Vol.2016-ARC-218 No.19
Vol.2016-SLDM174 No.19
2016/1/20

簡易合成法と並列処理を用いた論理合成の高速化手法

蘆苅将大　豊永昌彦　村岡道明
高知大学　大学院　理学専攻　情報科学分野
〒780-8520 高知県高知市曙町 2-5-1
E-mail: { ashikari, toyonaga, muraoka } @ is.kochi-u.ac.jp

概要　本報告では，簡易的な論理合成方法にマルチコアプロセッサによる並列化手法を適用した高速論理合成方法を提案する．本手法は対象とする論理回路を小規模回路に分割し，これらの部分回路を簡易な合成方法により合成することで，より高速な論理合成を可能とする．さらに，これらの部分回路をマルチコアプロセッサを用い，並列処理を行なうことにより高速な論理合成手法の確立を目指す．本手法に基づきプロトタイプを作成して，簡易論理合成手法の性能見積りを行なったところ，逐次処理と比べて，4 並列で最大 3.9 倍，8 並列で最大 7.7 倍の高速化を行うことが出来た．

キーワード：論理合成，マルチコアプロセッサ，設計自動化，並列処理，論理回路

An Acceleration Methodol of Logic Synthesis using Compact Synthesis and Parallel Processing

MASAHIRO ASHIKARI　MASAHIKO TOYONAGA
MICHIAKI MURAOKA
Graduate School of Science, Kochi University
2-5-1 Akebono－Cho, Kochi, 780-8520 Japan

Abstract: In this study, we propose a logic synthesis method adapt the multi-core processor to a simple logic synthesis method. This method is an algorithm that specializes in small part circuit. The results were adapted to this method a number of sub-circuits, it was possible to reduce the number of gates equal to the commercial tool. In addition, large-scale circuit is divided into a plurality of smaller sub-circuits, to perform parallel processing them by a multi-core processor. Parallel to adapt a simplified logic synthesis technique, where it was subjected to performance estimate, compared with the sequential processing, up to 3.9 times in four parallel, and could be performed for up to 7.7 times faster with eight parallel.

Keywords:Logic Synthesis, Multi-core Processor, EDA, Parallel Processing, logic circuit

1　はじめに

現在，さまざまな製品に LSI が用いられている．近年，LSI の微細化やシステムの大規模化が進んでおり，それに伴い回路設計やタイミング設計の長期化が問題となっている．タイミング設計の長期化の 1 つの要因として論理合成が挙げられる．大規模回路での論理合成には数日かかる場合もある．また，クリティカルパス部分などに問題があり，回路変更を行う際に再び全体として論理合成しなくてはならないという問題もある．そのため，回路仕様の変更があると再合成に膨大な時間を必要とする．そこで，回路仕様の変更がある場合，手軽に小規模な部分回路を取り出し，簡易的にかつ高速に合成する手法を提案する．また，大規模回路を小規模回路に分割し，この方法をマルチコアに適応させ，並列化することで高速に大規模回路の論理合成も可能とする．

2　研究目的・目標

今回の研究では，小規模回路のクリティカルパス部分に対して高速な論理合成を行うことで部分回路の合成時間の短縮を目標とする．合成手法として，中小規模回路全体を部分回路に分割し，それぞれについて簡易論理合成を行う．さらに，マルチコアに適応させることで，コア数に比例する処理時間を目標とする．

3　簡易合成手法

3.1 基本アルゴリズム

今回提案する簡易合成手法は真理値表を用いる方法である．基本アルゴリズムのフローチャートを図 1 に示す．

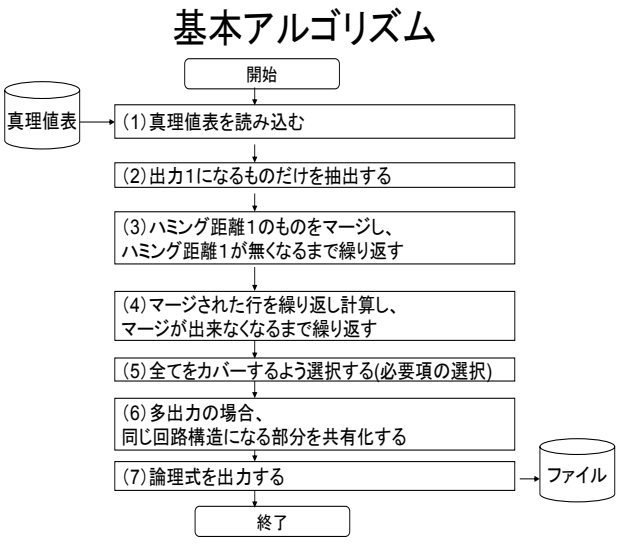

図 1　基本アルゴリズムの処理フロー

Vol.2016-ARC-218 No.19
Vol.2016-SLDM174 No.19
2016/1/20

基本アルゴリズムは 7 つのステップに分かれている．それぞれのステップについて詳しく説明する．

1) 真理値表の読み込み

回路仕様の変更やクリティカルパス部分など問題となる小規模回路を取り出し，その部分の入出力から真理値表を作成する．

2) 出力が 1 の行を抽出

読み込んだ真理値表から出力が 1 になるものだけを取り出し，新たに真理値表を作成する．

3) マージによる論理の簡単化

出力が 1 になる行を抽出した真理値表からハミング距離を総当たりに計算していく．ハミング距離とは，それぞれの入力の値の差の絶対値の合計である．ハミング距離が 1 のなるもの同士の行は 1 つの行にマージすることができる．マージできる部分は値が異なる部分であり，ドントケア(＊)で表記する．

4) 生成された行同士でのマージ

3）の方法によって生成された行同士でさらにマージを行う．この方法をマージができなくなるまで繰り返す．

5) 必要論理の選択

初めの出力が 1 になる行がすべてカバーできるように選択を行っていく．その際，ドントケアが最も多い行の方が多くの行をカバーし，ゲート数も少ない．そのため，最もマージされている行から優先し選択していく．

6) 多出力への対応

多出力への合成方法として，同じ論理部分を共有化する方法を提案する．多出力では同じ論理を持つ部分が重複する場合がある．そのため，論理の共有化を行うことが必要となる．詳細は 3.2 で説明する．

7) 論理式の出力

選択された行から論理式を出力する．選択された行から，入力の値が 0 であれば~入力名，1 であれば入力名という形式で論理式を生成する．

以上が基本アルゴリズムの詳細説明である．

3.2 多出力への対応（共有化）

多出力である場合には，同じ論理式をいくつも出力してしまう場合がある．その場合，重複した回路が複数個所に生成され，冗長性を持つ回路となる．そのため，同じ論理式になる部分は共有化を行うことで重複することなく，必要最低限のゲートでの合成が可能となる．この共有化を行うことで，多出力となる実用的な回路にも対応することができる．

基本アルゴリズムのステップ 6 である共有化部分について以下に説明する．

共有化の処理は 4 つのステップに分けることが出来る．(図 2)

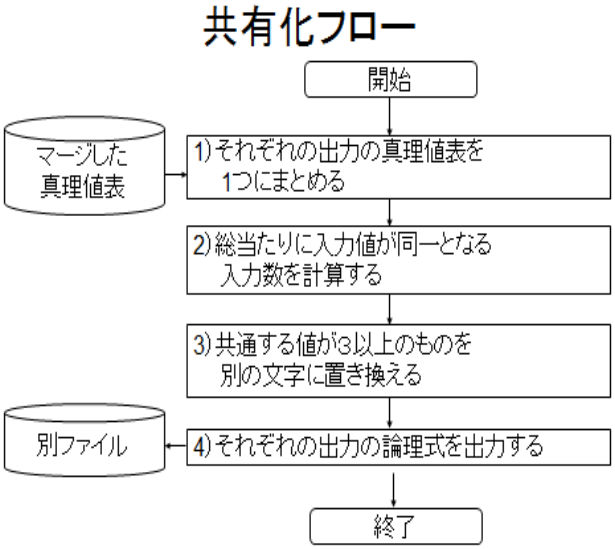

図 2 多出力のフロー図

それぞれのステップについて詳しく説明する．

1) それぞれの出力の真理値表を 1 つにまとめる

多出力の場合には，共有化を行う前にそれぞれの出力の真理値表を 1 つにまとめる．ここでは 4 入力 2 出力の場合を例にして説明する．入力は a,b,c,d で出力が y1,y2 である．(図 3)

図 3 真理値表のまとめ

2) 入力値が同一となる入力数を総当りに計算する

1 つにまとめた真理値表から，それぞれの入力の値が同じである入力数を計算する．例を図 4 に示す．図 4 での例では①と②の行は入力 a が 0，入力 b が 0 で同じ値である．しかし，入力 c は＊と 1，入力 d は 0 と＊で異なる．従って①と②の行の共通する値の数は 2 となる．この方法で①から⑦までの全ての行に対して総当りに行っていく．

Vol.2016-ARC-218 No.19
Vol.2016-SLDM174 No.19
2016/1/20

図 4　共通する値の計算方法

3)　同一になる入力値が 3 以上の行同士を共有化部分として置き換える

2)のステップで計算した同一になる入力値が 3 以上になる入力は共有化が可能であり，共有化ゲートに置き換える．今回は 2 入力のゲートで評価を行うため，入力値が同一となる入力数が 3 以上のものを共有化する．例として図 5 に示す．図 5 の例では共通する部分が 3 以上となるのが②，⑥の行と③，⑦の行の 2 つである．②，⑥の行では~a, ~b, c の部分が同じであるため A という共有化ゲートに置き換える．③，⑦の行は b ,~d を B という共有化ゲートに置き換える．この部分が重複していた部分であり，ゲート数を削減できる部分である．

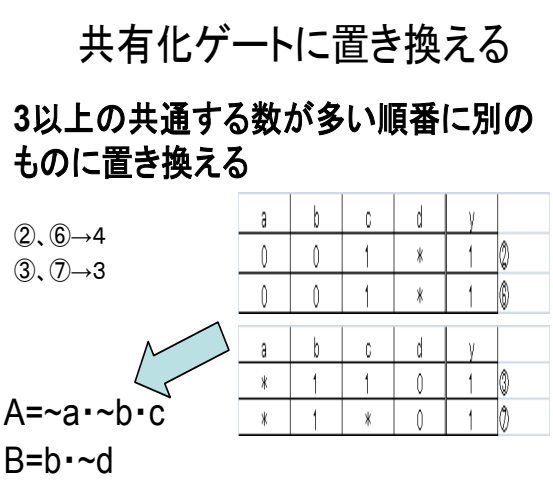

図 5　共有化ゲートへの置き換え

4)　それぞれの出力の論理式を生成する

それぞれの出力での論理式を生成する．論理式の生成方法は基本アルゴリズムと同じであり，共有化できる部分は共有化ゲートとして出力する．図 5 の例では A, B が共有化結果の論理式となる．

以上が，共有化のアルゴリズムである．

3.3 ネットリストインタフェース

部分回路の最適化や回路合成を行なう場合には，論理回路図とのインタフェースが必要となる．今回提案する方法として 2 つの方法がある．1 つはクリティカルパス部分を取り出し，合成する方法．2 つ目は回路全体を部分回路に分割し，合成する方法である．

1 つ目の方法では，クリティカルパスなどの問題となる部分の出力から関係する入力をコーンとして抽出する．(図 6)例では出力 O1 に関わる入力 I1, I2, I3 までの回路を抽出する．その抽出した部分回路を対象として本論理合成を行う．

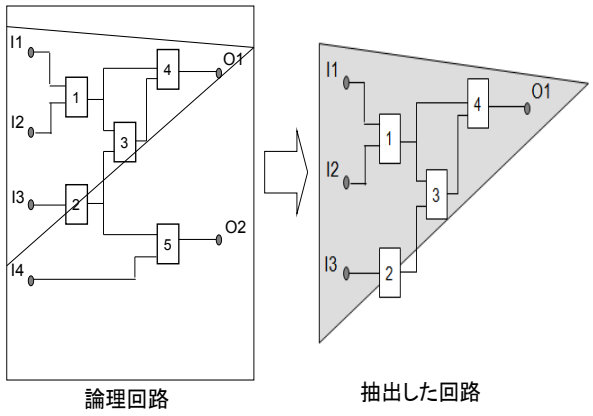

図 6　部分回路の抽出

2 つ目の方法では回路全体を制限した入出力で分割し，マクロを生成する方法である．全体を複数のマクロに分割した後，全てのマクロを簡易合成プログラムで合成することで回路全体の合成を行う．

全体の流れと，論理回路図のネットリストから簡易論理合成プログラムまでのインタフェースを図 7 に示す．市販ツールなどで論理合成されたネットリストの回路分割（回路抽出の場合にはここで行う）を行い，テーブルとして出力する．回路分割したものをマクロと呼ぶ．また，マクロの分割は最大入出力数を任意に決めることが出来る．分割したマクロはマクロネットリストとして出力する．そして，それぞれのマクロネットリストの入出力から真理値表を作成する．その真理値表を簡易論理合成プログラムに入力し，論理式を生成する．このように回路を分割し，部分回路ごとに合成手法を適応させることで大規模回路にも簡単に対応することができる．回路分割方法については、4.1 で詳しく説明する．[5]

Vol.2016-ARC-218 No.19
Vol.2016-SLDM174 No.19
2016/1/20

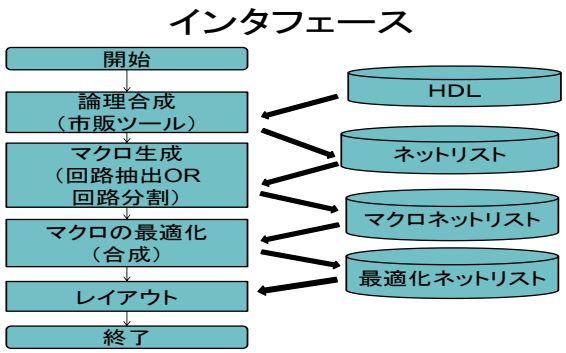

図 7 ネットリストインタフェースの図

4 並列処理

今回提案する簡易論理合成手法を，マルチコアプロセッサを用いて並列化することで高速に論理合成することを目指した．簡易合成方法を並列化手法に対応させるには回路分割方法と，その分割した回路をどのようにコアに割り当てて並列化するかが問題となる．本章では並列化に向けた回路分割方法と割り当て方法について述べる．

4.1 回路分割方法

回路を 3 入出力で分割する方法の例を以下の図 8 に示す．図の例では 8 入力(I1, I2, I3, I4, I5, I6, I7, I8)2 出力(O1, O2)となっている．回路の中身は論理素子 1, 2, 3, 4, 5, 6, 7 とフリップフロップ(FF)で形成される．回路を分割する条件は 2 つある．1 つ目は任意に指定した入出力数となるように分割する．例では(I3), (I4), (I5)と(I6), (I7), (I8)と(I1), (I2), (I3, I4, I5 の出力)での分割となる．2 つ目はフリップフロップを終点とした分割である．順序回路では，フリップフロップの状態が出力に影響するため，フリップフロップの前で分割する．

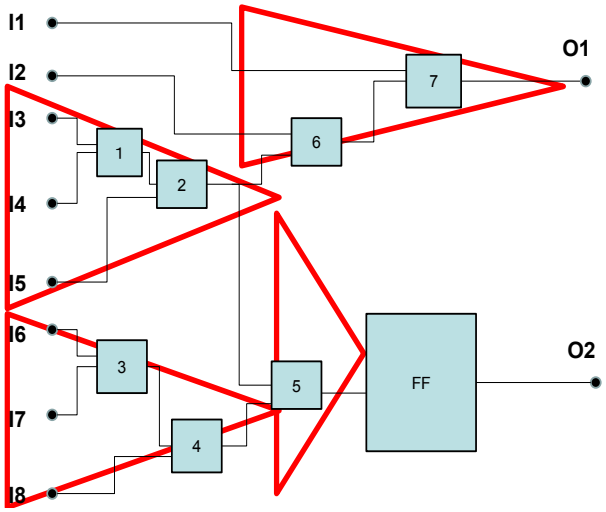

図 8 分割の図

4.2 並列化のための割当て方法

分割した回路をどのコア(並列)にどのように割当て，並列処理を行うのかを示す．今回の論理合成手法は真理値表の出力が 1 になる行を総当りにマージし比較していく．そのため，分割したマクロの出力が 1 になるものがコア(並列)により偏りがないように（極力均等になるように）割当てる．このように割当てることでマージ部分にかかる計算量を均等化できる．4 並列の例を表 1 に示す．表 1 では，マクロ名(出力が 1 の行の数)の形で表記する．マクロ名の番号は分割し生成される順に付与される．この表のように出来るだけコア間で出力が 1 になる行の数が偏りないように割当てる．この方法で割当てることにより効率よく(並列数に応じた理想的な高速化率で)処理することが出来る．

表 1 マクロの割当て

コア名	コア1	コア2	コア3	コア4
	マクロ23(31)	マクロ22(28)	マクロ18(27)	マクロ26(25)
	マクロ3(16)	マクロ2(16)	マクロ21(16)	マクロ25(20)
	マクロ9(16)	マクロ1(16)	マクロ4(16)	マクロ8(16)
	マクロ24(15)	マクロ7(15)	マクロ17(15)	マクロ12(15)
	マクロ10(8)	マクロ19(11)	マクロ16(12)	マクロ5(8)
	マクロ20(8)	マクロ13(8)	マクロ14(8)	マクロ11(8)
	マクロ30(5)	マクロ6(3)	マクロ15(1)	マクロ29(5)
			マクロ27(1)	
			マクロ28(1)	
1の行の合計	99	97	97	97

5 評価

性能評価としていくつかの回路に対して本手法を適応した結果をゲート数で比較する．また，本手法の処理速度を計測した．マルチコアによる並列化結果は逐次処理と 4 および 8 並列での処理速度を比較した．

5.1 評価方法

提案する論理合成の処理速度，合成結果について測定し評価を行なう．

以下に評価条件を示す．ゲート数の評価は 2 入力の AND,OR ゲートと NOT ゲートのみで行う．ゲートには様々な種類があり，XOR や NAND，NOR などがある．更に入力数 4 のゲートなどもある．今回の評価では 2 入力の AND，OR ゲートのみでの評価をすることにした．そのため，ゲート数は 2 入力の AND ゲート，OR ゲートそれから NOT ゲートの合計数とする．

並列処理による比較は，逐次処理に対して並列処理の処理時間の速度比で示すこととする．

5.2 評価回路

以下に評価回路を示す．

- decode16
- encode16
- pc
- sftreg16
- fadder8

・eccgc4
・spwm16
これらの回路は8ビットCPU を構成するブロックなど，実用的に使われる回路である．

5.3 評価環境

評価環境は以下の通りである．

1)PC 環境

OS：Windows 7
CPU：core i3　2.13GHz
RAM：4.00GB
コンパイラ：MinGW gcc

2)並列化環境

シミュレータ：JAXA-Elegant/Visual Spec(version4.1.6)
ターゲットCPU：ARM946E-S
コンパイラ：arm-elf-gcc
クロック周波数：200MHz

5　評価結果

5.1 回路合成結果

それぞれの回路に対して 3~5，8 入出力で評価したゲート数と市販ツールで論理合成した元のゲートを 2 入力 AND,OR,NOT ゲートに直したものを比較したものを表 2 に示す．回路分割の入出力数とは，回路を分割する際の制限として何入出力で分割したのかを示す．変換後のゲートとは本手法で変換したゲート数である．単位は個である．

表 2　回路合成結果

回路名	回路分割のマクロの入出力数	変換後のゲート数	市販論理合成ツールのゲート数
decode16	3	31	32
	4	32	
	5	33	
	8	32	
encode16	3	69	58
	4	61	
	5	59	
	8	53	
pc	3	76	72
	4	84	
	5	91	
	8	78	
sftreg16	3	55	74
	4	66	
	5	67	
	8	69	
fadder8	3	112	63
	4	113	
	5	114	
	8	113	
eccgc4	3	179	107
	4	126	
	5	155	
	8	215	
spwm16	3	285	143
	4	201	
	5	189	
	8	194	

この結果から市販ツールとほぼ同等のゲート数になるものと非常に増加してしまう回路がある．decode16 や encode16,sftreg16 に関しては元のゲート数よりも合成後のほうがゲート数を削減することが出来た．pc はほぼ同等となり，fadder8,eccgc4,spwm16 は増加してしまうという結果になった．fadder8 は一般的な加算器である．市販ツールのような論理合成では，このような一般的な回路は手設計の回路をライブラリに登録している．手設計の回路は最適化されており，最もゲート数の少ないような理想の回路である．そのため，市販ツールと比べると変換後のゲート数が増加した．eccgc4 では分割の入出力によってゲート数の差が大きい．4 入力での制限では 126 個と多少の増加であるが，8 入力での分割は 215 個となってしまう．この理由は eccgc4 が 4 ビットごとの処理を行う回路であるからである．そのため，4 入力での分割が分割の入力数において最もゲート数が少なくなった．spwm16 はデータパス部となるシフト演算を行う部分がある．市販ツールではデータパス部もライブラリから手設計の回路を用いている．そのため，変換後のゲートが増加してしまった．このように対象回路に合った入出力での回路分割や，データパス部のような回路を除くランダム回路では良い結果となることが分かった．

5.2 並列処理での処理時間

次に逐次処理とマルチコアプロセッサにより並列化した処理時間を比較した．単位はピコセカンド(ps)である．また，逐次処理と比べての高速化倍率も示す．

表 3　並列処理の結果

	逐次[ps] (a)	4並列[ps] (b)	8並列[ps] (c)	a/b[倍]	a/c[倍]
decode16	198,918,520,000	60,170,680,000	44,092,840,000	3.3	4.5
encode16	312,523,840,000	98,405,880,000	68,696,520,000	3.2	4.5
pc	95,754,280,000	33,252,200,000	25,745,200,000	2.9	3.7
sftreg16	787,797,280,000	207,100,640,000	105,478,720,000	3.8	7.5
fadder8	122,262,320,000	33,123,720,000	17,506,400,000	3.7	7.0
eccgc4	141,411,960,000	49,914,000,000	30,379,760,000	2.8	4.7
spwm16	1,096,184,080,000	278,634,680,000	142,170,000,000	3.9	7.7

並列処理を各回路に適応した結果，4 並列では最大で約 3.9 倍 (spwm16) ，8 並列では約 7.7 倍 (spwm16) の高速化を実現できる結果となった．この結果から，理想であるコア数に比例した高速化ができた．しかし，pc や eccgc4 などの一部の回路ついては並列数の 6~7 割程度の高速化しかできないという結果になった．

そこで，実際に最も高速化率が高い回路(4 並列: spwm16，8 並列：spwm16)と，最も高速化率の良くない回路(4 並列：eccgc4，8 並列：pc)を並列数ごとに各処理ブロックがどのくらいの割合を占めるかを比較した．(表 4)各処理のサイクル数と()の中は全体に対する割合[%]を示す．また，各処理の内容については表の下に示す．

表 4 各処理ブロックごとのサイクル数の割合

		read	merge	entry	make
4並列	eccgc4	5255216(42.67)	6316329(51.28)	685728.5(5.57)	70116.5(0.57)
	spwm16	25540020(23.5)	79176267.5(72.85)	3791969.5(3.49)	184916.5(0.17)
		read	merge	entry	make
8並列	pc	6200428(44.25)	7161102(51.11)	589622(4.21)	66636.5(0.48)
	spwm16	49758904(23.14)	157448969.5(73.23)	7474674(3.47)	352466(0.16)

表の数字の単位：サイクル数（割合[%]）

・read:真理値表を読み込む部分
・merge:マージを行う部分
・entry:必要な行を選択する部分
・make:論理式を生成する部分

この表から 4 並列, 8 並列どちらも理想の高速化率を得られなかった回路ではマージにかかる割合が少ないということが分かった．今回は出力が 1 になる行に差が出ないようにマクロを割当てた．このような方法をとった理由は，マージを行う際に出力が 1 になる行を比較していくからである．しかし，この方法では出力が 1 になる行はコアごとに差がなく割当てられるが，コアに割当てるマクロの数は偏る場合がある．そのため，真理値表の読み込みや論理式の生成部分がコアによって偏り，理想的な高速化が見込めなかったと考えることができる．今後はマクロ数と出力が 1 となる行のどちらも差が出ないような割当て方法を考える必要がある．

6 結論・まとめ

本論文では， 対象回路を小規模回路に分割し、分割された回路を簡易論理合成方法により高速に合成する手法を提案した．さらに，マルチコアプロセッサを用い，並列処理を行うことで更なる高速化を図った。本手法を実用的な回路に適応した結果，ランダム回路については手設計とほぼ同等の結果を得ることができた。ゲート数が増加してしまう回路の理由は，市販ツールでの四則演算やシフト演算などを行う一般的な回路では，あらかじめ手設計で設計された最適化されている回路をライブラリに入れており，そのライブラリの回路を用いることでよい回路を生成している．そのため，そのような一般的な回路では，市販ツールと比べゲート数が増えたと考える．さらに，マルチコアによる高速化を適応すると，4 並列で最大で 3.9 倍，8 並列で最大 7.7 倍とコア数に応じた高速化を実現することができた．しかし，中には理想的な高速化率が得られないものもあった．理想的な高速化率が得られなかった回路に関しては，コアに対する割当て方法に問題があることがわかった．今回はマージ部分がコア間で差が出ないように割当てたが，理想的な高速化率が得られなかった回路ではマージにかかる割合が少なかった．そのために理想的な高速化率を得ることができなかった．

7 今後の課題

今回の評価より，並列や回路に応じた最適な割当て方法，入出力・並列数の増加，大規模回路への対応，消費電力や遅延時間が課題として挙げられる．

1) 最適な並列化方法

今回は出力が 1 となる行をコア(並列)ごとに差が出ないように割当てた方法で実験を行った．これはマージ部分が最も時間のかかる処理であるため，マージ部分がコアによって差が出ないようにするためである．しかし，回路ごとのマージにかかる割合が異なるため，理想的な高速化率を得ることができないものがあった．そのため，どのような回路でも理想的な高速化率を得られるような新たな割当て方法が必要となる．

2) 入出力数の増加や並列数の最適化

今回の評価では主に 3～8 入出力での回路に分割し，評価を行った．しかし，更に効率のよい入出力に分割できる可能性がある．そのため，よりよい入出力での分割を知ることが必要である．また，その際に並列数を増やしても並列倍の高速化を実現することが必要である．

3) 大規模回路への対応

今回の評価は実用的な中小規模の回路であったが，大規模回路での評価が出来ていない．そのため，大規模回路での評価とその対策が必要となる．

4) 消費電力・遅延時間

今回の方法では回路面積を評価の基準とした．しかし，実際には消費電力や遅延時間に対応した回路生成も必要となってくる．そこで，スイッチング回数や論理段数の削減に着目し，どのように論理合成すれば省電力や遅延のない回路を簡易的に設計できるかが課題となる．

参考文献

[1] 大菊祥子 , 豊永昌彦 ,”高速論理検証法と ECO 論理合成法の研究” 平成 25 年度高知大学修士論文

[2] 蘆苅将大，大菊祥子，村岡道明，豊永昌彦,” 部分論理回路の簡易論理最適化手法の提案” 平成25 年度 電気関係学会四国支部連合大会

[3] Wenlong Yang , Lingli Wang , Alan Mishchenko , “Lazy Man’s Logic Synthesis” , Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference

[4] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, Daniel D. Gajski, “システム設計：SpecC による実現”, 2001 年

[5] 竹内勇矢, トウブンチク, 村岡道明 ”並列化アルゴリズムによる論理シミュレーションの高速化手法の提案”, DA シンポジウム 2013 論文集, pp.91-96, 2013 年 8 月 22 日

[6] 竹内勇矢, 豊永昌彦, 村岡道明, "マルチコアを用いた高速並列論理シミュレーション手法", SLDM 研究会 2015, 2015 年 1 月