Vol.2010-SLDM-144 No.19
Vol.2010-EMB-16 No.19
Vol.2010-MBL-53 No.19
Vol.2010-UBI-25 No.19
2010/3/26

ハードウェア合成用Cベース設計記述向けタスクレベルパイプライン化手法

浅野 裕史，瀬戸 謙修，丸泉 琢也[†1]

動作合成は C 記述からハードウェアを自動合成できるため，LSI の設計期間短縮に効果を発揮する。しかしながら，C 記述からそのまま動作合成によって生成されたハードウェアの性能は不十分なことが多く，人手による C 記述の並列化が必要となることが多い．本稿では，そのような C 記述並列化の一種である，タスクレベルパイプライン化の自動化に向けた手続きを検討した．提案する手順を手動により JPEG エンコーダの C 記述に適用してタスクレベルパイプライン化した結果，適用前と比べて性能が 1.81 倍向上した．

Task Level Pipelining of C-Based Design Description for Hardware Synthesis

Hirofumi Asano，Kenshu Seto，Takuya Maruizumi[†1]

Behavioral synthesis automatically generates hardware from C descriptions, and therefore reduces the design time of LSIs significantly. The generated hardware by behavioral synthesis with these C descriptions untouched, however, most likely leads to insufficient performance, so that manual parallelization of the C descriptions is usually necessary. In this paper, we study a procedure for task-level pipelining, one of such parallelization techniques for C descriptions that can be used for the automation. By applying the proposed procedure manually to the C description for JPEG encoder and performing the task-level pipelining, the execution speed improved by 1.81 times compared to the C description without our technique.

†1 東京都市大学
Tokyo City University

1. はじめに

動作合成は，C 記述からハードウェアを自動合成する技術であり，LSI の設計期間短縮に効果を発揮するため，実設計に使われはじめている[1]。しかし，与えられた C 記述に対してそのまま動作合成を適用して生成されたハードウェアの性能は不十分なことが多い。そのため，現状では，動作合成を適用する前に，設計者は C 記述のアルゴリズムを理解した上で，人手による C 記述の最適化を行う。

そのような最適化の一つに，タスクレベルパイプライン化がある[3][4]。動画処理や信号処理のアルゴリズムでは，一番外側のループの中に，複数の連続する関数呼出しを持つことが多いが，タスクレベルパイプライン化では，それらの関数呼出しをループをまたいで並列実行することにより，最小限のリソース増加のもとで，性能向上が達成できる。現状では，タスクレベルパイプライン化は人手によって行われているため，設計期間の増大や設計誤りの混入を招き，動作合成の利点を打ち消してしまう方向に働いてしまう。

逐次的な C 記述からの自動タスクレベルパイプライン化について，マルチコアプロセッサを対象としたコンパイラの分野でいくつかの手法が提案されている。Thies らは，自動タスクレベルパイプライン化に必要となるタスクグラフを，シミュレーションベースの方法で自動構築する方法を提案した[4]。しかし，タスクレベルパイプライン化を行うためには，人手で適切な位置にパイプラインの境界などを示すマクロを C 記述中に挿入する必要がある。一方，Kudlur らは，8 個の SIMD 型プロセッサを持つ Cell プロセッサを対象として，与えられた C 記述（実際には，ストリーム処理用言語) の並列化を行い，メインメモリとローカルメモリの DMA 転送も考慮したタスクレベルパイプライン化技術を提案し，性能向上を達成した[3]。この研究でも，本研究同様，モジュロスケジューリング[5] をベースとしているが，マルチコアプロセッサ向け技術であり，本研究とはターゲットとするアーキテクチャが根本的に異なる。ハードウェア合成を対象とした研究として，同じく Kundlur らは，複数のループを持つプログラムから，各ループを実行するハードウェアを接続，並列動作させることで，タスクレベルパイプライン化を実現する方法を提案した[2]。彼らの方法は，与えられたスループットを満たすように，各ループをループパイプライン化する際の最適なパイプライン間隔を決定するとともに，複数ループを一つのハードウェアにまとめることでリソース共有により面積の削減を達成しているが，バッファメモリの共有を行っていないため，バッファメモリが最終的なハードウェア面積の大きな割合を占めてしまっている。本研究では，バッファメモリの共有が可能な，タスクレベルパイプライン化の手続きを提案する。

Vol.2010-SLDM-144 No.19
Vol.2010-EMB-16 No.19
Vol.2010-MBL-53 No.19
Vol.2010-UBI-25 No.19
2010/3/26

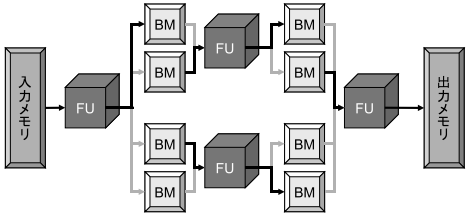

図 1 合成ハードウェア

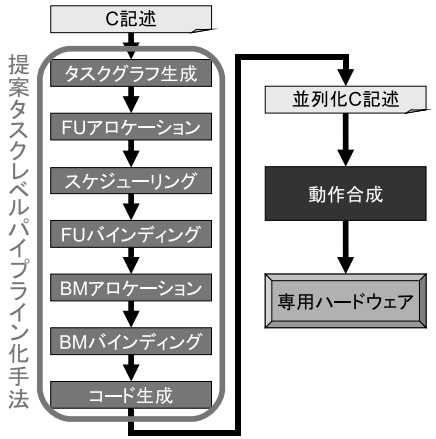

図 2 タスクレベルパイプライン化手順

```
1 int DI[M], DO[M];
2 int D1[B], ..., D4[B];
3
4 for (i=0; i<MAX; i++) {
5   FA(DI, D1);   // T1
6   FB(D1, D2);   // T2
7   FC(D2, DO);   // T3
8   FA(DI, D3);   // T4
9   FB(D3, D4);   // T5
10  FC(D4, DO);   // T6
11 }
```

図 3 C 記述

```
1 int DI[M], DO[M];
2 int BM1[B], ..., BM7[B];
3
4 for (i=0; i<MAX+2; i++) {
5   par {
6     FB(BM1, BM2);   // T2
7     FB(BM4, BM5);   // T5
8     {
9       par {
10        FA(DI, BM7);// T4
11        FC(BM3, DO);// T3
12      }
13      par {
14        FA(DI, BM3);// T1
15        FC(BM4, DO);// T6
16      }
17    }
18  }
19 }
```

図 4 並列化 C 記述

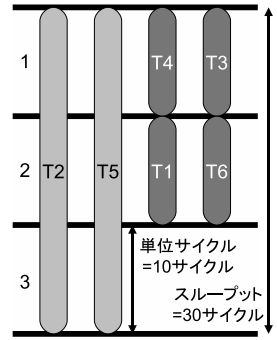

図 5 タスク生存期間

2. タスクレベルパイプライン化

この節では，本研究で提案するタスクレベルパイプライン化手法について説明する．本節では，2.3 小節以降，同じ例題を用いて本手法を解説する．

2.1 合成ハードウェア

図 1 にタスクレベルパイプライン化されたアルゴリズムをハードウェア合成した結果のモデルを示す．入力データ用と出力データ用に専用のメモリを持つ．タスクを処理するファンクションユニット（FU）が，中間データを蓄えるバッファメモリ（BM）により接続される．各 FU は独立したスレッドとして，並列に動作可能である．

2.2 タスクレベルパイプライン化手順の概要

図 2 に提案するタスクレベルパイプライン化手法の手順を示す．入力は C 記述，出力は並列化 C 記述である．タスクグラフ生成では，タスク間の依存関係を解析する．FU アロケーションでは，FU の種類と数を決定する．スケジューリングでは，FU アロケーション結果を基にタスクグラフのパイプラインスケジューリングを行う．FU バインディングでは，FU にタスクを割り当てる．BM アロケーションでは，BM の数を決定する．BM バインディングでは，中間配列に BM を割り当てる．コード生成では，並列化 C 記述を出力記述として生成する．

2.3 入出力記述

図 3 に入力記述の例を示す．入力記述はアルゴリズムを記述した C 記述である．このアルゴリズムはループ中に連続する複数の関数呼び出しを持つ．以降，この関数呼出しをタスクとして扱う．図 3 の例では，6 個の関数呼び出しをタスク T1〜T6 として定義する．各タスクから呼び出される関数 FA()，FB()，FC() はそれぞれ異なる FU として合成される．この合成結果から，各 FU の処理サイクル数が与えられる．配列 DI は入力配列，配列 DO は出力配列，配列 D1〜D4 は中間配列である．入力配列は入力メモリ，出力配列は出力メモリ，中間配列は BM として合成される．

図 4 に出力記述の例を示す．出力記述は C ベース設計言語により記述された並列化 C 記述である．本手法で用いる C ベース設計記述では，並列構文“ par ”を用いて記述の並列性を明示できる．並列構文“ par ”内に記述された文は並列に実行される．並列実行される文の同期は並列構文“ par ”の開始時と終了時に行われる．入れ子にすることも可能である．図 4 では，並列構文“ par ”を用いて，関数呼び出しを並列実行し，パイプラインを実現している．配列 BM1〜BM7 は中間配列であり，図 3 における中間配列 D1〜D4 に相当する．図 5 に図 4 の実行例を示す．並列構文“ par ”で囲まれたタスクは，同時に開始され，処理が終了しても並列化されている他の全てのタスクが終了するまで待ち続ける．全てのタスクの処理が終了して，並列構文“ par ”が完了する．したがって，開始時と終了時に同期をとる．例では，期間 1 において 5 行目の“ par ”により囲まれるタスク T2，T5，8 行目か

Vol.2010-SLDM-144 No.19
Vol.2010-EMB-16 No.19
Vol.2010-MBL-53 No.19
Vol.2010-UBI-25 No.19
2010/3/26

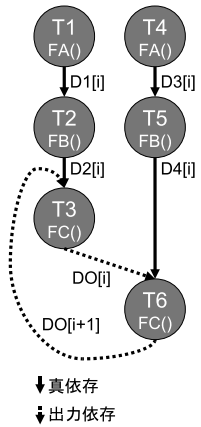

図 6 タスクグラフ

表 1 FU アロケーション結果

FU	FU_A	FU_B	FU_C
関数名	FA()	FB()	FC()
タスク	T1,T4	T2,T5	T3,T6
サイクル数	10	25	10
FU数	1	2	1

ら始まるコードブロックが同時に開始される．8 行目から始まるコードブロック内の 13 行目の“ par ”で囲まれるタスク T1，T6 は，9 行目の“ par ”で囲まれるタスク T4，T3 の終了後に実行される．期間 1 においてタスク T2，T5，9 行目の“ par ”で囲まれるタスク T4，T3 が並列実行される．期間 2 では，タスク T2，T5 は期間 1 から引き続き並列に実行される他，13 行目の“ par ”で囲まれたタスク T1，T6 が並列に実行される．期間 3 ではタスク T2，T5 が終了し，全てのタスクが完了する．

2.4 タスクグラフ生成

図 6 に図 3 の入力記述から生成したタスクグラフを示す．タスクグラフ生成では，入力記述からタスク間に生じる依存関係を解析する．図 6 のノードはタスク，エッジはタスク間の依存関係である．タスクは関数呼び出し，依存関係はタスク間で共有される変数により生じる．図 6 では，タスク T1〜T6 をノード，中間配列 D1〜D4 により生じる真依存，出力配列 DO により生じる出力依存をエッジとしてグラフを生成している．エッジの添え字“ i ”は現在のイタレーション，“ i+1 ”はその次のイタレーションで生じる依存関係を示す．図 6 の例を見ると，タスク T1 → T2 間では，中間配列 D1 がタスク T1 で書き込まれ，タスク T2 で読み出されることにより生じる真依存を表す．その他の中間配列 D2〜D4 によるエッジも同様である．タスク T3 → T6 間では，配列 DO がタスク T3 で書き込まれた後に，連続してタスク T6 で書き込まれる必要があることから生じる出力依存を表す．

表 2 スケジューリング結果

サイクル	FU_A	FU_B1	FU_B2	FU_C
1-10				
11-20	T4			
21-30	T1			
31-40		T2	T5	
41-50				
51-60				
61-70				
71-80				T3
81-90				T6

表 3 パイプラインカーネル

サイクル	FU_A	FU_B1	FU_B2	FU_C
1-10				
11-20	T4[i]	T2[i+1]	T5[i+1]	T3[i+2]
21-30	T1[i]			T6[i+2]

表 4 ハードウェア予約表

サイクル	FU_A	FU_B1	FU_B2	FU_C
1-10				
11-20				
21-30				

表 5 タスクのスケジューリング順序取得結果

タスク	ASAP	ALAP	移動度	深さ	高さ	順序
T1	10	10	0	10	60	4
T2	40	40	0	40	50	3
T3	50	50	0	50	20	2
T4	10	20	10	10	50	6
T5	40	50	10	40	40	5
T6	60	60	0	60	10	1

2.5 FU アロケーション

表 1 に図 6 の FU アロケーション結果を示す．FU アロケーションでは，各 FU の個数を設計者が決定する．表 1 では，FU の種類，FU で処理される関数名，FU で処理するタスクの種類，FU のハードウェア合成結果からわかる処理サイクル数，設計者が決めた FU の個数をまとめる．各 FU の処理サイクル数は，あらかじめわかっているものとする．本手法は，FU の処理サイクル数がデータにより変化する場合にも対応可能である．

2.6 スケジューリング

表 2 に図 6 のタスクグラフの 1 イタレーション分のスケジューリング結果を，表 3 にパイプラインカーネルのスケジューリング結果を示す．スケジューリングでは，図 6 のタスクグラフに対し，表 1 の FU アロケーション結果を用いて，パイプラインスケジューリングを行う．タスクレベルにおけるスケジューリングの単位サイクル数は設計者が設定する．ここでは単位サイクル数を 10 とする．スループットは，処理サイクル数が最も大きい FU を実行するのに必要となる単位サイクル数の倍数の内，最小のものに，その FU が 1 イタ

Vol.2010-SLDM-144 No.19
Vol.2010-EMB-16 No.19
Vol.2010-MBL-53 No.19
Vol.2010-UBI-25 No.19
2010/3/26

レーションの内に最低限実行しなければならないタスクの数を掛けたものである．表 1 の例では，FU_B が最大の処理サイクル数，25 サイクルを持つ．この実行に最小限必要な単位サイクル数の倍数は 30 サイクルである．FU_B は 2 個であり，タスク T2，T5 の 2 個のタスクを実行しなければならないことから，FU_B が 1 個あたりで実行しなければならないタスク数は 1 個である．すなわち，30 × 1 = 30 サイクルがスループットとなる．

本研究では，スケジューリングアルゴリズムとして，命令レベルのパイプラインスケジューリングとしてレジスタ削減で高い効果が期待できる Swing Modulo Scheduling（SMS）を応用する．SMS ではスケジューリングを行う際のタスクの順序と方向を， ASAP スケジューリング，ALAP スケジューリングの結果から求める．表 5 に図 6 のスケジューリング順序の取得結果を示す．タスクレベルのパイプラインスケジューリングでは，SMS と同様にハードウェア予約表を用いてスケジューリングを行う．表 4 にハードウェア予約表を示す．提案するタスクレベルのパイプラインスケジューリング手法では，並列構文“ par ”の制限から，タスクがスループットをまたいで実行できない．そのため，パイプライン化した際にタスクがスループットを超過することを防ぐため，ハードウェア予約表に従い，タスクがスループットを超過しないようにスケジューリング位置を移動させる．表 2 のスケジューリング結果をスループット単位で重ねることで，表 3 のパイプラインカーネル生成できる．添え字は，“ i ”を現在のイタレーションとして，相対位置を示す．

2.7 FU バインディング

FU バインディングでは，スケジューリング結果を基に FU にタスクを割り当てる．本手法では，スケジューリング完了と同時に，FU バインディングも完了する．FU バインディング結果は表 3 に一致する．

2.8 BM アロケーション

表 6 に BM の使用期間，表 7 に BM アロケーション結果を示す．BM アロケーションでは，タスクグラフ 1 イタレーション分のスケジューリング結果から，BM の使用期間が求まる．その結果をスループット単位で重ねあわせることにより，パイプラインの実現に必要となる BM の個数が求まる．

2.9 BM バインディング

表 8 に BM バインディング結果を示す．BM バインディングでは，FU バインディング，BM アロケーション結果から，中間配列に BM を割り当てる．BM の割り当てにはラウンドロビン方式を用いる．表 8 の例では，BM1〜BM3 は 90 サイクル（3 イタレーション）毎に，BM4〜BM7 は 60 サイクル（2 イタレーション）毎に同じ割り当てとなる．

表 6 BM 使用時期間

サイクル	D1	D2	D3	D4
1-10				
11-20				
21-30				
31-40				
41-50				
51-60				
61-70				
71-80				
81-90				

Write
Read
Store

表 7 BM アロケーション結果

サイクル	D1	D2	D3	D4	BM数
1-10					6
11-20					7
21-30					7

表 8 BM バインディング結果

タスク	BM1	BM2	BM3	BM4	BM5	BM6	BM7
1-10	RD1	WD2	SD2	RD3	WD4	SD4	
11-20			RD2				WD3
21-30			WD1			RD4	SD3
31-40	WD2	SD2	RD1	WD3	SD4	WD4	RD3
41-50		RD2					
51-60		WD1		SD3	RD4		
61-70	SD2	RD1	WD2	RD3	WD4	SD4	
71-80	RD2						WD3
81-90	WD1					RD4	SD3
91-100	RD1	WD2	SD2	WD3	SD4	WD4	RD3
101-110			RD2				
111-120			WD1	SD3	RD4		

```
1 if (! ((i==0)||(i==MAX+1))) {
2   if (st2==0) {
3     FB(BM1, BM2);
4   }
5   else if (st2==1) {
6     FB(BM3, BM1);
7   }
8   else if (st2==2) {
9     FB(BM2, BM3);
10  }
11 }                          // T2
```

図 7 タスク記述例

2.10 コード生成

図 4 に，パイプラインカーネルの記述例を示す．コード生成では，4 パイプラインカーネルのスケジューリング結果，BM バインディング結果から出力記述を生成する．出力記述では，並列構文“ par ”を用いることで，パイプラインカーネルについてスケジューリングの単位サイクル数でタスクの同期をとる．複数単位サイクル数にまたがるタスクは，そのタスクの開始と終了でのみ同期をとる．

また，図 7 にタスクへの BM 割り当て例，及び，パイプラインの開始状態，終了状態の

Vol.2010-SLDM-144 No.19
Vol.2010-EMB-16 No.19
Vol.2010-MBL-53 No.19
Vol.2010-UBI-25 No.19
2010/3/26

```
int DY[M], DCB[M], DCR[M];
int DO[M];
int D1[B], ..., D18[B];
int p_y1, p_y2, ..., p_cr;
int dc_y, dc_cb, dc_cr;

for (i=0; i<YSIZE; i+=YSTEP) {
  for (j=0; j<XSIZE; j+=XSTEP) {
    block(DY, D1, p_y1);     // BY1
    dct(D1, D2);             // DY1
    quantize(D2, D3);        // QY1
    encode(D3, DO, &dc_y);   // EY1

    block(DY, D4, p_y2);     // BY2
    dct(D4, D5);             // DY2
    ...

    block(DCR, D16, p_cr);   // BCR
    dct(D16, D17);           // DCR
    quantize(D17, D18);      // QCR
    encode(D18, DO, &dc_cr) // ECR
  }
}
```

図 8 JPEG エンコーダ記述

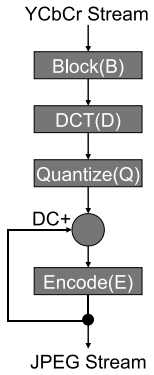

図 9 JPEG エンコーダブロック図

記述例を示す．BM の切り替えは分岐構文“ if ”を用いて，状態変数を用いてで行う．図 7 の例ではタスク T2 を例に，状態変数“ st2 ”を用いて状態遷移を実現する．開始状態と終了状態については，分岐構文“ if ”を用いてタスクを実行するイタレーションを指定することで実現する．イタレーションは変数“ i ”により決まる．

3. 実　　験

提案するタスクレベルパイプライン化手法の効果を検証するために，JPEG エンコーダの C プログラムに提案手法を適用した．提案手法の適用前，適用後それぞれのハードウェア合成結果について速度を比較することで，本手法によるハードウェアの高速化の効果を検証した．同時に，面積への悪影響も確認した．

3.1 実 験 方 法

図 8 に実験に用いた JPEG エンコーダの記述，図 9 にブロック図を示す．実験に用いた JPEG エンコーダは，Independent JPEG Group（IJG）の提供する JPEG エンコーダの C プログラムを簡単化したものである．最適化前の行数は約 2300 行である．JPEG エンコーダのメインアルゴリズムを入力記述，パイプライン化されたアルゴリズムを出力記述とする．動作合成には“ Sharp Bach System ver.3.6a ”を用いた．動作合成結果の論理検証には“ Mentor Graphics ModelSim SE ver.6.2e ”を，論理合成には“ Synopsys Design Compiler ver.W-2004.12-SP2 ”用いた．プロセスライブラリとして，“ Oklahoma State University FreePDK ver.2.7 ”を用いた．以上のツールを用いて， TSMC 0.18 μ m プロセスのデータを利用して，周波数が 100MHz の ASIC としてハードウェア合成した．

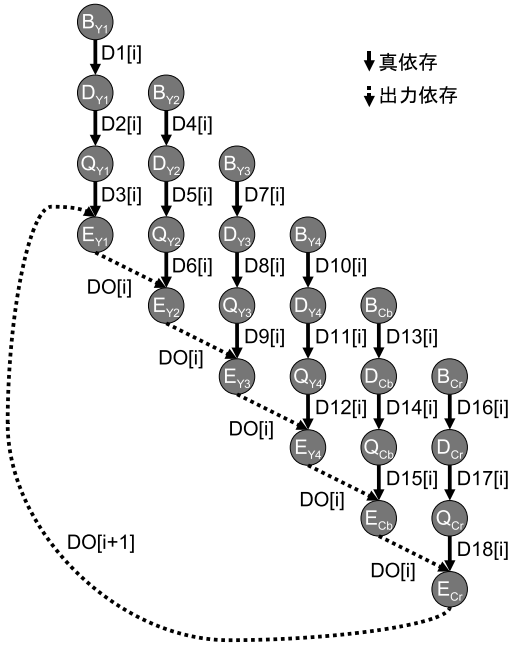

図 10 JPEG エンコーダタスクグラフ

表 9 JPEG エンコーダ FU アロケーション結果

FU	FU_B	FU_D	FU_Q	FU_E
関数名	block()	dct()	quantize()	encode()
タスク	B_{Y1},B_{Y2}, B_{Y3},B_{Y4}, B_{Cb},B_{Cr}	D_{Y1},D_{Y2}, D_{Y3},D_{Y4}, D_{Cb},D_{Cr}	Q_{Y1},Q_{Y2}, Q_{Y3},Q_{Y4}, Q_{Cb},Q_{Cr}	E_{Y1},E_{Y2}, E_{Y3},E_{Y4}, E_{Cb},E_{Cr}
サイクル数	64	128	64	260
FU数	1	1	1	1

3.2 実 験 結 果

図 10 に実験対象とした JPEG エンコーダのタスクグラフを，表 9 に FU アロケーション結果を示す．実験対象とした JPEG エンコーダは，4 種 24 個のタスクを持つ．関数 block()，dct()，quantize() はの処理サイクル数は一定である．関数 encode() の処理サイクル数は入力データにより変化するため，今回は平均値である 260 サイクルを用いた．パイプライン化されていない状態での JPEG エンコーダのスループットは，3076 サイクルである．

表 10 にスケジューリングの単位サイクル数を 65 サイクルとしてスケジューリングを行っ

Vol.2010-SLDM-144 No.19
Vol.2010-EMB-16 No.19
Vol.2010-MBL-53 No.19
Vol.2010-UBI-25 No.19
2010/3/26

表 10 JPEG エンコーダ スケジューリング結果

サイクル	FU_B	FU_D	FU_Q	FU_E
1-65	B_{Y1}			
66-130		D_{Y1}		
131-195				
196-260			Q_{Y1}	
261-325	B_{Y2}			
326-390		D_{Y2}		E_{Y1}
391-455				
456-520			Q_{Y2}	
521-585	B_{Y3}			
586-650		D_{Y3}		E_{Y2}
651-715				
714-780			Q_{Y3}	
781-845	B_{Y4}			
846-910		D_{Y4}		E_{Y3}
911-975				
976-1040			Q_{Y4}	
1041-1105	B_{Cb}			
1106-1170		D_{Cb}		E_{Y4}
1171-1235				
1236-1300			Q_{Cb}	
1301-1365	B_{Cr}			
1366-1430		D_{Cr}		E_{Cb}
1431-1495				
1496-1560			Q_{Cr}	
1561-1625				
1626-1690				E_{Cr}
1691-1755				
1756-1820				

表 11 JPEG エンコーダ パイプラインカーネル

サイクル	FU_B	FU_D	FU_Q	FU_E
1-65	$B_{Y1}[i]$			
66-130		$D_{Y1}[i]$		$E_{Cr}[i+1]$
131-195				
196-260			$Q_{Y1}[i]$	
261-325	$B_{Y2}[i]$			
326-390		$D_{Y2}[i]$		$E_{Y1}[i]$
391-455				
456-520			$Q_{Y2}[i]$	
521-585	$B_{Y3}[i]$			
586-650		$D_{Y3}[i]$		$E_{Y2}[i]$
651-715				
714-780			$Q_{Y3}[i]$	
781-845	$B_{Y4}[i]$			
846-910		$D_{Y4}[i]$		$E_{Y3}[i]$
911-975				
976-1040			$Q_{Y4}[i]$	
1041-1105	$B_{Cb}[i]$			
1106-1170		$D_{Cb}[i]$		$E_{Y4}[i]$
1171-1235				
1236-1300			$Q_{Cb}[i]$	
1301-1365	$B_{Cr}[i]$			
1366-1430		$D_{Cr}[i]$		$E_{Cb}[i]$
1431-1495				
1496-1560			$Q_{Cr}[i]$	

表 12 JPEG エンコーダ ハードウェア合成結果

	適用前	適用後	比率
スループット [サイクル]	3076	1560	0.5
BM数	2	3	1.5
速度 [画素/サイクル]	0.072	0.130	1.81
面積 [/NANDゲート]	105141	84795	0.81

た結果を，表 11 にパイプラインカーネルを示す．表 11 を見ると，3076 サイクルであったスループットが，パイプライン化により 1560 サイクルとなり，約 1/2 となる．したがって，2 倍の性能向上が期待できる．

表 12 に JPEG エンコーダのハードウェア合成結果を示す．速度を比較すると，提案手法の適用後には約 1.81 倍の性能向上が見られる．2 倍に届かなかったのは，関数 encode() の処理サイクル数がスループットを上回ることが複数回発生したためだと考えられる．一方，面積を比較すると，提案手法の適用後では面積が 0.81 倍に削減された．順序回路が NAND 換算で約 1200 ゲート増加したが，組み合わせ回路は約 21500 ゲート減少した．

4. おわりに

本研究は，集積回路の設計期間を短縮するために，タスクレベルパイプライン化の自動化を目指し，タスクレベルパイプライン化手順を体系化したものである．提案手法の有効性を確認するために，JPEG エンコーダの C プログラムを例題として，提案するタスクレベルパイプライン化手法を適用した．本手法を適用した結果，提案手法適用前と比較して，合成されたハードウェアの性能が 1.81 倍に向上した．一方で，提案手法を適用したハードウェアの面積は，適用前と比較して，0.81 倍に減少した．この実験結果から本研究で提案するタスクレベルパイプライン化手法が，アルゴリズムから集積回路ハードウェアを自動生成するための最適化手法として効果が非常に期待できることがわかった．

謝　辞

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，シノプシス株式会社，メンターグラフィクス株式会社，シャープ株式会社の協力で行われたものである．

参 考 文 献

1) Takashi Kambe, *et al*, “A C-based Synthesis System, Bach, and its Application,” The 2001 Asia and South Pacific Design Automation Conference, pp.151-155, 2001.
2) Manjunath Kudlur, *et al*, ”Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines,” The 4th International Conference on Hardware/Software Codesign and System Synthesis, pp.270-275, 2006.
3) Manjunath Kudlur, *et al*, ”Orchestrating the Execution of Stream Programs on Multicore Platforms,” ACM SIGPLAN Notices Volume 43, pp.114-124, 2008.
4) William Thies, *et al*, “A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs,” The 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp.356-369, 2007.
5) Josep Llosa, *et al*, “Lifetime-Sensitive Modulo Scheduling in a Production Environment,” The 2001 IEEE Transactions on Computers, pp.234-249, 2001.
6) Greg Snider, “Performance-Constrained Pipelining of Software Loops onto Reconfigurable Hardware,” The 2002 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp.177-186, 2002.