C-008

OpenMP ハードウェア動作合成システム：コードジェネレータの実装と画像処理による評価

Design, Implementation and Evaluation of a Hardware Behavioral Synthesis System with OpenMP

松崎 裕樹† 中谷嵩之† 山崎 勝弘†
Hiroki Matsuzaki Takayuki Nakatani Katsuhiro Yamazaki

1. はじめに

LSI の開発期間を短縮するため，C ベース言語のハードウェア動作合成手法が設計に利用されている[1]．しかし，C 言語のような抽象度の高い言語では，空間的な概念や並列動作の概念が含まれておらず，自動合成で最適なハードウェアが合成されるとは限らない．また，並列化したハードウェアの検証において，主に RTL レベルのシミュレータを用いるため，検証に時間がかかるという問題もある．本研究では，これらの問題を解決するために，並列プログラミングに使用される OpenMP を用いたハードウェア動作合成手法を提案し，システム全体を設計・実装して，画像処理など各種応用に対する評価を行うことを目的とする．我々は昨年度 OpenMP プログラムから中間表現に変換するトランスレータを実装し，信号処理に対する動作合成の有効性を示した[2]．本研究では，生成された中間表現から対応するハードウェアを動作合成するコードジェネレータを実装し，画像処理アルゴリズムであるエッジ検出とハフ変換に対して，本システムを用いて動作合成を行い，実行速度，回路規模，並列効果の観点からシステムを評価する．

2. OpenMP ハードウェア動作合成システム

2.1 OpenMP による動作合成システムの構成

OpenMP とは，SMP 環境における並列プログラミングの標準 API である．C，Fortran のプログラミング言語にプラグマを追加することにより，繰り返し処理を並列化する並列リージョン，及び複数の異なる処理を並列化する並列セクションの範囲を指示することができる．逐次プログラムに対して，共有変数や並列動作するノード数などの並列化構造を容易に追記可能である．

ハードウェア動作合成システムの構成を図１に示す．本研究で提案するハードウェア動作合成システムは，アルゴリズム評価系とハードウェア動作合成系で構成される．アルゴリズム評価系では，動作記述として書かれた OpenMP プログラムを，OpenMP コンパイラによってマルチスレッドプログラムに変換し，PC クラスタやマルチコア PC などの SMP 環境で実行して，アルゴリズムの検証と並列化の評価を行う．SMP 環境で高速な実行ができるため，大規模な問題の検証時間の短縮と，並列化アルゴリズムの評価を設計早期に行うことが可能である．ハードウェア動作合成系では，検証後の OpenMP プログラムをトランスレータで中間表現に変換した後，コードジェネレータで並列動作ハードウェアを生成する．中間表現には，OpenMP で指定された並列化情報が含まれており，コードジェネレータではそれらを用いて最適化を行い，並列動作ハードウェアを生成する．

† 立命館大学大学院 理工学研究科, Graduate School of Science and Engineering, Ritsumeikan University

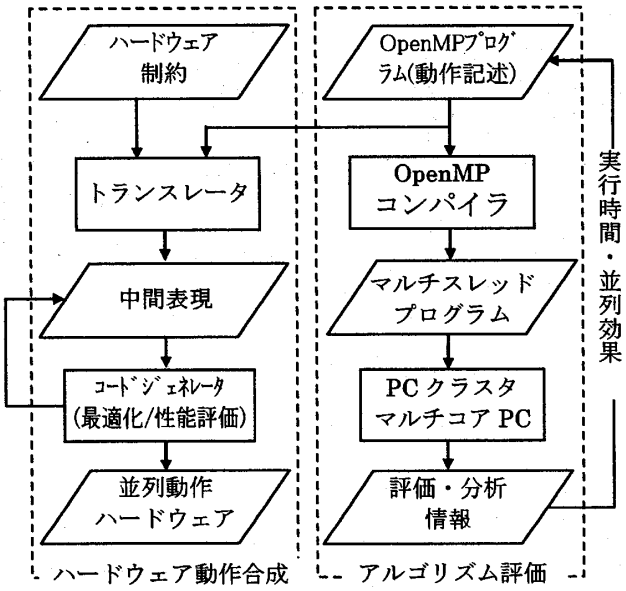


図 1：OpenMP によるハードウェア動作合成システム

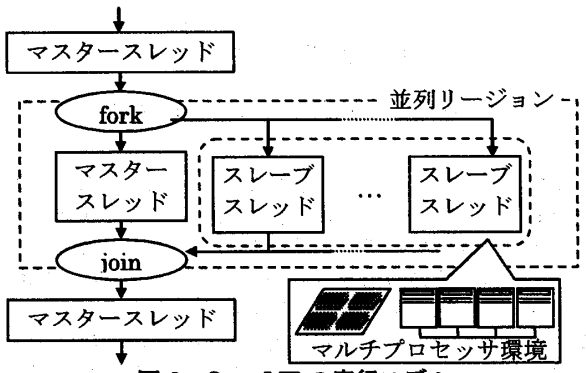


図 2：OpenMP の実行モデル

2.2 OpenMP の実行モデルと生成ハードウェア

OpenMP の共有メモリ環境における実行モデルを図２に示す．プログラム中の逐次処理部分に対して，マスタースレッドが実行される．プログラム実行がプラグマによって指定された並列リージョンに到達すると，マスタースレッドがスレーブスレッドを生成し，各スレッドが各プロセッサ上で並列に実行される．図２の実行モデルから動作合成によって生成する並列ハードウェアの構成を図３に示す．生成されるハードウェアは，図２の fork-join モデルにおいて，マスタースレッドを逐次処理ハードウェアに，並列リージョンの複数スレッドを並列ハードウェアとして合成する．生成ハードウェアは有限状態機械(FSM)とデータパス(DataPath)で構成され，各々の実行モデルに対応した処理を行う．まず逐次ハードウェアが実行され，処理が並列リージョンに到達すると，逐次ハードウェアが並列ハード

ウェアの動作を開始させる．並列ハードウェアの各ノードは同時に動作し，各ノードの動作が終了すると逐次ハードウェアに実行が戻る．共有データはメモリに格納され，各ノードが並列動作する場合はメモリアクセスを調停するアービタを介してアクセスされ，調停によって許可されたノード以外は停止する．OpenMP の SMP 環境での実行モデルと比べ，スレッド生成や引数のコストが必要ないため，処理の粒度が小さくても高い並列化効果が期待できる．

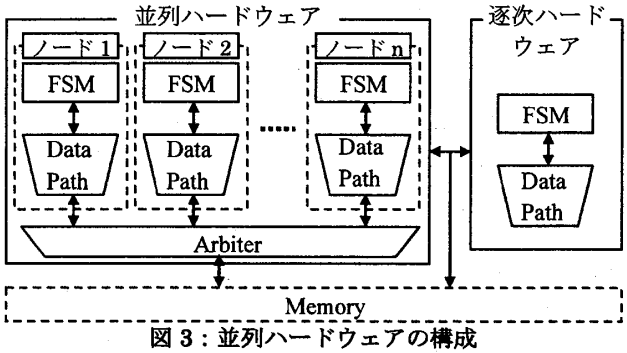


図３：並列ハードウェアの構成

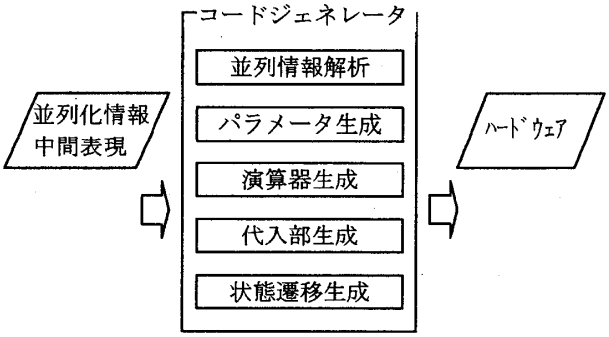


図４：コードジェネレータの構成

並列化において，データ並列を用いる場合は各ノードの DataPath はほとんど同じとなり，FSM はノードによって割り当てられるデータの範囲に合わせて若干異なっている．一方，タスク並列の場合は，各ノードのタスクの処理内容によって DataPath，及び FSM は各々大きく異なることになる．アルゴリズムの並列化を検討する際には，データ並列，タスク並列共に，各ノードの負荷均衡，及び同時に発行されるメモリアクセスを減少させることが性能向上を達成する上で重要である．

3. コードジェネレータの実装

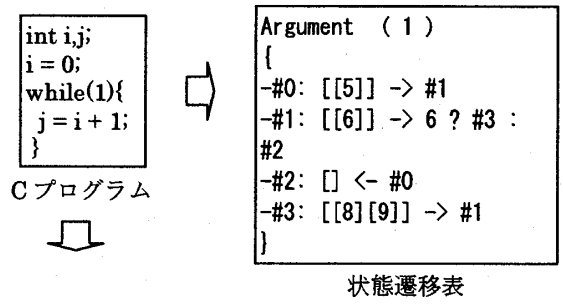
3.1 コードジェネレータの構成

コードジェネレータは，トランスレータによって生成された中間表現を用いてハードウェアの生成を行う．作成したコードジェネレータの構成を図４に示す．コードジェネレータは５つのモジュールから構成される．並列情報解析では，生成された中間表現と OpenMP によって指示された並列化情報を用いて，並列処理部の中間表現の複製や内部定数の変換を行う．パラメータ生成では，変換された中間表現を用いて，レジスタや定数を出力する．演算器生成では，用いられる各種演算器の生成とレジスタとの結線を行うコードを生成する．代入部生成では，演算された結果や，メモリなどの外部からの入出力をレジスタに代入するコードを生成する．状態遷移生成では，ハードウェアの FSM にあたる内部処理のコントロールを行うコードを出力する．

3.2 中間表現からのハードウェア生成方法

(1) 中間表現

トランスレータによって生成される中間表現の例を図５に示す．中間表現では処理を表すシンボルテーブルと制御情報を表す状態遷移表が出力され，両方を合わせてコントロールフローグラフを表す．シンボルテーブルは演算される変数や処理，代入先を示しており，状態遷移表によって次に遷移する状態が示される．シンボルテーブルの"//"は対応する演算を示す．本システムではポインタ，浮動小数点数などは使用できない．




```
0: Auto Obit : <function> main( )
1: Auto Obit : null
2: Auto Signed 32bit : i
3: Auto Signed 32bit : j
4: Auto Const 32bit : *4 := 0
5: Auto Signed 32bit : =( 2 4 ) //i=0
6: Auto Const 32bit : *6 := 1
7: Auto Const 32bit : *7 := 1
8: Auto Signed 32bit : +( 2 7 ) //i+1
9: Auto Signed 32bit : =( 3 8 ) //j=i+1
```

シンボルテーブル

図５：中間表現の例

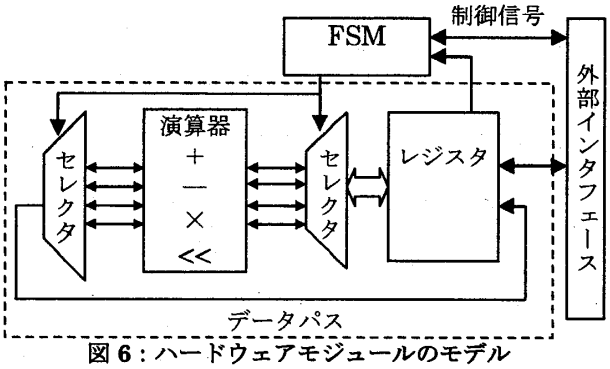


図６：ハードウェアモジュールのモデル

(2) ハードウェアモジュールの生成方法

動作合成を行う際に想定するハードウェアモジュールのモデルを図６に示す．動作合成では，状態遷移表で表現される演算に対して，演算の数だけ演算器を生成することも可能であるが，使用頻度の低い演算器が増加し，冗長な回路となる．また，同時実行できる演算器の性能と回路の冗長性のバランスを取ることは一般的に難しく，これはデータと処理のバインディングの問題となる．そこで，本研究

では１サイクル１演算と設定し，各モジュール単位で最低限の演算を持つ非常にシンプルなモデルを想定する．想定するデータパスでは，＋，－，×などの演算器に対して，変数を格納するレジスタを大きなセレクタを介して接続する．演算器からの出力は，セレクタを介して必要なレジスタへ書き戻す．FSM は状態遷移表，及びシンボルテーブルから生成し，内部状態のコントロールを行う．状態遷移に必要な変数についてはレジスタの値を参照する．また，FSM はセレクタへの制御信号，及びアービタや外部モジュールへの制御信号をコントロールする．本研究では，演算器とデータパスのビット幅を 32 ビットとしている．

4. 画像処理に対するハードウェア動作合成

4.1 対象のアルゴリズム

(1) エッジ検出

エッジ検出とは画像から対象物の外形を表す輪郭を抽出する処理である．明るさや色が急激に変化する部分を微分して強調することにより検出を行う．本研究では，１次微分において最もよく用いられる Sobel フィルタで実験を行っている．x 方向と y 方向の偏微分の定義式を式(1)，(2)に，変化の強さを式(3)に示す．偏微分の重み付けに用いる Sobel オペレータを図７に示す．

x 方向：$\Delta_x f(i,j)=\dfrac{f(i+1,j)-f(i-1,j)}{2}$ ...(1)

y 方向：$\Delta_y f(i,j)=\dfrac{f(i,j+1)-f(i,j-1)}{2}$ ...(2)

強さ：$\sqrt{\Delta x^2+\Delta y^2}$ ...(3)

$\Delta_x f$

-1	0	1
-2	0	2
-1	0	1

$\Delta_y f$

-1	-2	-1
0	0	0
1	2	1

図 7：Sobel オペレータ

対象画素に対し，近傍画素の x 方向，y方向に Sobel オペレータに従って演算を行う．Sobel フィルタではx方向，y 方向について，それぞれ 4 回の加減算と 2 回の乗算，強さの計算に 1 回の加算と 2 回の乗算が行われる．ルートの演算についてはハードウェアでは実装が難しいため，プログラムではビットシフトによる近似演算を用いて実装している．

(2) ハフ変換

ハフ変換とは，画像の中から，直線，円，任意図形などの抽出を行う際に用いる手法の一つであり，直線を検出するプログラムの実装を行った．直線を表す代数方程式を(4)に示す．原画像を走査し対象画素 p(x,y)を検出したときに，その座標(x,y)を式(4)の x,y に代入する．

$$\rho = x\cos\theta + y\sin\theta \quad ...(4)$$

ここで，ρ は座標原点から直線へ下ろした垂線の長さ，θ は垂線と x 軸との間の角度を示すパラメータである．コンピュータ上では連続量である θ の変化をすべて計算することはできないため，Δθ を設定し，θ ρ を有理数上の離散値として表現する．Δθ を小さくし，ρ 軸の刻みを小さくすることで計算精度は向上するが，計算負荷が増加する．また，計算されるべき特徴点が多ければ多いほど，写像の計算が増える．ソフトウェアでは，マクローリン展開を用いて cos 関数と sin 関数の計算を行うが，ハードウェアではマクローリン展開の実装が難しいため，テーブル演算によって実装している．

4.2 並列ハードウェア構成

エッジ検出とハフ変換のラスタスキャン部分に対してデータ並列を用いて並列化を行った．エッジ検出の場合に生成される並列ハードウェアの構成を図８に示す．

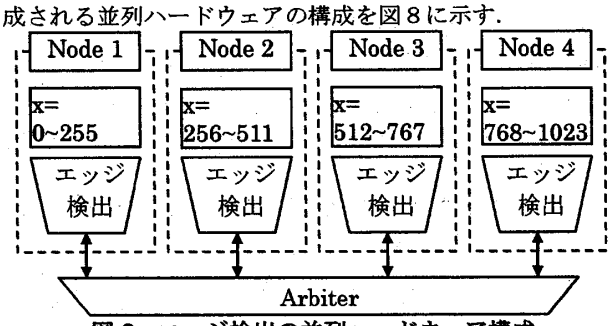


図 8：エッジ検出の並列ハードウェア構成

図８は画像サイズが 1024×1024 の画像に対し，x 方向に対してデータ並列を行った場合を示している．対象の画像空間を x 方向に等分割し，各ノードに処理を割り当てる．各並列ノードは逐次ハードウェアからのスタート信号によって動作を開始する．ハフ変換の画像空間は 256×256 であり，ハードウェアの構成は同じである．

5. 生成ハードウェアとシステムの評価

5.1 SMP と生成ハードウェアの実行時間

動作合成システムの実行環境を表 1に示す．

表 1：実験環境

SMP 環境	Quad Xeon 3.0GHz, 4GB RAM
OpenMP コンパイラ	Intel コンパイラ 9.1.038
論理合成ツール	Xilinx ISE 8.2i
PC 環境	Intel Core2 duo 2.66GHz, 4GB RAM
シミュレーションツール	ModelSim SE 5.8c

エッジ検出に用いた画像は，解像度 1024×1024，輝度 0~255 の pgm 画像である．また，ハフ変換に用いた画像は，解像度 256×256，輝度が 0,255 の 2 値画像，θ ρ 空間は 720×1440，輝度が 0~255 である．エッジ検出とハフ変換の SMP クラスタでの実行時間を，それぞれ表 2，表 3に示す．速度向上はエッジ検出の方が高い．これは，エッジ検出はすべての画素に対して規則的に同じ処理を行うのに対して，ハフ変換では画像の対象画素を発見した場合に処理を行うため，各ノードで負荷が均等にならなかったからと考えられる．

表 2：エッジ検出の SMP クラスタでの実行時間

スレッド数	1	2	4
実行時間(ms)	39.39	21.41	13.97
速度向上比	1.00	1.84	2.82

表 3：ハフ変換の SMP クラスタでの実行時間

スレッド数	1	2	4
実行時間(ms)	38.05	28.26	25.77
速度向上比	1.00	1.35	1.48

エッジ検出とハフ変換の生成ハードウェアの実行時間を表４，表５に示す．クロック減少比は，スレッドが一つである逐次処理の場合に対する動作クロック数の比率を示している．

表４：エッジ検出の生成ハードウェアの実行時間

	スレッド数	1	2	4
回路シミュレータ	動作クロック数(Mcycle)	158.7	79.4	39.8
	クロック減少比	1.00	2.00	3.99
並列ハードウェア	動作周波数(MHz)	88	88	88
	速度向上比	1.00	2.00	3.99

表５：ハフ変換の生成ハードウェアの実行時間

	スレッド数	1	2	4
回路シミュレータ	動作クロック数(Mcycle)	51.3	29.5	26.2
	クロック減少比	1.00	1.74	1.96
並列ハードウェア	動作周波数(MHz)	89	89	89
	速度向上比	1.00	1.74	1.96

動作周波数については，想定するハードウェアのデータパスがほぼ同じであるため，エッジ検出，ハフ変換共に並列数に関係なくほぼ同じであった．動作クロック数については，エッジ検出において理想的にクロック数が減少し，ほぼ理想的な速度向上を得ることができた．これは各ノードのメモリアクセス頻度に対し，演算に必要なクロック数の方が十分に多かったため，ストールの発生頻度が少なかったことが考えられる．また，エッジ検出はほとんど分岐のない規則的な処理であるため，各並列ノードにおいて，図９に示すように，メモリアクセスが生じるタイミングがストールによって一度ずれれば，次のストールが発生しにくかったことが考えられる．

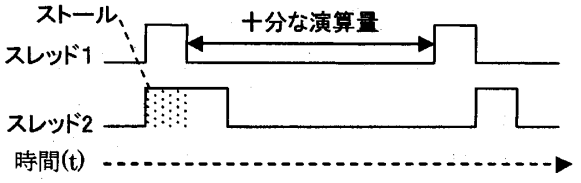


図９：メモリアクセス

ハフ変換のクロック減少比は４スレッドで1.96倍である．エッジ検出に比べ並列化効果が低くなっているのは，SMPクラスタでの実行と同様に，ハフ変換がデータ依存のアルゴリズムであるからと考えられる．エッジ検出とハフ変換の実験結果から，共に SMP 環境での速度向上比と動作合成時のスレッド数による速度向上比に同様な相関が見られたことは，並列アルゴリズムを検証する際に有効であると思われる．エッジ検出とハフ変換の両方において，速度向上比は SMP 環境よりも動作合成によって生成されたハードウェアの方が高い．これは，本研究で想定したハードウェアモデルでは，動作合成されるハードウェアの動作クロック数全体に占めるメモリアクセスが少ないため，アービタを通したメモリアクセスを効率的に処理できたからである．

5.2　回路規模と回路シミュレーション時間

エッジ検出とハフ変換の回路規模と回路シミュレーション時間を，それぞれ表６，表７に示す．

表６：エッジ検出の回路規模とシミュレーション時間

スレッド数	1	2	4
回路規模(Slices)	2295	6484	12745
回路面積比	1.00	2.83	5.55
シミュレーション時間(s)	3916	3912	3879

表７：ハフ変換の回路規模とシミュレーション時間

スレッド数	1	2	4
回路規模(Slices)	1627	4207	8232
回路面積比	1.00	2.59	5.06
シミュレーション時間(s)	977	1015	1021

回路規模については共に並列数以上に増加している．これは，並列化のオーバーヘッドにあたるアービタやレジスタ，配線などの回路の増加によるものと思われる．

回路シミュレーション時間は，SMP 環境と比較してエッジ検出で 10~30 万倍，ハフ変換で 2~7 万倍程度である．すなわち，SMP 環境を用いることで，高速に並列アルゴリズムやプログラムの検証を行うことができる．回路シミュレーション時間がほとんど変化しないのは，スレッド数の増加によって動作クロック数が減少するが，回路規模が増大するためと考えられる．

6.　おわりに

本研究では，OpenMP を用いたハードウェア動作合成システムのコードジェネレータを実装し，画像処理による評価を行った．トランスレータによって生成された中間表現に対して，ハードウェアモデルを想定し，中間表現からのハードウェア生成方法を示した．また，エッジ検出とハフ変換の OpenMP プログラムから本システムを用いて動作合成を行った．エッジ検出はメモリアクセス間に規則的で十分な演算量があるため，生成ハードウェアで理想的な速度向上が得られた．ハフ変換はデータに依存性があるため，速度向上が４スレッドで２倍程度となった．回路規模は，アービタやレジスタの増加のために，並列数よりも多くなった．

今後の課題として，タスク並列におけるパイプライン処理の実装，コードジェネレータにおける演算器やレジスタのバインディング，必要な動作クロック数とメモリアクセスのバランスの達成などがあげられる．

参考文献

[1] 井上,近藤,泉,福井:“C 言語からの高位合成を用いたハードウェア最適化に関する一検討”,情報処理学会研究報告,Vol.2005,No.102 pp.55-60,2005.

[2] 中谷，松崎，山崎:“OpenMP によるハードウェア動作合成システムの設計と検証”,FIT2007,C-006,2007.

[3] 荒本,冨山,村上:“動作合成の効率化を指向した動作レベル記述・トランスフォーメーション”，情報処理学会研究報告 Vol.2003 No.120 pp.67-72, 2003.