社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－SLDM－135（4）
2008／5／8

マルチプレクサの削減を目的としたバインディング改善手法

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 リソース共有型の動作合成では，バインディングの結果としてレジスタや演算器の入力側に必要に応じてマルチプレクサが挿入される．マルチプレクサは合成結果の回路の性能低下や面積増大の原因となるため，バインディングにおいて発生するマルチプレクサは少ないほうが望ましい．本論文では，マルチプレクサの総入力数削減を目的としたバインディング改善手法を提案する．提案手法は，適当な演算器バインディングおよびレジスタバインディングを初期解として，タブーサーチをベースとした局所改善によって演算器バインディングおよびレジスタバインディングの変更を反復して行なう．実験の結果，提案手法はマルチプレクサの総入力数を既存研究に対して平均で約30%削減することが可能であり，計算時間は数秒から数分であった．
キーワード 動作合成，バインディング，EDA

Improvement Technique of Binding for Multiplexer Reduction

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 Behavioral Synthesis for resource shared architecture, multiplexers are inserted between registers and functional units as a result of binding if necessary. Multiplexer optimization in binding is important for performance and area of synthesized circuit. In this paper, we propose an improvement technique of binding to reduce total amount of multiplexer ports. In our approach, starting point is initial register binding and initial functional unit binding. Both functional unit binging and register binding are modified by local improvements based on taboo search iteratively. Experimental results show that our approach is able to reduce total amount of multiplexer ports by 30% on an average compared to a traditional binding algorithm. Computation time of our approach is several seconds to a few minutes.
Key words Behavioral Synthesis , Binding , EDA

1. はじめに

近年LSIの高集積化に伴う設計コストの増大および設計期間の長期化が問題となっている．これらの問題に対する機能設計レベルでの解決策として動作合成がある．動作合成とは，現在の機能設計において主に用いられているRTL(Register Transfer Level)記述を，より高い抽象度を持った動作記述から自動生成する設計支援技術である．RTL記述がクロック毎の動作と回路の内部構造を詳細に定義するのに対して，動作記述では処理の順序が定義される．データパスやコントローラなど，回路の詳細な内部構造は，動作合成ツールに与える制約に応じて自動的に生成される．よって，動作合成を機能設計に組み込むことで，設計コストの削減，設計期間の短縮が期待できる．また，動作合成ツールに与える制約を変えることで異なった構造のデータパスを生成することが可能であり，仕様変更への柔軟な対応やIP(Intellectual Property)再利用性の向上などの利点がある．

一方，リソース共有型の動作合成(以下，動作合成)では，演算器やレジスタの入力ポートに必要に応じてマルチプレクサが挿入される．マルチプレクサは合成結果の回路の性能や面積，消費電力に対して大きな影響を与える．特にFPGA(Field Programmable Gate Array)の場合にはマルチプレクサが回路の性能や面積，消費電力に対して与える影響が大きいことが報告されている[8]．そのため，動作合成におけるマルチプレクサの削減が必要であると考えられる．

動作合成は，モジュールアローケーション，スケジューリング，バインディングから構成される．モジュールアロケーションでは使用する演算器の種類と数を決定する．スケジューリングでは各演算を実行するクロックサイクルを決定する．バインディングでは，スケジューリングされた各演算を演算器に割り当てる演算器バインディングと演算間を流れる変数をレジスタに割り当てるレジスタバインディングを行なう．これらの処理は生成されるマルチプレクサに関してそれぞれ影響をあたえる．本論文では，モジュールアロケーション，スケジューリングは終了しているものとし，バインディングに着目する．

マルチプレクサの最小化を目的としたバインディング手法については多くの既存研究が存在する．[4] [5] ではクリーク分割を用いたバインディングが提案されている．クリーク分割がNP困難な問題であることから近似アルゴリズムを用いており，レジスタ数の最小を保証していない．[6] では，演算器バインディングの過程で分枝限定法を用いたレジスタバインディングを行なっている．[2] では演算器バインディングおよびレジスタバインディングをそれぞれ重み付き二部グラフの最大マッチングを用いて行なう．二部グラフのエッジには，変数や演算を共有した場合にマルチプレクサが発生する確率が重みとして付加される．本手法はレジスタ数の最小を保証しており，マルチプレクサの総入力数において [4] [5] [6] よりも良い結果が得られている．[3] ではスケジューリングと演算器バインディング，レジスタバインディングをクロックサイクルごとに同時に行なう．クロックサイクルごとに，出来る限り多くの演算をスケジューリングし，スケジューリングされた演算およびこれらの演算によって生成される変数のバインディングを行なう．各バインディングは，演算，演算器，レジスタからなるネットワークの最小コスト最大フローを得ることによって行なわれる．実験ではマルチプレクサの総入力数について，[2] よりも良い結果が得られているがレジスタ数が増大している．[1] では演算器バインディングは終了しているものとし，レジスタバインディングのみを行なっている．レジスタバインディングは最小コスト最大フロー問題として解かれる．また，レジスタバインディング終了後にポート割り当ての変更を行なうことで更にマルチプレクサを削減している．以上の既存研究では，[3] を除いて演算器バインディングとレジスタバインディングを逐次的に行なっている．そのため，先に行なわれるバインディングにおいては，演算や変数の割り当てがマルチプレクサの発生に対して与える影響を把握することが困難である．[9] では，演算器バインディングとレジスタバインディングを同時に行なう手法が提案されている．各変数と演算に対して一つずつレジスタと演算器を割り当てた状態から開始して，演算器数とレジスタ数を指定した数になるまで徐々に減少させていくことでバインディングを行なう．各反復では，変数や演算を割り当てた場合のマルチプレクサのコストと前回の反復から得られる情報を利用する．実験の結果，既存手法と比較してマルチプレクサの面積と回路のクロック周期が10%程度削減されている．また，整数線形計画問題への定式化も行なっており，サイズの小さい動作記述について，最適解に対するマルチプレクサの面積の増加は約 5%となっている．

本論文では，適当な演算器バインディングおよびレジスタバインディングを初期解として，各バインディング結果を変更することによってマルチプレクサの削減を行なうバインディング改善手法を提案する．提案手法はタブーサーチをベースとした局所改善を反復的に行なう．実験の結果，既存手法と比較してマルチプレクサの総入力数を平均で約 30%，最大で約 40%削減することが可能であり，計算時間は数秒から数分であった．

本論文の構成は以下の通りである．第二章で用語の定義と問題の定義を行い，第三章で提案手法について述べる．第四章で実験結果を示し，第五章で本論文をまとめる．

2. 準　　備

動作合成において，動作記述は内部中間表現であるCDFG(Control Data Flow Graph) に変換される．CDFG は一つの CFG(Control Flow Graph) と一つ以上の DFG(Data Flow Graph) からなる．CFG は制御の流れを表す有向グラフである．DFG はデータの流れを表す非循環有向グラフであり，各ノードが演算を表し，エッジが演算間を流れる変数を表す．CFG の各ノードは一つの DFG に対応する．バインディングにおける入力はスケジューリング済みの全 DFG である．バインディング対象となる演算の集合を O，変数の集合を V で表し，レジスタの集合を R，演算器の集合を F で表す．二つの演算 $o_i, o_j \in O$ は実行されるクロックサイクル (以下，ステップ) が異なる場合に同一の演算器を共有することが出来る．二つの変数 $v_i, v_j \in V$ は生成されるステップと消費されるステップの間の区間 (以下，ライフタイム) が重複しない場合に同一のレジスタを共有することが出来る．二つの演算 $o_i, o_j \in O$ が同一の演算器を共有可能である時，二項関係 $\approx$ を用いて $o_i \approx o_j$ と表す．二つの変数 $v_i, v_j \in V$ が同一のレジスタを共有可能である時，二項関係 $\approx$ を用いて $v_i \approx v_j$ と表す．よって，演算器バインディング $g_o : O \longrightarrow F$ が満たすべき条件は，演算器 f に割り当てられている演算の集合を $O_b(f)$ として以下の式で表される．

$$\forall f_i \in F, \forall o_j, o_k \in O_b(f_i), o_j \approx o_k \tag{1}$$

また，レジスタバインディング $g_v : V \longrightarrow R$ が満たすべき条件は，レジスタ r に割り当てられている変数を $V_b(r)$ として以下の式で表される．

$$\forall r_i \in R, \forall v_j, v_k \in V_b(r_i), v_j \approx v_k \tag{2}$$

演算器バインディング g_o およびレジスタバインディング g_v におけるマルチプレクサの総入力数を $mux(g_o, g_v)$ で表す．本論文では，レジスタ数 $|R|$ および，演算器数 $|F|$ は固定であるものとし，レジスタ数は最小とする．また，異なる DFG に属する演算および変数は共有可能であるものとする．よって，本論文において対象とするバインディングは以下のように定義される．

入力：スケジューリング済みの演算の集合 O および変数の集合 V，演算器の集合 F とレジスタの集合 R

目的：式 (1)，式 (2) を満たし，マルチプレクサの総入力数

が小さい演算器バインディング g_o およびレジスタバインディング g_v を求める

3. 提案手法

提案手法は，適当な演算器バインディングおよびレジスタバインディングを初期解としてタブーサーチをベースとした局所改善を反復的に指定された回数行う．反復ごとに，最もマルチプレクサの総入力数が少ない演算器バインディング及びレジスタバインディングを最良解として保持し，全反復終了後に最良解を出力する．図 1 に提案手法の動作の概要を示す．

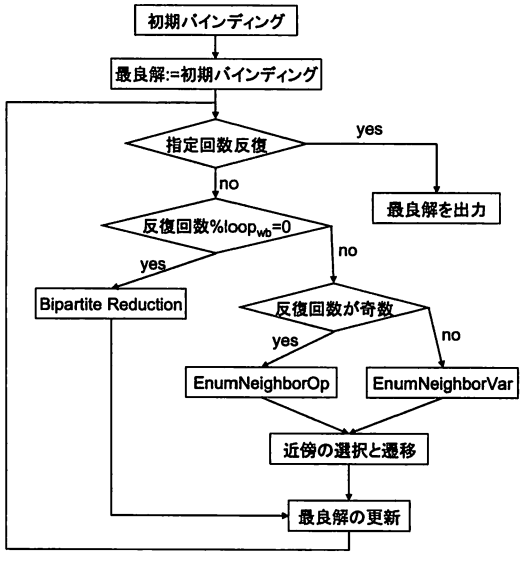

図 1 提案手法の動作

提案手法においては，演算器バインディングの変更とレジスタバインディングの変更を交互に行なう．また，一定の回数 $loop_{wb}$ に一回，二部グラフマッチングを用いた演算器バインディングおよびレジスタバインディングを行なう．

3.1 EnumNeighborOp

現在の演算器バインディングを変更した新たな演算器バインディングを近傍として複数生成する．近傍操作は，各演算器に割り当てられている演算の移動および交換である．各近傍操作において，演算は二個以上同時に移動，交換される場合がある．

演算器バンディングの変更によるマルチプレクサの総入力数削減を行なう場合，演算器とレジスタ間の接続を考慮した近傍操作を行なう必要がある．図 2 を例に必要となる近傍操作について述べる．図 2 において $p1, p2$ は各演算の入力ポートを表し，$Reg1$〜$Reg5$ はレジスタを表す．o_1〜o_3 は演算を表し，f_1 は演算器を表す．$p1'p2'$ は演算器 f_1 の各入力ポートを表す．この例では，レジスタ $Reg5$ の入力ポートと，演算器 f_1 の入力ポート p_2' に二入力のマルチプレクサが挿入される．まず，入力ポート $p2'$ に挿入されるマルチプレクサの入力数削減を行なう場合について考察を行なう．演算 o_3 を他の演算器に移動可能な場合，レジスタ $Reg3$ とポート $p2'$ 間の接続が不必要となるためポート $p2'$ に挿入されるマルチプレクサを削除することができる．演算 o_3 を他の演算器に移動できない場合，演算 o_1, o_2 の移動について考慮する必要があるが，演算 o_1, o_2 のポート $p1$ に消費される変数はレジスタ $Reg1$ とポート $p1'$ 間の接続を共有している．また，演算 o_1, o_2 のポート $p2$ に消費される変数は，レジスタ $Reg2$ とポート $p2'$ 間の接続を共有している．よって，演算 o_1 あるいは o_2 をそれぞれ単独で移動した場合，ポート $p2'$ に挿入されるマルチプレクサを削除することは出来ない．一方，演算 o_1, o_2 を同時に移動した場合，レジスタ $Reg2$ とポート $p2'$ 間の接続が不必要となるため，ポート $p2'$ に挿入されるマルチプレクサを削除することができる．次に，レジスタ $Reg5$ の入力ポートに挿入されるマルチプレクサの入力数削減を行なう場合には，演算器 f_1 とレジスタ $Reg5$ の接続を削除する必要があるが，この接続は演算 o_2, o_3 によって生成される変数によって共有されている．よって，レジスタ $Reg5$ の入力ポートに挿入されるマルチプレクサを削除するには演算 o_2, o_3 を同時に移動する必要がある．

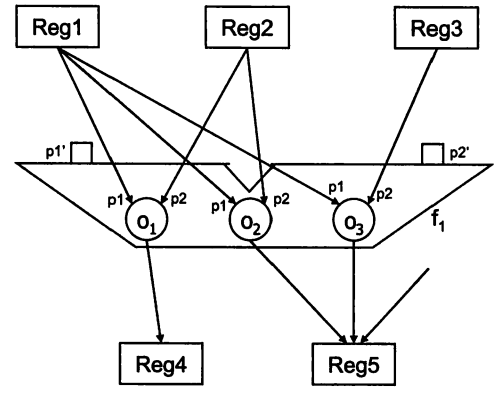

図 2 演算器バインディングの変更によるマルチプレクサ削減

上述のように，演算の移動および交換によって各マルチプレクサの入力数を削減するには，レジスタと演算器間の接続を共有している変数を求め，それらの変数を消費あるいは生成している演算を同時に移動，交換する必要がある．よって，各演算器において移動と交換の対象とするのは式 (3)，式 (4) のいずれかを満たす演算の集合である．演算器ごとに式 (3)，式 (4) を満たす演算の集合 O_{neib} を全列挙し，演算の集合を要素とするリスト $list_f(f_i)$ に保持する．ただし，$list_f(f_i)$ は要素の重複を許さない．$numin(f_i)$ は演算器 f_i の入力ポート数を表し，$v_{cons}(o)$ は演算 o によって消費される変数を表す．$v_{gen}(o)$ は演算 o によって生成される変数を表す．$r_b(v)$ は変数 v が割り当てられているレジスタを表す．

$$\left| \bigcup_{o_j \in O_{neib}} r_b(v_{cons}(o_j)) \right| = numin(f_i) \quad (3)$$

$$\forall o_j, o_k \in O_{neib}, r_b(v_{gen}(o_j)) = r_b(v_{gen}(o_k)) \quad (4)$$

図 2 の場合，演算の集合のリスト $list_f(f_1)$ は以下のように

表される．

$$list_f(f_1) = \{\{o_1, o_2\}, \{o_3\}, \{o_2, o_3\}, \{o_1\}\}$$

近傍操作を演算の集合 O_{neib} の移動と交換とすることで，近傍生成において列挙される近傍の数を削減することができる．O_{neib} を移動する場合に移動先となり得る演算器の集合 $F_c(O_{neib})$ は，各演算 $o_m \in O_{neib}$ を割り当て可能な演算器の集合を $F_t(o_m)$ として以下の式で表される．

$$F_c(O_{neib}) = \bigcap_{o_m \in O_{neib}} F_t(o_m) \tag{5}$$

O_{neib} に含まれる演算を一つずつ移動した場合，各演算 $o_n \in O_{neib}$ の移動先となり得る演算器数の合計値 N は以下の式で表される．

$$N = \sum_{o_n \in O_{neib}} |F_t(o_n)| \tag{6}$$

明らかに，$|F_c(O_{neib})| < N$ であるので，生成される近傍の数を削減することが出来る．

各演算器 f_i について演算の集合のリスト $list_f(f_i)$ を得た後，$list_f(f_i)$ を各演算の集合の要素数をキーとして昇順ソートする．次に $list_f(f_i)$ の先頭から $k = round(ratio \times list_f(f_i).size())$ 番目よりあとの要素を削除する．ただし，$k \geqq 1$ となるように調整を行なう．$ratio$ は 1.0 以下かつ min_{ratio} 以上の正数であり，各反復において最了解が更新されれば $delta_{ratio}$ 減少し，一定の回数 $loop_{ratio}$ に渡って最良解が更新されなければ $delta_{ratio}$ 増加する．$ratio$ の初期値は 1.0 である．近傍生成の対象となる演算の集合のリスト $list_f(f_i)$ を生成した後，各演算器について演算の集合を割り当て可能な演算器に移動し，演算器の対ごとに演算の集合を交換する．各近傍操作を適応する前のバインディングを g_o, g_v，適応した後のバインディングを g'_o, g'_v として各近傍のゲイン $gain(g'_o, g'_v)$ を以下の式で定義する．

$$gain(g'_o, g'_v) = mux(g_o, g_v) - mux(g'_o, g'_v) \tag{7}$$

各近傍操作はそれぞれのゲインと共にリストとして保持される．

3.2 EnumNeighborVar

現在のレジスタバインディングを変更した新たなレジスタバインディングを近傍として複数生成する．近傍操作は，変数の移動と交換である．まず，EnumNeighborOp と同様の概念に基づいて，移動と交換の対象となる変数の集合のリストをレジスタごとに生成する．各レジスタ $r_i \in R$ に割り当てられている変数の集合 $V_b(r_i)$ から式 (8)，式 (9) のうちいずれかを満たす変数の集合 V_{neib} を全て列挙し，変数の集合を要素とするリスト $list_r(r_i)$ に保持する．ただし，$list_r(r_i)$ は要素の重複を許さない．$p_{source}(v)$ は変数 v を生成する演算器の出力ポートを表し，$P_{sink}(v)$ は変数 v を消費する演算器の入力ポートの集合を表す．

$$\forall v_j, v_k \in V_{neib}, p_{source}(v_j) = p_{source}(v_k) \tag{8}$$

$$\forall v_j, v_k \in V_{neib}, P_{sink}(v_j) \cap P_{sink}(v_k) \neq \phi \tag{9}$$

式 (8) は，V_{neib} に含まれる変数が，同一の演算器の同一のポートから出力されていることを示す．式 (9) は，V_{neib} に含まれる変数が，同一の演算器の同一のポートによって消費されることを示す．即ち，V_{neib} は同一の接続を共有する変数の集合である．以後の処理については EnumNeighborOp において行なわれるものと同様である．

3.3 Bipartite Reduction

探索が停滞することを防ぐために，一定の反復回数ごとに，重み付き二部グラフの最大マッチングを利用したレジスタバインディングおよび演算器バインディングを指定した回数行なう．なお，Bipartite Reduction の入力となる演算器バインディングおよびレジスタバインディングは，以下のように決定する．

- 前回の Bipartite Reduction の後に最良解の更新が起こった場合には，現在の最良解
- 前回の Bipartite Reduction の後に最良解の更新が起こらなかった場合には，現在の演算器バインディングおよびレジスタバインディング

Biparite Reduction の処理の概要を以下に示す．

Algorithm 1 Bipatite Reduction($g_o, g_v, O, V, F, R, num$)

```
1: best functional unit binding: best_o
2: best register binding: best_v
3: for loop = 1 to num do
4:    reset register binding
5:    g_v := RegBindingBipartite(R, V)
6:    if loop = 1 then
7:       best_v = g_v
8:    else
9:       if mux(g_o, g_v) < mux(best_o, best_v) then
10:         best_v := g_v
11:      end if
12:   end if
13:   reset functional unit binding
14:   g_o := FUBindingBipartite(O, F)
15:   if mux(g_o, g_v) < mux(best_o, best_v) then
16:      best_o := g_o
17:   end if
18: end for
19: return best_o, best_v
```

まず，全変数のバインディングをリセットし，Register Binding Bipartite でレジスタバインディングを行なう．次に，全演算のバインディングをリセットし，FU Binding Bipartite で演算器バインディングを行なう．Register Binding Bipartite と FU Binding Bipartite が終了するたびにマルチプレクサの総入力数が最小となる演算器バインディング $best_o$ およびレジスタバインディング $best_v$ の更新を行い，全反復終了後に $best_o, best_v$ を出力する．

Register Binding Bipartite と FU Binding Bipartite は [2]

と同様の手法でバインディングを行なう．Register Binding Bipartite では，全変数を互いに共有できない変数の集合 (以下，クラスタ) に分割し，クラスタごとにレジスタに割り当てる．まず，クラスタ内の変数の集合 $V_{bipartite}$ とレジスタの集合 R からなる二部グラフ $G = (V_{bipartite} \cup R, E)$ を生成する．エッジ $e_{ij} \in E$ は，変数 v_i をレジスタ r_j に割り当て可能な場合に生成される．各エッジ $e_{ij} \in E$ には，以下のように定義される重み w_{ij} が付加される．

- 変数 v_i をレジスタ r_j に割り当てた時に増大するマルチプレクサの総入力数

次に重み付き二部グラフ G の最大マッチングを Hungarian Method [7] によって求める．Hungariam Method の計算量は各クラスタについて $O(|R|^3)$ である．マッチングに含まれる各エッジ e_{mn} について，変数 v_m をレジスタ r_n に割り当てる．FU Binding Bipartite の処理は Register Binding Bipartite と同様である．また，タブーリストは Bipartite Reduction の終了後はリセットされる．

3.4 近傍の選択

EnumNeighborOp あるいは EnumNeighborVar によって近傍を生成した場合，適応する近傍 (以下，受理近傍) を選択する必要がある．近傍選択の概要を以下に示す．まず，近傍生成において生成された近傍のリスト *neighborlist* をゲインをキーとして降順にソートする．次に，ゲインが等しい近傍同士を，以下のように定義される値をキーとして昇順にソートする．

- 近傍において移動，交換される変数あるいは演算が過去の反復において受理近傍に含まれていた回数の平均値

neighborlist をソートした上で，*neighborlist* の先頭から順に受理近傍の選択を行なう．*neighborlist* に含まれる近傍 $neighbor_i$ が受理近傍として選択されるための条件は，$neighbor_i$ がタブーリスト内のタブーによって禁止されていないか，禁止されていても現在の最良解を更新することである．選択された受理近傍は現在のバインディングに対して適応される．また，受理近傍に対応したタブーがタブーリストに追加される．

タブーリストは一定の長さを持ったキューであり，探索の循環を防ぐために使用される．タブーリストには，禁止する近傍操作が各反復ごとにタブーとして記録される．提案手法では，変数あるいは演算が移動，交換された後に，移動元に戻ることを禁止する．例として，受理近傍が変数 v_1, v_2 をレジスタ r_2 からレジスタ r_5 に移動するという操作だった場合，タブーとして登録されるのは，変数 v_1 あるいは変数 v_2 をレジスタ r_2 へ割り当てるという操作を含む近傍である．また，提案手法では演算器バインディングの変更とレジスタバインディングの変更を交互に行なっているため，タブーリストは演算器バインディングに関するタブーリスト $taboolist_f$ とレジスタバインディングに関するタブーリスト $taboolist_r$ をそれぞれ用いる．タブーリストのサイズは $taboolist_f, taboolist_r$ 共に同一とする．

4. 実　　験

4.1 既存研究との比較

マルチプレクサの総入力数について，提案手法と LYRA [2] との比較を行なった．各手法を C++言語によって実装し，Xeon 5140(2.33GHz Dual Core)，メインメモリ 8GB の計算機を用いて実験を行なった．実験環境は以後の実験においても同一である．ベンチマークは表 1 に示すものを使用した．

表 1　ベンチマーク

ベンチマーク	変数の数	演算の数	レジスタ数
jped_dct	118	98	21
diff_eq	24	11	6
blowfish_encrypt	325	243	19
jacobi	304	270	73
sobel_filter	56	45	10

演算器数の制約は，ASAP(As Soon As Possible) スケジューリングから得られる各演算器数を 0.7 倍し，四捨五入を行なったものを用いる．スケジューリングはリストスケジューリングによって行なった．実験において用いた提案手法のパラメータおよび実験結果を表 2，表 3 に示す．

表 2　提案手法のパラメータ

初期解	LYRA
反復回数	5000
Bipartite Reduction を行なう間隔	1000
タブーリストのサイズ	10
min_{ratio}	0.3
$delta_{ratio}$	0.05
$loop_{ratio}$	100

表 3　マルチプレクサの総入力数と計算時間

	mux_input		run time[s]	
	LYRA	Proposed	LYRA	Proposed
jpegdct	132	89	0.1	12.6
blowfish_encrypt	112	88	0.5	13.0
sobel_filter	61	47	0.0	2.4
jacobi	421	259	4.1	134.1
diff_eq	24	18	0.0	0.7
Ave.	1.00	0.72	–	–

表 3 より，提案手法は LYRA と比較して，マルチプレクサの総入力数を平均で 28%，最大で 38%削減している．また，計算時間に関しては，ほとんどの回路において数秒から十数秒であり，比較的大きな回路においても数分である．

4.2 収束性の評価

提案手法はタブーサーチをベースとしており，演算器バインディング，レジスタバインディングについて最適解が得られることを保証していない．そのため，提案手法の収束性について評価を行なった．jpeg_dct において，演算器バインディングおよびレジスタバインディングをランダムに行なったものを初期解とし，提案手法を適応した．また，Bipartite Reduction が収束性に与える影響を評価するために，同一の初期解に対して Bipartite Reduction 無しの提案手法を適応した．試行回数は

500 回とし，演算器数の制約および提案手法のパラメータは既存研究との比較において用いたものと同一である．

全試行におけるマルチプレクサの総入力数の平均値および最小値，最大値を表 4 に示す．

表 4　収束性の評価

初期解			Proposed		
Ave.	Min.	Max.	Ave.	Min.	Max.
255.7	235	276	89.8	88	95

表 4 より，マルチプレクサの総入力数の平均値と最小値との差は 2.0%である．また，マルチプレクサの総入力数の最大値と最小値との差は 8.0%である．よって，提案手法は初期解に対する依存性が低く，収束性に優れていると考えられる．

次に，提案手法において，Bipartite Reduction を行なわない場合と行なう場合のマルチプレクサの総入力数の差を各試行について求めたものを図 3 に示す．また，Bipartite Reduction を行なわなかった場合のマルチプレクサの総入力数の平均値および最小値，最大値を表 5 に示す．

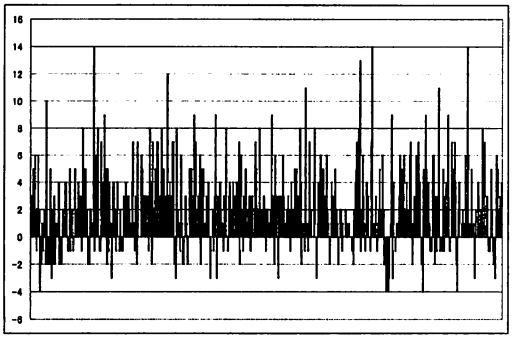

図 3　Bipartite Reduction の影響の評価

表 5　Bipartite Reduction を行なわない場合のマルチプレクサの総入力数

mux_input		
Ave.	Min.	Max
91.9	88	106

図 3，表 5 より，同一の初期解に対して提案手法を適応する場合，Bipartite Reduction を行なうことによって，マルチプレクサの総入力数をより削減できる可能性が高いと考えられる．また，収束性の観点からも Bipartite Reduction を行なう方が優れていると考えられる．

5. ま と め

本論文では，マルチプレクサの削減を目的としたバインディング改善手法を提案した．提案手法は，適当な演算器バインディングおよびレジスタバインディングを初期解として，タブーサーチをベースとした局所改善を反復して行なう．近傍生成において，各インターコネクトを流れる変数に着目することで，計算時間の削減および探索の効率化を図っている．また，一定の反復回数ごとに重み付き二部グラフの最大マッチングを用いた演算器バインディングおよびレジスタバインディングを行なうことによって探索の停滞を防止している．実験の結果，提案手法は既存研究と比較してマルチプレクサの総入力数を平均で約 30%削減しており，計算時間は数秒から数分であった．また，収束性についての評価より，提案手法は初期解に対する依存性が低く，収束性に優れている．今後の課題として，回路の遅延を考慮した手法および計算時間の短縮が挙げられる．

謝　辞

本研究の一部は，平成 19 年度科学研究費基盤研究 A，19200004，「価値と信用を搭載するディペンダブルな LSI の設計手法の研究」および平成 19 年度産学連携研究費受託研究，「統合的高信頼化設計のためのモデル化と検出・訂正・回復技術」によるものである．

文　献

[1] Deming Chen , Jason Cong. “Register Binding and Port Assignment for Multiplexer Optimization”, Proceedings of the 2004 ASP-DAC, pp.68-73, January 27-30, 2004

[2] Chu-Yi Huang, Yen-Shen Chen, Youn-Long Lin, Yu-Chin Hsu. “Data Path Allocation Based on Bipartite Weighted Matching”, Proceedings of the 27th Design Automation Conference, pp.499-504, 1990

[3] Teewhan Kim, C.L. Liu. “An Integrated Data Path Synthesis Algorithm Based on Network Flow Method”, Proceedings of the IEEE Custom Integrated Circuits Conference, 1995

[4] C-J. Tseng and D. P. Siewiorek, “Automated Synthesis of Data Path in Digital Systems”, IEEE Transaction on CAD of ICAS, Vol.CADJ, No.3, pp.379-395, Jul. 1986.

[5] P. G. Paulin, J. P. Knight, and E. F. Girczyc, “HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis”, 23rd IEEE Design Automation Conference, pp. 263-270, Jul. 1986.

[6] B. M. Pangrle, “Splicer: A Heuristic Approach to Connectivity Binding”, 25th ACM/IEEE Design Automation Conference, pp. 536-541, Jun. 1988.

[7] C. H. Papadimitriou and K. Steiglitz, “Combinatorial Optimization”, Prentice-Hall, 1982.

[8] Deming Chen, Jason Cong, and Yiping Fan, “Low-Power High-Level Synthesis for FPGA Architectures”, Proceedings of the International Symposeum on Low Power Electronics and Design, Aug. 2003.

[9] Jason Cong, Junjuan Xu, “Simultaneous FU and Register Binding Based on Network Flow Method”, Proceedings of the Design Automation and Test in Europe, 2008