある場合は, 動的な排他性を考慮していないため, 最適解が得られるとは限らない. [7] ではこの定式化における演算の制御ステップへの割当を表す変数が 0-1 の値をとることに注目し, 1 ～ 3 の制約式をブール式で表し, BDD[4] を用いて効率よく解を求めている. 我々は [1][7] の定式化に基づいて, 条件分岐中の動的な排他性を考慮して, スケジューリング問題をブール式を用いて定式化する.

以下本章では, まず本スケジューリング手法の入力となるフローグラフについて述べ, その後でスケジューリング問題のブール式による定式化を行なう.

3.1 スケジューリングのためのフローグラフ表現

本文のスケジューリング手法は, 2.1 節で述べた CDFG から次の情報を抽出したフローグラフを入力として用いる. スケジューリング中に必要となる演算間の情報は,

1. 演算間の先行関係
2. 演算間の相互排他性

である.

1, 2の情報を持つフローグラフ $G_f = (O, E_f)$ を以下のように定義する. 節点集合 O は CDFG 中の演算節点集合であり, 各節点 o_i は CDFG 上で付けられた実行条件のタグ $tag(o_i)$ を持つ. 枝集合 E_f は互いに素な 2 つの部分集合 H, T $(H \cup T = E_f)$ からなる. 節点 o_i で生成された値が節点 o_j で使用される時, かつその時に限り有向枝 $(o_i, o_j) \in T$ を設け, $o_i \rightarrow o_j$ と書くことにする. 節点 o_i から T に属する枝を通って到達可能な節点を o_i の子孫と呼ぶことにする. 節点 o_i を始点とし T に属する枝を通る有向道の中で, 節点 o_j が, 節点 o_i と同じ制御ステップに割り付けることができない最も近い子孫である時, かつその時に限り有向枝 $(o_i, o_j) \in H$ を設け, $o_i \Rightarrow o_j$ と書くことにする. $o_i \rightarrow o_j$ は, o_j を o_i より前の制御ステップに割り付けた場合に正しい演算結果が得られないことを表し, $o_i \Rightarrow o_j$ は, o_j が o_i より後の制御ステップに割り付けられなければならないことを表す. 図では $o_i \rightarrow o_j$ を細い実線で, $o_i \Rightarrow o_j$を太い実線で示す. このように 2 種類の枝を用いるのは演算のチェインに対応するためである (この考え方は [1] で提案されたものと同じである).

このフローグラフ G_f では, 枝が演算間の先行関係を表し, 節点のタグが演算間の相互排他性を表している.

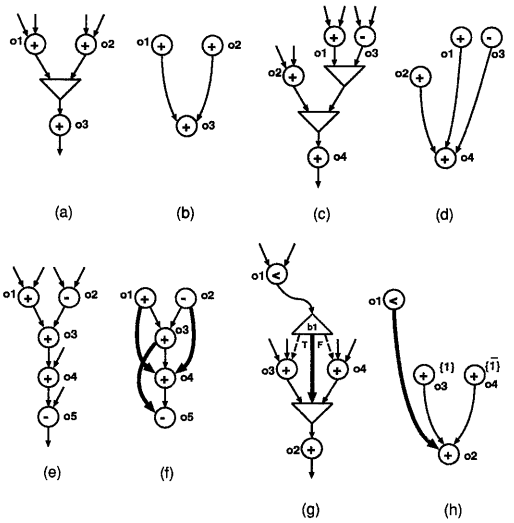

図 2: CDFG からフローグラフへの変換.

フローグラフ G_f を CDFG から作成する手順を述べる. 演算間の先行関係について次の 2 つを考えなければならない.

(i) データ依存によるもの

(ii) コントロール依存によるもの

(i) は CDFG の S-枝に相当するものであり, (ii) は条件演算とその条件分岐中で生成された値を使用する演算との間の関係である.

まず CDFG で $(o_i, o_j) \in S$のとき, $o_i \rightarrow o_j$とする. CDFG 中のマージ節点 m_i については, m_i を終点とする S-枝の始点である演算節点 o_j で生成された値が, m_i を始点とする S-枝 の終点である演算節点 o_k で使用されるので $o_j \rightarrow o_k$ とする. 例えば図 2(a) の CDFG に対しては $o_1 \rightarrow o_3$, $o_2 \rightarrow o_3$ となる (同図 (b)). マージ節点間に S-枝がある場合には, 演算節点 o_j からマージ節点と S-枝のみを通って演算節点 o_k に到達できる時, 枝 $o_j \rightarrow o_k$ を設ける. 図 2(c) の CDFG の場合には同図 (d) のようになる. この段階で, o_i と同じ制御ステップで実行することができない子孫のうち最も近い

子孫 o_j に対して $o_i \Rightarrow o_j$ とする．例えば演算のチェインを行なう場合に，1制御ステップが 100 ns，各演算の実行時間が 50ns とした時，図 2(e) の CDFG に対するフローグラフは，図 2(f) のようになる (ここでは，簡単にするためにデータの転送に要する時間は無視して考えている)．

次に (ii) による枝を設ける．2.2 節で述べたように，条件演算と条件分岐中の演算の間には先行関係はないが，条件分岐中で生成された値を使用する演算については，条件演算の終了した後の制御ステップでないと，データの値が確定していないため実行することができない．したがって，条件演算 o_i と条件分岐中で生成された値を使用する演算 o_j の間に枝 $o_i \Rightarrow o_j$ を設ける．例えば図 2(g) の CDFG に対するフローグラフは，図 2(h) のようになる．

図 1(b) の CDFG に対するフローグラフは，演算のチェインをしない場合と，チェイン数を 2 とした場合，それぞれ図 3(a), (b) のようになる．図では，$o_i \rightarrow o_j$ かつ $o_i \Rightarrow o_j$ の時は，$o_i \Rightarrow o_j$ のみで表している．

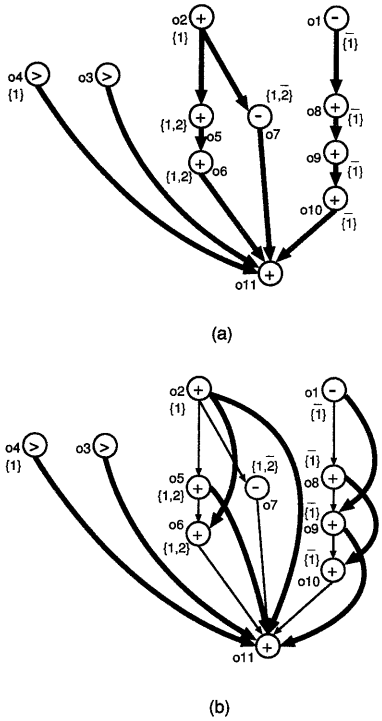

図 3: 図 1 (b) に対するフローグラフ: (a) チェインをしない場合，(b) チェイン数が 2 の場合．

以上のようにして作成したフローグラフ G_f はスケジューリングに必要な情報を持っているが，3.3節で述べる定式化の中では各条件分岐の条件演算が 1 つであることを前提とするので，もし，条件演算が複数ある場合は以下のようにグラフを変形する．条件演算が複数ある場合というのは，例えば図 4(a) のように，ある条件分岐の条件演算が他の条件分岐中にある場合である．この例では o_1, o_2, o_3 が b_2 の条件演算であるから，このときのフローグラフは同図 (b) のようになる．このグラフに同図 (c) のように遅延時間が 0 である擬似演算節点 o_p を加え，この演算を b_2 の新しい条件演算とする．この o_pは o_1, o_2, o_3 よりも前の制御ステップに割り付けられてはならないので，$o_1 \rightarrow o_p, o_2 \rightarrow o_p, o_3 \rightarrow o_p$ を設ける．条件演算が o_1, o_2, o_3 から o_pになったので，(ii) の先行関係を表す $o_1 \Rightarrow o_4, o_2 \Rightarrow o_4, o_3 \Rightarrow o_4$ を開放除去し，新たに $o_p \Rightarrow o_4$ を設ける．このようにして擬似演算節点を入れても，演算間の先行関係は保たれ，しかも o_p はどの演算器でも実行されないのでスケジューリングの時に必要となる演算器の個数にも影響を及ぼさない．

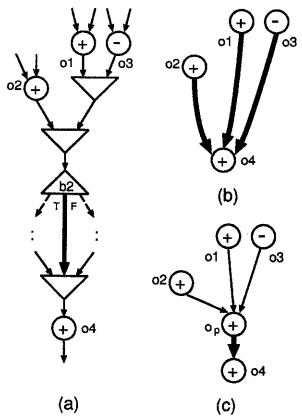

図 4: 複数の条件演算がある場合:(a) CDFG, (b) 変形前のフローグラフ, (c) 変形後のフローグラフ．

CDFG 中の分岐節点をグラフとして表現しないが，定式化の中で条件演算についての情報や，条件分岐がネストしている場合にどの条件分岐がどの条件分岐中にあるかについての情報が必要となるため，グラフとは別に条件分岐の集合 B として表すことにする．各条件分岐 $b_l \in B$ は，対応する条件演算の情報と，CDFG 上で付けられたタグ $tag(b_l)$ を持っている．

以後のスケジューリングの処理は CDFG から抽出したフローグラフ G_f と，条件分岐の集合 B を用いて行なう．

3.2 準備

本節では，3.3 節の中で用いる記号の定義を行なう．

フローグラフ G_f 中の演算数を n，与えられた制御ステップ数を r とする．演算の集合を $O = \{o_i | 1 \leq i \leq n\}$，制御ステップの集合を $S = \{s_j | 1 \leq j \leq r\}$ とする．演算 o_i を実行する演算器の種類を $type(o_i)$ で表し，演算器のすべての種類の集合を $T = \{t_k | 1 \leq k \leq m\}$ と表す．種類 t_k の演算器のコストを c_k とする．種類 t_k の演算器で実行される演算の集合を O_k で表す．$O_k = \{o_i | o_i \in O, type(o_i) = t_k\}$ である．フローグラフ G_f に対して，演算 o_i を ASAP (As Soon As Possible) 法，ALAP (As Late As Possible) 法で割り当てた制御ステップをそれぞれ E_i, L_i とする．o_i は必ず E_i と L_i の間に割り付けられることになる．この範囲を $range(o_i) = \{s_j | E_i \leq j \leq L_i\}$ で表し，o_i の実行可能範囲と呼ぶことにする．フローグラフ中の条件分岐の個数を q とし，全条件分岐の集合を $B = \{b_l | 1 \leq l \leq q\}$ で表す．条件分岐 b_l の条件演算を $o_{cnt(b_l)}$ で表す．演算 o_i や条件分岐 b_l が含まれる条件分岐のネストのレベルとは，タグ $tag(o_i), tag(b_l)$ の要素数であり，それぞれ $level(o_i), level(b_l)$ と書くことにする．制御ステップ s_j で必要となる種類 t_k の演算器の個数を $N_{k,j}$，全体として必要となる種類 t_k の演

算器の個数を表す整数変数を M_k とする．演算 o_i が制御ステップ s_j に割り付けられたかどうかを変数 $x_{i,j}$ で表す．$x_{i,j}$ は o_i が s_j に割り付けられた時 1，そうでなければ 0 となるブール変数である．なお本文では，論理和，論理積を $\vee$，$\wedge$ で表し，算術和，算術積を $+$，$\times$ で表すことにする．

3.3 ブール式による定式化

本節では条件分岐中の動的な排他性を考慮して，スケジューリング問題をブール式を用いて定式化する．

制約 1 ～ 3 は，次のように書くことができる．

制約条件 1: フローグラフ中の全演算が割り当てられる [7]

$$C1 = \bigwedge_{i=1}^{n} \left(\bigvee_{a=E_i}^{L_i} \left(\bigwedge_{j=E_i}^{L_i} \Delta(\delta, x_{i,j}) \right) \right) \tag{1}$$

ここで，

$$\Delta(1, x_{i,j}) = x_{i,j}, \Delta(0, x_{i,j}) = \overline{x_{i,j}}, \delta = \begin{cases} 1 & \text{if } a = j \\ 0 & \text{if } a \neq j \end{cases}$$

である．

制約条件 2: フローグラフ中の全演算が実行可能な順序に保たれる

$$\begin{aligned} C2 \;=\; & \left(\bigwedge_{o_m \Rightarrow o_n} \left(\bigwedge_{a=E_n}^{L_n} \left(\bigwedge_{b=a}^{L_m} (\overline{x_{n,a} \wedge x_{m,b}}) \right) \right) \right) \\ \wedge \; & \left(\bigwedge_{o_m \rightarrow o_n} \left(\bigwedge_{a=E_n}^{L_n} \left(\bigwedge_{b=a+1}^{L_m} (\overline{x_{n,a} \wedge x_{m,b}}) \right) \right) \right) \end{aligned} \tag{2}$$

制約条件 3: 各制御ステップで必要な演算器 t_k の数が演算器の総数 M_k を越えない

$$C3 = \bigwedge_{k=1}^{m} \left(\bigwedge_{j=1}^{r} (M_k - N_{k,j} \geq 0) \right) \tag{3}$$

条件分岐がある場合に考慮しなければならないのは，制約条件 3 の $N_{k,j}$ である．以後，$N_{k,j}$ について考えることにする．

条件分岐がない場合，

$$N_{k,j} \;=\; \sum_{i|(o_i \in O_k)} x_{i,j} \tag{4}$$

と表せる [7]．

条件分岐がある場合は，基本的には条件演算を先に行なう場合と後で行なう場合についてそれぞれ必要な演算器数を求め，条件演算が実行される制御ステップにしたがって，それらの一方を選択すればよい．

図 5のフローグラフにおいて，$+$ を加算器 (t_1)，$<$ を比較器 (t_2) で行なう場合を考える．条件演算 o_1 $(range(o_1) = \{s_1, s_2, s_3\})$ を制御ステップ s_1 で実行する (すなわち条件演算を先に実行する) 場合，演算 o_5 と o_6 を制御ステップ s_3 に割り当てると，どちらか一方の加算しか実行されないので 1 つの加算器を共有できる．しかし，o_1 を制御ステップ s_3 で実行する (すなわち条件演算を後に実行する) 場合，制御ステップ s_3 に o_5 と o_6 を割り当てると，両方の加算を実行しなければならず 2 つの加算器が必要になる．o_1 が制御ステップ s_3 より前に実行されたかどうかは $\bigvee_{1 \leq j \leq 2} x_{1,j}$ の値を調べることによってわかる．この値が 1 の時には実行が終了

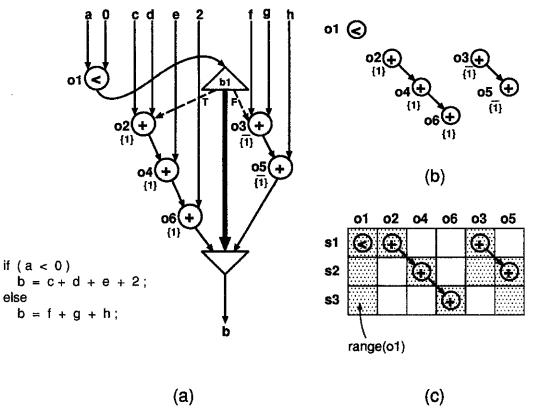

図 5: 条件分岐のあるフローグラフの例：(a) CDFG, (b) フローグラフ, (c) 各演算の実行可能範囲．

しており，0 の時にはまだ終了していないことになる．したがって，制御ステップ s_3 で必要となる加算器の個数 $N_{1,3}$ は

$$\begin{aligned} N_{1,3} \;=\; & \left(\bigvee_{1 \leq j \leq 2} x_{1,j} \right) \times \max(x_{5,3}, x_{6,3}) \\ + \; & \left(\overline{\bigvee_{1 \leq j \leq 2} x_{1,j}} \right) \times (x_{5,3} + x_{6,3}) \end{aligned} \tag{5}$$

と表現できる．

一般の場合は次のように定式化できる．まず条件分岐がネストしていない場合について考える．制御ステップ s_j で，条件分岐 b_l 中の演算に必要な種類 t_k の演算器について考えると，真のパス中の演算に必要な演算器の個数 $NT_{k,j,l}$，偽のパス中の演算に必要な演算器の個数 $NF_{k,j,l}$ はそれぞれ，タグ中に $l, \bar{l}$ がある演算の和であるから，

$$NT_{k,j,l} \;=\; \sum_{i|(o_i \in O_k) \wedge (l \in tag(o_i))} x_{i,j} \tag{6}$$

$$NF_{k,j,l} \;=\; \sum_{i|(o_i \in O_k) \wedge (\bar{l} \in tag(o_i))} x_{i,j} \tag{7}$$

と表せる．制御ステップ s_j で，条件分岐 b_l 中の演算に必要な種類 t_k の演算器の個数 $NB_{k,j,l}$ は，

$$NB_{k,j,l} = \begin{cases} \max(NT_{k,j,l}, NF_{k,j,l}) & o_{cnt(b_l)} \text{ が } s_j \text{ より前で実行される時} \\ NT_{k,j,l} + NF_{k,j,l} & o_{cnt(b_l)} \text{ が } s_j \text{ 以降に実行される時} \end{cases}$$

であるから，

$$\begin{aligned} NB_{k,j,l} \;=\; & \left(\bigvee_{E_{cnt(b_l)} \leq i \leq \min(j-1, L_{cnt(b_l)})} x_{cnt(b_l),i} \right) \\ & \times \; \max(NT_{k,j,l}, NF_{k,j,l}) \\ & + \; \left(\overline{\bigvee_{E_{cnt(b_l)} \leq i \leq \min(j-1, L_{cnt(b_l)})} x_{cnt(b_l),i}} \right) \\ & \times \; (NT_{k,j,l} + NF_{k,j,l}) \end{aligned} \tag{8}$$

となる．ここで，$E_{cnt(b_l)} \geq j$ の時は

$$\bigvee_{E_{cnt(b_l)} \leq i \leq \min(j-1, L_{cnt(b_l)})} x_{cnt(b_l),i} = 0$$

とする．

(8) 式では，条件分岐内の演算だけを考えたが，条件分岐外の演算も加えると，$N_{k,j}$は，

$$\begin{aligned} N_{k,j} &= (\text{条件分岐内の演算に必要な演算器数}) \\ &+ (\text{条件分岐外の演算に必要な演算器数}) \\ &= \sum_{l|(b_l \in B)} NB_{k,j,l} + \sum_{i|(o_i \in O_k) \wedge (level(o_i)=0)} x_{i,j} \end{aligned} \quad (9)$$

と表せる．

次に条件分岐がネストしている場合を考える．レベルが 0 である条件分岐 b_l については $NB_{k,j,l}$ は (8) 式と同じである．条件分岐 b_l の真のパス中に条件分岐 b_c $(level(b_c) = level(b_l)+1)$ があるとすると，$NT_{k,j,l}$ は，条件分岐 b_c 中の演算に必要な演算器数と，それ以外の演算に必要な演算器数の和となるので，

$$\begin{aligned} NT_{k,j,l} &= \sum_{c|(b_c \in B) \wedge (l \in tag(b_c)) \wedge (level(b_c)=level(b_l)+1)} NB_{k,j,c} \\ &+ \sum_{i|(o_i \in O_k) \wedge (l \in tag(o_i)) \wedge (level(o_i)=level(b_l)+1)} x_{i,j} \end{aligned} \quad (10)$$

と書くことができる．$NF_{k,j,l}$ も同様に，

$$\begin{aligned} NF_{k,j,l} &= \sum_{i|(b_c \in B) \wedge (\overline{l} \in tag(b_c)) \wedge (level(b_c)=level(b_l)+1)} NB_{k,j,c} \\ &+ \sum_{i|(o_i \in O_k) \wedge (\overline{l} \in tag(o_i)) \wedge (level(o_i)=level(b_l)+1)} x_{i,j} \end{aligned} \quad (11)$$

と書くことができる．$NB_{k,j,c}$ はレベル 0 の条件分岐の時と同様，

$$NB_{k,j,c} = \begin{cases} \max(NT_{k,j,c}, NF_{k,j,c}) & o_{cnt(b_c)} \text{ が } s_j \text{ より前で実行される時} \\ NT_{k,j,c} + NF_{k,j,c} & o_{cnt(b_c)} \text{ が } s_j \text{ 以降に実行される時} \end{cases}$$

であるから，(8) 式と同じ形で表される．したがって $NB_{k,j,l}$ は (8), (10), (11) 式を再帰的に適用することによって求めることができる．$N_{k,j}$ はレベル 0 の演算に必要な演算器数とレベル 0 の条件分岐内の演算に必要な演算器数の和であり，

$$\begin{aligned} N_{k,j} &= \sum_{l|(b_l \in B) \wedge (level(b_l)=0)} NB_{k,j,l} \\ &+ \sum_{i|(o_i \in O_k) \wedge (level(o_i)=0)} x_{i,j} \end{aligned} \quad (12)$$

と表せる．

(8), (10), (11), (12) 式が $N_{k,j}$の一般形であることを示す．(10) 式についてネストがない場合を考えると，レベルが 1 以上の条件分岐はないので第 1 項は 0 となる．第 2 項についてはタグに l を含む演算はすべてレベルが 1 であるため，結局 (6) 式と同じである．(11) 式についても同様である．また，条件分岐がない場合を考えると，(12) 式の第 1 項は 0 であり，演算のレベルはすべて 0 であるから (4) 式と一致する．したがって，制約条件 3 における $N_{k,j}$ は，条件分岐の有無，条件分岐のネストの有無に関係なく (8), (10), (11), (12) 式で表すことができる．

以上の制約の元で演算器のコストを最小にするために目的関数 Cを

$$C = \sum_{t_k \in T} c_k \times M_k \rightarrow min. \quad (13)$$

とする．制約条件を満たす変数の値の組のうち Cが最小となるものを選択すればそれが解となる．

制約条件を満たす変数の値の組を求めるためには，$C1 \wedge C2 \wedge C3$ を BDD を用いて表現し，1 の定数節点への経路を求めればよい．このとき (変数の個数) × (制約条件を満たす変数の値の組の総数) に比例した時間ですべての組を列挙することができる．

$C2$ には整数変数 M_k があるが，これは [7] と同様 0-1 の値をとる変数 $m_{k,a}$ を導入し，

$$M_k = \sum_{a=1}^{N_k} a \times m_{k,a} \quad (14)$$

で表す．ここで，N_k は各制御ステップにおける種類 t_k の演算器で実行される演算の実行可能範囲の重なりの最大値である．$m_{k,a}$ は $1 \leq a \leq N_k$ で 1 つだけ 1 となり，残りは 0 となるものとする．また式中の整数の算術演算は 2 進化符号で扱い，負の数は 2 の補数で表す．$C2$ には算術乗算があり，一般には BDD で効率良く表現できないが，ここでは 2 項の内一方が 0 か 1 しか値をとらないため，他方の整数の各ビットとの論理積をとることで表現できる．

全体のスケジューリングの手順は以下のようになる．

[手順]

1 ° CDFG 上で各演算のタグを求める．

2 ° CDFG からフローグラフ G_f を作成する．

3 ° ASAP, ALAP 法により，フローグラフ G_f 中の各演算の実行可能範囲を求める．

4 ° 制約条件 $C1, C2, C3$ を求め，$C1 \wedge C2 \wedge C3$ を満足し，コスト関数 Cを最小にする変数の値の組を求める．

4 実験結果

本スケジューリング手法を SPARC station 2 (主記憶 32 MB) 上に C 言語を用いて実現し，[3] で用いられた "maha" と "parker" の 2 つのデータを用いて，制御ステップ数とチェイン数を変化させてスケジューリングを行なった．BDD パッケージは [8] の SBDD を用いた．表 1 の OURS は本手法を示し，ALPS は [1], KIM は [2], CVLS は [3], MAHA は [9] の手法を示している．Add と Sub はスケジューリングに要した加算器と減算器の個数，Step は制御ステップ数を表している．ch はチェイン数を示している．# Sol's は本手法で列挙された解の個数，# Nodes は制約式を表すのに必要になった BDD の節点の個数，CPU time は計算時間を示している．KIM, CVLS は演算器の個数を制約として与えて最小となる制御ステップ数を求めている．ALPS と本手法は制御ステップ数を制約として与えて演算器の個数を最小化する．MAHA はどちらを制約とすることもできる．

制御ステップ数が同じ場合，チェイン数が少なく，演算器数が少ない方が良いスケジューリング結果と言える．データ maha の制御ステップ数が 3 のスケジューリング結果を見ると，CVLS, KIM, ALPS はチェイン数が 3, 加算器が 2，減算器が 3 必要になる．本手法はチェイン数が 2，加算器，減算器ともに 2 でスケジューリングできる．制御ステップ数が 4 の時は，チェイン数を 1 とすると CVLS が加算器が 2，減算器が 3 であるのに対し，本手法では加算器，減算器がともに 2 で実現できる．また MAHA はチェイン数を 3 としても加算器が 2，減算器が 3 必要である．

以上のように本手法は，条件演算のスケジューリングと，それに伴って変化する条件分岐中の演算間の演算器の共有

表 1: 実験結果

Data	Methods	Step	Add	Sub	Ch	# Sol's	# Nodes	CPU time
maha	**OURS**	**3**	**2**	**2**	**2**	2904	6800	6.9 sec
	KIM	3	2	3	3	–	–	–
	CVLS	3	2	3	3	–	–	–
	ALPS	3	3	3	3	–	–	–
	OURS	**4**	**2**	**1**	**2**	32064	88973	96.6 sec
	CVLS	4	2	1	2	–	–	–
	ALPS	4	2	2	2	–	–	–
	OURS	**4**	**2**	**2**	**1**	3840	2593	8.2 sec
	MAHA	4	2	3	3	–	–	–
	CVLS	4	2	3	1	–	–	–
	OURS	**5**	**1**	**1**	**1**	768	45129	6.0 sec
	CVLS	5	1	1	1	–	–	–
	KIM	6	1	1	2	–	–	–
	KIM	8	1	1	1	–	–	–
	ALPS	8	1	1	1	–	–	–
	MAHA	8	1	1	2	–	–	–
parker	**OURS**	**3**	**2**	**2**	**2**	4608	5218	10.1 sec
	OURS	**4**	**2**	**2**	**1**	11880	2780	22.6 sec
	CVLS	4	2	3	1	–	–	–
	OURS	**5**	**1**	**1**	**1**	1920	38414	8.5 sec
	CVLS	5	1	1	1	–	–	–

という2つの問題を同時に解くことにより, 従来よりも良いスケジューリング結果を得ている.

5 おわりに

本文では, 条件分岐を含む動作記述の最適な時間制約スケジューリングを求める手法を提案した. 本手法は, 動作記述が与えられた時, 制御ステップ数を制約として, 演算間の動的な排他性まで考慮して最小の演算器コストでスケジューリングを行なうことができる. 実験の結果, いくつかのベンチマークに対して従来手法よりも優れた結果を得ることができた.

しかし, 動作記述中の演算の個数が多くなると 制約条件を BDD で表現しきれないという問題点がある. これを解決するためには, 整数線形計画法に対して用いられているような変数の数を制限する手法 [7][10] や, 他のスケジューリング手法と組み合わせて部分問題を本手法で解くことが有効であると考えられる.

謝辞

SBDD に基づく論理関数処理プログラムを提供して頂いた NTT(株) の湊真一氏に感謝致します.

参考文献

[1] J-H. Lee, Y-C. Hsu, and Y-L Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis", *Proc. ICCAD '89*, pp. 20–23, 1989.

[2] T. Kim, J. W.S. Liu, and C. L. Liu, "A scheduling algorithm for conditional resource sharing", *Proc. ICCAD '91*, pp. 84–87 , 1991.

[3] K. Wakabayashi and H. Tanaka, "Global scheduling independent of control dependencies based on condition vectors", *Proc. 29th Design Automation Conf.*, pp. 112–115, 1992.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Trans. Comput.*, Vol. C–35, No. 8, pp. 677–691, August, 1986.

[5] M. Rim and R. Jain, "Representing conditional branches for high-level synthesis applications", *Proc. 29th Design Automation Conf.*, pp. 106–111, 1992.

[6] 若林, 田中, 藤田, "機能合成システム Cyber: 制御構造制約を越えたスケジューリング手法", 信学技法, VLD91–94, pp. 41–48, 1991.

[7] T. Miyazaki, "Boolean-based formulation for data path synthesis", *Proc. IEEE Asia Pacific Conf. on Circuits and Systems*, pp. 201–206, 1992.

[8] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation", *Proc. 27th Design Automation Conf.*, pp.52–57, 1990.

[9] A. C. Parker, J. "T". Pizarro, and M. Mlinar, "MAHA: A program for datapath synthesis", *Proc. 23rd Design Automation Conf.*, pp. 461–466, 1986.

[10] H. Komi, S. Yamada, and K. Fukunaga, "A scheduling method by stepwise expansion in high-level synthesis", *Proc. ICCAD '92*, pp. 234–237, 1992.