修士論文

OpenMP によるハードウェア動作合成システムの設計と検証

氏名	**：**	**中谷 嵩之**
学籍番号	**：**	**6124050162-6**
指導教官	**：**	**山崎 勝弘 教授**
提出日	**：**	**2007 年 2 月 16 日**

立命館大学大学院　理工学研究科　情報システム学専攻

内容梗概

本研究では，並列プログラミング言語である OpenMP を入力として並列動作ハードウェアを自動的に生成する新しい動作合成手法の提案と評価を行った．

提案する動作合成手法は PC クラスタやマルチプロセッサ PC など共有メモリ環境を用いたアルゴリズム検証・評価系と，OpenMP 記述からハードウェアを自動的に生成するハードウェア動作合成系で構成される．アルゴリズム検証・評価系では OpenMP に対応したコンパイラや解析ツールを用いて，アルゴリズムや並列化手法の評価と検討を行い，要求に応じて OpenMP 記述を改良する．ハードウェア動作合成系ではアルゴリズム検証・評価系から得られた OpenMP 記述を自動的にハードウェアに変換し，変換中に回路面積やサイクル数などの制約を満たすように様々な最適化を行う．

本論文ではハードウェア動作合成系を実際に設計・実装し，FIR フィルタとウェーブレット変換を対象としてアルゴリズムと並列化手法の評価を行い，その後，動作合成及び生成されたハードウェアに対し評価を行った．ウェーブレット変換の動作合成ハードウェアでは, 並列化手法の 1 つであるデータ分割において, 並列実行ノードが 2 ノードで 1.99 倍, 4 ノードでは 3.99 倍という理想的な速度向上が得られた．またアルゴリズム検証・評価系での評価結果と比較することで，ソフトウェアとハードウェアにおける OpenMP 実行モデルのふるまいについて評価した．

本論文では提案した動作合成手法を実現するシステムを実装し，OpenMP を用いた動作合成手法の特徴と性能評価の結果を示すことにより，提案手法の有用性を示す．

目次

図目次

表目次

1. はじめに

近年，大規模計算機からパーソナルコンピュータ，安価な玩具に至るまで様々な電子機器に LSI が搭載されるようになり，LSI は電子機器において新しい機能やサービスを実現する最も重要な要素となっている．日々高まるユーザの要求を実現するため，電子機器に搭載される LSI にはさらに高い処理性能，多彩な機能，高い信頼性，低い消費電力などが要求されており，LSI の回路規模や複雑さは著しく増加している．しかし，多様な製品に LSI を供給する多品種少量生産の現代においては製品の開発サイクルが短縮され, 短期間で高性能な LSI を設計する必要があり，設計規模の増大に設計能力が追いつかないという状況が生じ，設計生産性の危機が問題となっている．

設計生産性の向上が可能な技術として，LSI の回路の動作を C 言語などのプログラミング言語を用いてより抽象的に記述し，LSI の回路構造を動作の記述から自動合成する動作(高位)合成技術が多数提案されている[1-6]. 動作合成では回路記述を抽象化することで回路設計が容易に行えることに加え，最適化により抽象的な記述から ASIC や FPGA など実装環境に適した回路を生成することで性能のさらなる向上が見込める．また HDL などを用いた RTL(Register Transfer Level)の回路検証と比べ, C 言語などのプログラミング言語を用いた検証では, 同一の機能の検証速度が 1 万～100 万倍以上も高速であり，検証期間の短縮が可能である[7]．このような多くの利点から動作合成技術は商用ツールとしても多数販売され，実際の製品開発に適用され始めている[8,9].

実際に高性能な回路を実現するためには，処理の並列化が重要な要素となる．しかし現在の動作合成技術では C 言語など既存の逐次処理を実行モデルとして扱う．このような言語を入力とした動作合成技術では，問題に対する並列化手法の有効性の推定や，設計者の意図した並列動作回路を自動で生成することは難しい. 実際の高位合成技術において, Spec C や Handel C などの言語では，並列化の制御を RTL での動作を考慮して設計者が記述する必要がある．またその他の多くの高位合成系では最適化による並列化では，演算レベルの最適化が主であり，大規模な並列化は設計者が責任を持って記述しなければならない.

そのため主な並列化部位の選定や並列動作回路の設計は，熟練した設計者による手動での並列化に依然として頼っており，動作合成手法を導入しているにも関わらず，設計者の負担が非常に大きくなっている．また並列化したハードウェアの検証においては主に RTL のシミュレータなどを用いるため，機能及び性能の検証が設計後期になりコストが増大するという問題もある.

本研究では以上のような問題を解決するため，動作レベル記述に並列プログラミング言語 OpenMP を用い，設計者の意図した回路を構成可能な動作合成手法の提案と設計を行うことを目的とする.

OpenMP は既存の逐次処理プログラムに対し，短い指示文を追加するだけで並列処理プログラムに拡張することが可能である．ベースとなる言語としては Fortran と C/C++をサポートしている[10]．また，通信や信号で並列動作を記述するのではなく，抽象度の高い処

理自体に対して並列動作の振る舞いや構造を指定するため，MPI や PVM など既存の並列プログラミング言語と比べて記述が容易という特徴を持つ．さらに指示文は OpenMP をサポートしないコンパイラはコメントとして扱うので，逐次プログラムと並列プログラムで同一のソースコードを用いることができ，管理や移植が容易である．

OpenMP は 1997 年に Fortran 向けに最初の規格が発表され，以降多くのハードウェアおよびソフトウェア・ベンダが参加する非営利団体(Open MP Architecture Review Board)によって管理と仕様の公開がされている．主に大規模計算機などで用いられ，主に科学技術計算を目的とした大規模計算機のコンパイラなどで採用されていたが，マルチコアプロセッサや PC クラスタなどに代表される PC ベースの SMP(Symmetric Multiple Processor)環境が普及し，Microsoft Visual Studio や Intel C/C++ Compiler など様々なコンパイラでサポートされている[11-13]．OpenMP が策定される以前は，並列プログラミングの記述法や指示文は並列化コンパイラに対して並列動作情報を与えるものであり，コンパイラが対象とするシステムに依存していたが，OpenMP は多くのベンダーから提供される計算機とコンパイラでサポートされることが想定されており，動作を仕様として規定している．

OpenMP の並列動作を容易に記述が可能で抽象度が高いという利点により，本研究で提案する動作合成システムでは OpenMP を並列動作回路の動作記述に用いる．並列プログラミング言語をハードウェアの設計に用いることで並列動作の記述や分析，性能の検証を容易にし，ハードウェアの動作合成における設計者の負担を軽減することが可能である．

本研究では動作合成手法の有効性の検証のため，OpenMP 動作記述の入力から VerilogHDL を出力する動作合成システムを設計し，ツールとして実装した．動作合成の対象となる並列化手法として，一般的な並列化手法であるデータ並列とタスク並列を主な対象としている．さらに，システムの性能評価のために画像処理や音声処理で使用されるウェーブレット変換と，信号処理でよく用いられる FIR フィルタを対象とし，データ分割とタスク分割による並列動作ハードウェアを動作合成システムで生成し，ハードウェアの性能評価を行った．その結果から動作合成システムの性能の評価と OpenMP を入力とする動作合成手法の有効性について考察を行う．

本論文では 2 章において提案する動作合成システムの概要と，その入力となる OpenMP の詳細と実行モデルについて述べる．第 3 章では OpenMP の実行モデルに対応して動作合成システムが実際に出力するハードウェアの構成について，第 4 章では動作合成システムの実装環境について述べる．第 5 章では実際のアプリケーションとしてウェーブレット変換と FIR フィルタを対象として動作合成システムを用い，その性能評価を行った．

2. OpenMP を用いたハードウェア動作合成システム

2.1 OpenMP の概要と fork-join 実行モデル

OpenMP は SMP 環境でのプログラミングを目的とした API(Application Programming Interface)であり，ベースとなる言語を指示文を用いて並列プログラム化ができるように拡張した言語である．

SMP 環境ではすべての並列処理ノードが同じ処理装置を持ち，メモリ空間を共有する．そのため SMP 環境では通信など並列処理における難度が高い制御が必要ない．

OpenMP の指示文は既存の並列化コンパイラの指示文のようにコンパイラに対して自動並列化を補助する情報を提供するのではなく，仕様により規定された並列実行モデルへの API を提供している．そのため仕様に基づいた記述を行うことで，異なる環境下でも実行可能で同一の結果が得られる，移植性の高いプログラムを作成可能である．

OpenMP は設計者にとって以下のような特徴を持つ．

- 既存の逐次プログラムに段階的に指示文を挿入することで，指示文によって指定した範囲の段階的なプログラムの並列化が容易に可能である．
- 自動で並列化が実行されず，並列部位の抽出や依存性の分析・解決は設計者が行う．
- 共有変数への同時アクセスや各処理の同期制御は設計者が意図しなくて良い．
- 自動並列化による抽出が不可能な大きな処理の並列化に適している．
- 共有メモリモデルであるためデータのノードへのマッピングが不要である．

OpenMP プログラムでは並列実行する領域の構造を設計者が OpenMP 指示文によって指定し，その範囲がコンパイラによってマルチスレッドプログラムに変換されるため，様々な SMP 環境で実行可能である．

OpenMP の並列処理の実行モデルは設計者にとって理解が容易な fork-join モデルである．fork-join モデルを図 1 に示す．fork-join モデルでは逐次実行領域を実行するスレッドはマスタースレッドと呼ばれ，プログラムの実行を制御する．プログラムが指定された並列実行領域の開始点に到達すると，マスタースレッドがスレーブスレッドを生成し，並列処理を実行する．このスレーブスレッドの生成と並列処理の開始を fork と呼ぶ．並列処理後，全てのスレッドの終了を待つ同期処理や各スレッドの結果の統合を行い，全ての処理が終了後，スレーブスレッドが終了し，マスタースレッドのみ逐次処理へ復帰して処理を継続する．この終了の同期処理を join と呼ぶ．また，fork-join 間の並列処理区間を並列リージョンと呼ぶ．

OpenMP ではマスタースレッドによる逐次処理と fork-join モデルの並列処理を繰り返すことで並列処理を行う．OpenMP は実行モデルは規定されているが，その実装は実行モデルを守る限り自由である．そのため同一のプログラムを用いた場合でも，スレッドの実装を行うスレッドライブラリやオペレーティングシステムによって処理性能は異なる．

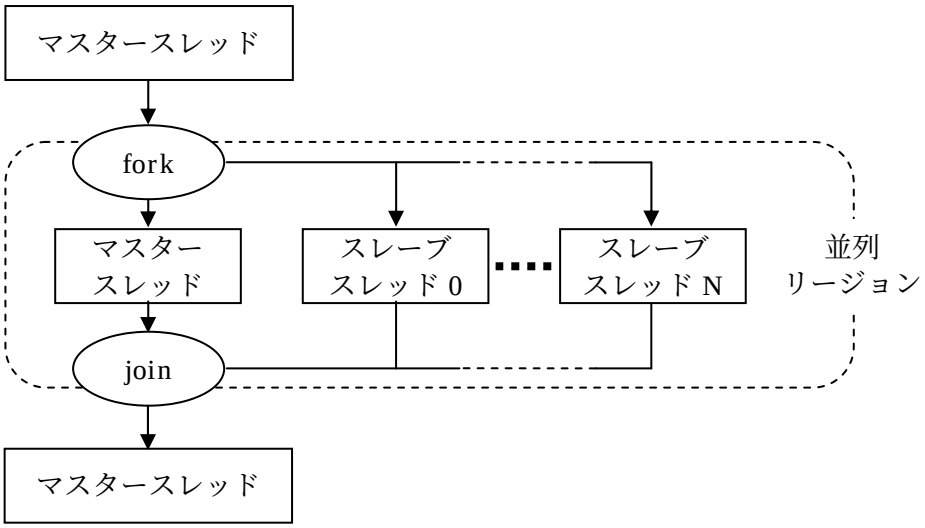

図 1　OpenMP の fork-join モデル

以下に OpenMP で扱う代表的な並列化手法である，データ分割とタスク分割の実行モデルを示す.

(1) データ分割による並列処理

データ分割の実行モデルでは，処理対象の大規模なデータを分割し，複数の並列実行ノードが分担して処理する．主にループの並列化に用いられる.

データ分割の例として，1 から 100 までの数の自乗を求めるプログラムの OpenMP による記述例を図 2 に示す．図 2 では 100 回の繰り返し処理を分割した．繰り返し処理の並列化は“ #pragma omp parallel for ”で指定し，並列実行ノードの数は環境変数で静的に指定するか，ライブラリ関数で動的に指定する.

データ分割による並列化では各並列実行ノードが分担するデータに偏りがあると，最も処理の遅いノードがボトルネックとなり並列化効果が低くなるため，ノードに対する負荷分散が非常に重要である．また，各並列処理の順序は不定であるため，設計者は並列処理の対象のデータに前後のループ処理の結果との依存性を解消する必要がある．図 2 に示した問題の場合，1 から 100 までの自乗はそれぞれ独立して計算できるため，どの数から計算を行っても結果は同じとなり，演算の順番に依存性はない．また，自乗計算では単に乗算を行うだけのため，分割したノードの負荷は全て同じであり，特定のノードに負荷が集中し，ボトルネックとなることはない．自乗ではなく平方根を求める場合，求める数値によって処理の負荷が異なるため，ボトルネックが発生する場合がある.

この問題を 4 つのノードで処理を分担した場合の実行モデルを図 3 に示す．マススレッドが OpenMP 指定文の位置へ到達すると，スレッド生成を行い，100 回の繰り返し処理を分割して各ノードに 25 回ずつ割り当て，並列処理を開始する．すべてのノードの処理を終了すると，マススレッド以外のスレッドを終了する.

```
#pragma omp parallel for
for(i=1;i<=100;i++) { //変数 i が 1 から 100 までのループ処理
   squaring[i] = i * i; //i の自乗を計算
}
```

図 2　OpenMP によるデータ分割手法の記述例

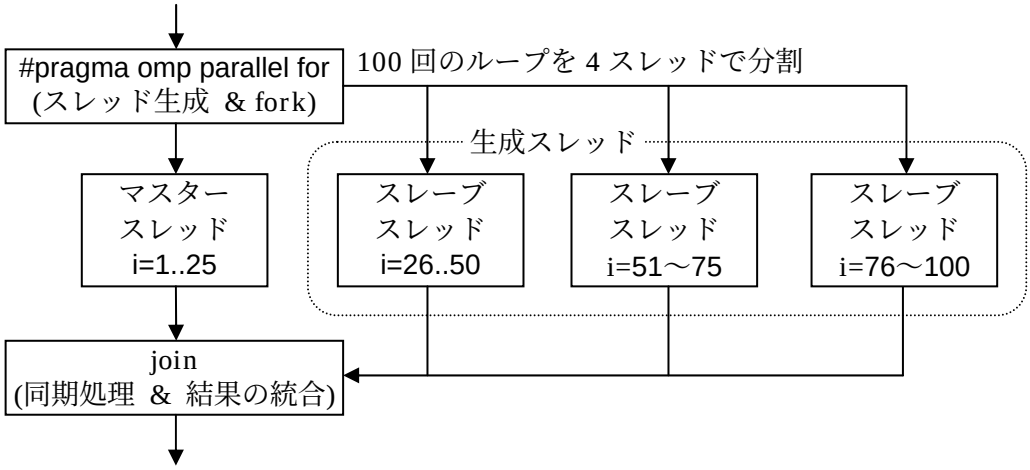

図 3　OpenMP によるデータ分割手法の実行モデル

(2)　タスク分割による並列処理

タスク分割の実行モデルでは，並列に実行できる複数の処理を実行ノードが分担して処理する．主に逐次処理の中に依存性がなく並列に実行できる処理が複数ある場合に用いる．

タスク分割の例として, 関数 A, 関数 B, 関数 C が独立して実行可能である場合の OpenMP による記述例を図 4 に示す．図 4 では並列処理の開始点を“ #pragma omp parallel sections ”で指定し，各並列処理の範囲を“ #pragma omp section ”で指定している．並列実行ノードの数は環境変数で静的に指定するか，ライブラリ関数で動的に指定する．

タスク分割の実行モデルを図 5 に示す．

```
#pragma omp parallel sections
{
   #pragma omp section
   { A() } //並列処理 1
   #pragma omp section
   { B() } //並列処理 2
   #pragma omp section
   { C()} //並列処理 3
}
```

図 4　OpenMP によるタスク分割手法の記述例

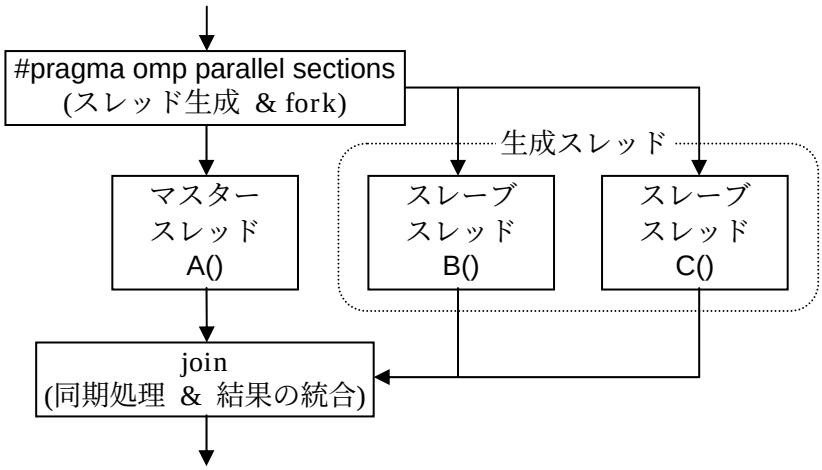

図 5　OpenMP によるタスク分割手法の実行モデル

タスク分割では，データ分割と同じように分割した処理の大きさに偏りがあると最も遅いノードがボトルネックとなる．処理の内容を理解し，全てのノードが同程度の処理負荷になるように逐次処理の分割を行わなければ性能向上を得ることが難しい．

2.2 ハードウェア動作合成システムの構成

本研究で提案するハードウェア動作合成システムは見積もりや検証を行うアルゴリズム検証・評価系と実際にハードウェアを出力するハードウェア合成系で構成される．
ハードウェア動作合成システム全体の構成を図 6 に示す．図 6 の右側であるアルゴリズム検証・評価系では，OpenMP コンパイラの出力したプログラムに対し，SMP 環境での実行やツールを用いて，OpenMP 記述のデバッグや性能評価を行い，結果を基にアルゴリズムや並列化手法の改善を行うことが目的である．図 6 の左側のハードウェア動作合成系では，OpenMP 記述を中間表現へ変換し，最適化を行い，並列動作ハードウェアの HDL 記述を出力することを目的とする．

実際の設計手順では，最初にアルゴリズム検証・評価系においてアルゴリズムや並列化手法の十分な検討を行い，その後，同じ OpenMP 動作記述をハードウェア動作合成系に入力して並列動作ハードウェアを得る．

2.3 アルゴリズムの検証・評価

アルゴリズム検証・評価系では既存の OpenMP に対するツールを利用し，並列アルゴリズムの検証及び評価を行う．その検証結果を基に動作記述を改善すると共に，設計初期において並列化手法の評価を行い，検証のコストを削減することを目的とする．入力である OpenMP プログラムによる動作記述は，対象のアプリケーションの逐次プログラムに対して OpenMP の指示文を追加することにより，プログラムを並列化したものである．動作記述は OpenMP コンパイラによってマルチスレッドプログラムへ変換する．

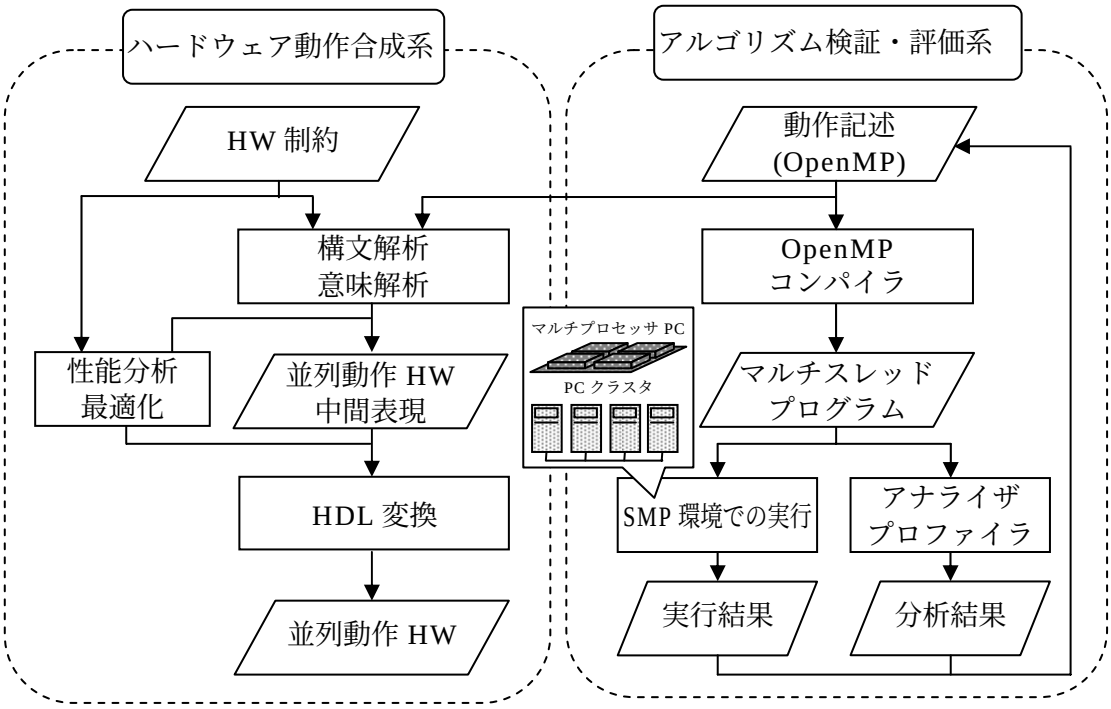

図 6　OpenMP を用いた動作合成システムの構成

並列化は指示文の追加だけで行うことができるので，並列動作回路や並列動作の制御に熟知した設計者でなくても並列化が容易である．またマルチスレッドプログラムに変換することにより，大規模計算機や PC クラスタ，マルチプロセッサ PC 上などの SMP 環境での高速なシミュレーションを実行することができる．これにより，並列動作の検証にかかっていた時間を大幅に短縮することができる．

さらに，SMP 環境に対応したプロファイラや，OpenMP プログラムに対するアナライザを用いることで，各ノードの負荷や共有変数へのアクセス頻度，メモリ使用量など非常に詳細な分析結果を得ることが可能である．これにより並列処理に必要なメモリ帯域や各処理ノードで十分な負荷分散が行われているかを入念に検討できる．

これらの実行結果や分析結果を基に OpenMP プログラムの記述の修正やアルゴリズムの改良を行うことで，回路設計前の設計初期において，並列処理アルゴリズムの検討や改良が可能となる．これまでも並列処理を並列プログラムで検証することは可能であったが，検証に使用した並列プログラムを動作合成することができなかったため，設計に加えてアルゴリズムの検討に追加のコストが必要となり，コストの削減が難しかった．

しかし，本研究で提案する新しい動作合成システムでは検証時に設計した OpenMP プログラムをそのまま動作合成する．アルゴリズムの検証が完了すると同時に，実際の並列動作回路が生成可能となるため，設計コストの大幅な削減が可能である．

2.4 ハードウェア動作合成

ハードウェア合成系では OpenMP による動作記述と設計者が与えるハードウェア制約から，並列動作回路の HDL 記述を得る． ハードウェア動作合成系の構成とその中で用いられる中間表現を図 7 に示す．

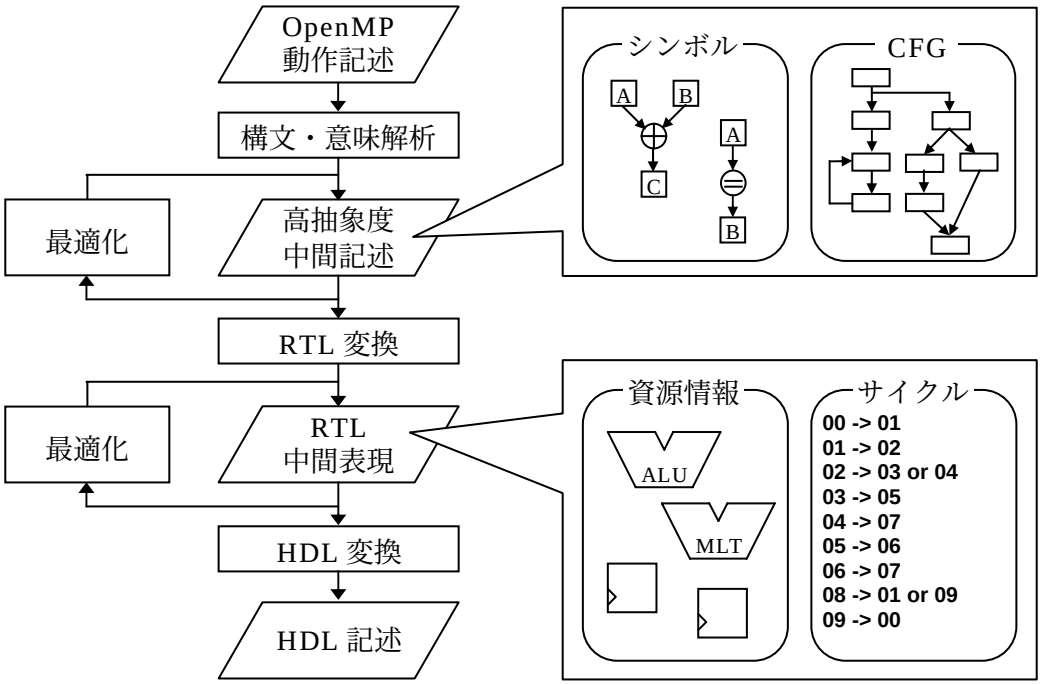

図 7　ハードウェア動作合成系の中間表現

入力である OpenMP プログラムは構文解析，及び意味解析によって内部に含まれる関数毎に中間表現に変換される．中間表現は高抽象度の中間表現と RTL の中間表現がある．高抽象度の中間表現は演算をシンボルで表現し，制御情報は CFG によるグラフ表現であり，通常のコンパイラで用いられる最適化や OpenMP の指示文から得られる並列化情報を用いた最適化を行う．高抽象度の中間表現における CFG の制御構造を図 8 に示す．RTL の中間表現はレジスタやメモリ，加算器といった資源情報とクロックサイクルレベルでの時間情報を持ち，最適化では入力として与えられた制約に従って必要な回路規模やサイクル数を最適化する．

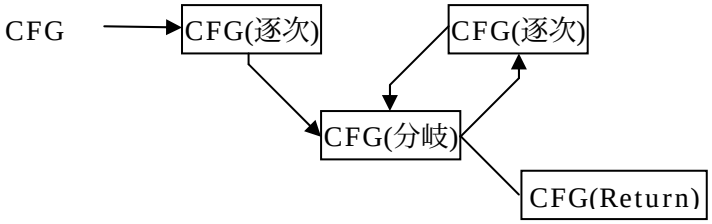

図 8　CFG による動作情報

最適化後の中間表現を HDL に変換し，並列動作回路の HDL 記述を出力する．

動作合成で出力されたハードウェアは，処理の制御を行う FSM と，データの保持と演算を行うデータパス，共有変数やグローバル変数へアクセスするためのメモリーインターフェイスで構成される．一般的な動作合成ハードウェアの構成を図 9 に示す．

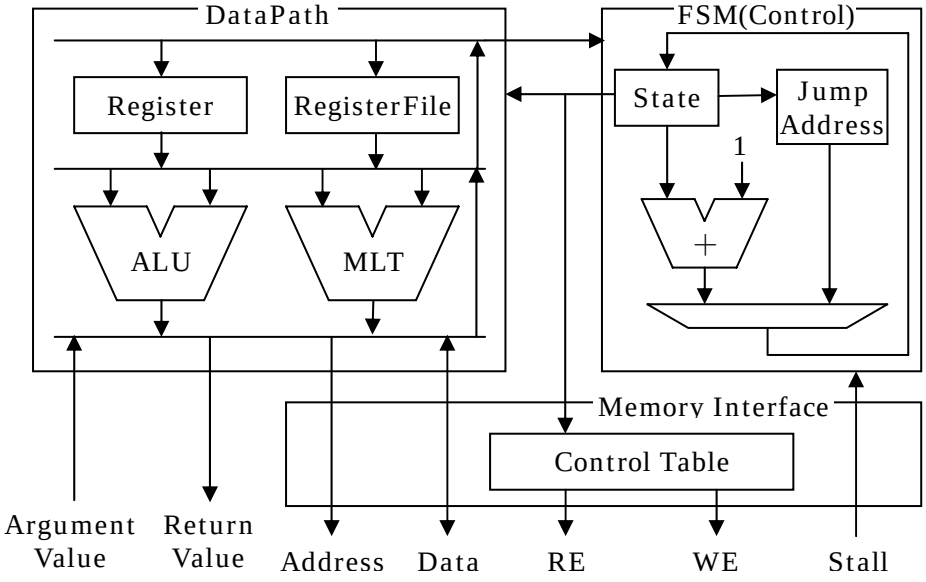

図 9　動作合成システムにより出力されるハードウェアの構成

FSM には内部の状態を保持する State レジスタと加算器，分岐時の状態の遷移先を保持する Jump Address メモリで構成される．FSM は通常は State を加算することで状態を遷移するが，分岐時はデータパスからの情報によって Jump Address に格納された遷移先が State へ入力される．データパスは関数内部のローカル変数を保持するためのレジスタや配列保持のためのレジスタファイル，演算に使用する加減算器や ALU，乗算器などで構成される．最適化時に動作記述と同時に入力された回路規模や必要サイクル数などの制約を満たすように演算器やレジスタの数が変化する．メモリーインターフェイスはハードウェアがグローバル変数や並列動作時に他のハードウェアと共有するようなデータへアクセスするために用いる．メモリアクセスが失敗した場合は Stall が入力され，回路全体の動作が停止する．

OpenMP を入力とした場合，ハードウェアには図 9 の構成に加え，並列処理用のデータパスが含まれる．並列処理用のデータパスには，並列処理用のノードである並列動作ハードウェアが複数含まれ，それらはアービタによって接続される．それぞれのノードは同時に動作するため，複数のハードウェアが同時に一つの変数やメモリにアクセスする場合がある．そのため，他のハードウェアと同時にアクセスする可能性がある変数については外部のメモリに格納し，メモリには常にアービタを通してアクセスするようにする．ある変数を外部のメモリへ格納すべきか，各並列動作ハードウェアの内部に格納すべきかは，

OpenMP の記述文により指定可能である．アービタは同時に複数のハードウェアが共有メモリへアクセスする場合は，一定の優先度に従って調停し，アクセスが許可されなかったハードウェアには Stall を出力する．

本研究では OpenMP を用いた動作合成手法の実験及び評価のため, OpenMP 動作記述から並列動作ハードウェアを自動的に生成する動作合成系を設計・実装した．これにより，アルゴリズム検証・評価系で十分な性能が得られると確認できた動作記述を入力とすることで，高性能な並列動作回路の自動設計が可能になる．

3. 動作合成システムによる並列ハードウェア構成

3.1 データ分割並列処理のハードウェア構成

データ分割は複数のノードが大規模なデータに対し，同一の処理を行う並列化手法であり，その実行モデルは図 3 のようになる．このモデルを実現するため，並列処理を含む関数のハードウェアは逐次処理を処理するデータパスと並列処理を行うデータパスによって構成される．並列処理を行うデータパスはスレッドのかわりに同じ処理を行う複数のハードウェアで構成され，各ハードウェアは共有変数へのアクセスを行う場合，アービタを通してアクセスする．また，並列処理を行う並列動作ハードウェアの起動や終了を制御するための制御部と，並列動作における引数を制御する引数制御部も含む．データ分割による並列処理を実現する動作合成ハードウェアの構成を図 10 に示す．

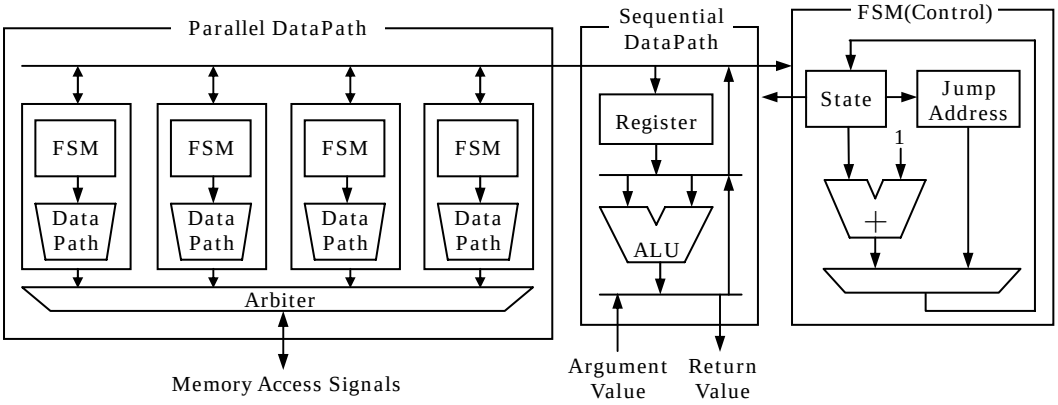

図 10　データ分割手法のハードウェア構成

並列処理を行うハードウェアは逐次処理を行うハードウェアと同じように制御部である FSM と演算を行うデータパスによって構成される．データ分割の並列処理では同じ処理を分担して行うため，すべての並列動作ハードウェアの内部構成も同一である．

関数は逐次処理から開始され，逐次処理を制御する FSM と逐次処理部分のデータパスにが処理を行う．FSM の状態が fork 処理に遷移すると，データパスから各並列動作ハードウェアに必要な変数を入力する．処理に必要な値を全て渡した後，並列処理ハードウェアを起動する．いずれかの並列動作ハードウェアが処理を行っている間，逐次処理の FSM とデータパスは停止し，同期を取るため全ての並列処理ハードウェアが処理を終了するまで待機する．全ての並列動作ハードウェアが処理を終了すると，必要に応じて結果のリダクション演算などを行った後，逐次処理の FSM が次の状態へ遷移し，逐次処理を継続する．

3.2 タスク分割並列処理のハードウェア構成

タスク分割では複数の異なる処理を複数のノードで分担する実行モデルであり，その実行モデルは図 5 のようになる．異なる処理を並列に実行するため，処理毎に異なるハードウェアを生成し，それらを複数並べて並列処理ノードを担当するハードウェアを構成する．

タスク分割による並列処理を実現する動作合成ハードウェアの構成を図 11 に示す. 各ハードウェアはデータ分割と同様に，共有変数へのアクセスを行う場合はアービタを通してアクセスし，並列処理を行う並列動作ハードウェアの起動や終了を制御するための制御部と，並列動作における引数を制御する引数制御部を含む.

タスク分割では複数の異なった処理を並列に実行するため，並列動作ハードウェアに含まれる各処理を担当するハードウェアは, 処理毎に異なる FSM とデータパスで構成される. タスク分割においても，並列動作ハードウェアはタスク分割と同様の手順で起動される.

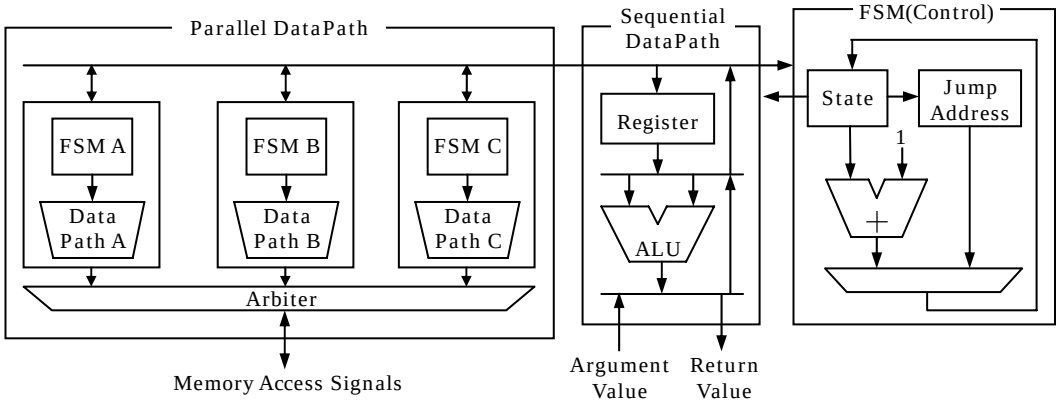

図 11　タスク分割手法のハードウェア構成

3.3 タスク分割のハイプライン化

ハードウェアではパイプラインという並列処理が性能向上において重要な要素となり，現在ではプロセッサや信号処理などの分野で高性能な回路を生成するために必須の技術となっている．パイプライン処理では複数の処理を繰り返し行う場合において，処理の開始時間を少しずつずらして同時並行に処理を実行する．処理 A と処理 B を繰り返し実行する場合，それをパイプライン化した場合の処理の流れを図 12 と図 13 に示す.

処理 A に必要な時間を 3，処理 B に必要な時間を 2 とする．パイプライン処理を行わない場合，各データにつき処理 A と処理 B を順次行うため，その結果，N データに対する処理に必要な時間は 5N となる．しかし，パイプライン処理の場合，処理 A と処理 B の間に結果を保存しておくバッファを加えることで，処理 A の処理終了時に処理 B を開始すると共に，同時に処理 A を再び開始することができる．処理 A の結果はバッファに保存されているので，処理 B の入力が処理の途中で更新されることはなく，正しい結果が得られる.

ハードウェア動作合成系の最適化において，タスク分割による並列化を用いており，特定の制御構造であれば，処理のパイプライン化を行うことが可能である．パイプライン化を行うためには, 処理 A と処理 B がタスク分割を行い, 処理 A の処理結果を処理 B が入力とし，処理 A，B にそれ以外に共有する変数がない場合である．この場合，処理 A と処理 B の間に処理結果を格納するレジスタやメモリを挿入し，処理 A と処理 B で同時に処理を開始することで，パイプライン処理が可能となる．処理 B の処理結果を次の処理の入力と

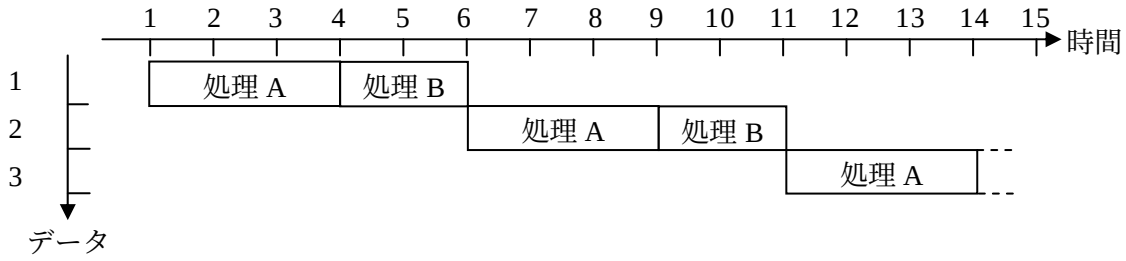

図 12　逐次処理における処理の流れ

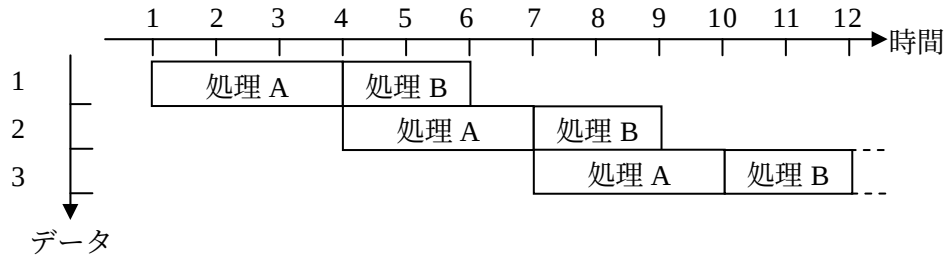

図 13　パイプライン処理における処理の流れ

する場合，その次の処理も含めたパイプライン化が可能であり，上記の条件が成立する限り，パイプラインの段数を増やし，処理の並列性を高めることが可能である．

また，パイプライン処理では処理 A と処理 B にデータ依存があっても並列処理が可能であるため，より多くの処理に対して並列化が可能である．しかし，タスク分割を用いるため最も遅い処理に全ての処理が同期しなければならず，パイプライン化を行う処理の負荷に違いがあると，負荷の大きな処理がボトルネックとなる．図 13 では処理 A と処理 B は同時に処理を開始するが，処理 B は処理 A が終了するのを待ち，同期を行わなければならない．

また，OpenMP のタスク分割における実行モデルにより，工夫が必要である．単純に処理 A と処理 B の間にレジスタやメモリを挟んだ場合，OpenMP では並列に実行される処理の順序までは規定されていないため，処理 B が処理 A の結果を読み出す前に，処理 A が新しいデータを処理し，結果を書き込む可能性がある．この場合，結果のデータが不定になる．

このような問題の発生を防ぐため，本研究におけるハードウェア動作合成システムでは，バッファを 2 重化する．OpenMP の実行モデルにおけるハードウェアによるパイプライン処理の動作を図 14 に示す. n 番目のデータの結果を処理 A が片方のバッファに書き込んだ場合，次の並列処理において処理 A は結果をもう片方のバッファへ書き込む．同時に処理 B は前の並列処理において処理 A が書き込んだバッファから結果を読み出し，処理を実行する．読み出しと書き込みを 2 つのバッファで交互に行うことにより，どのようなタイミングで並列処理の実行が終了しても正しいデータが出力される．

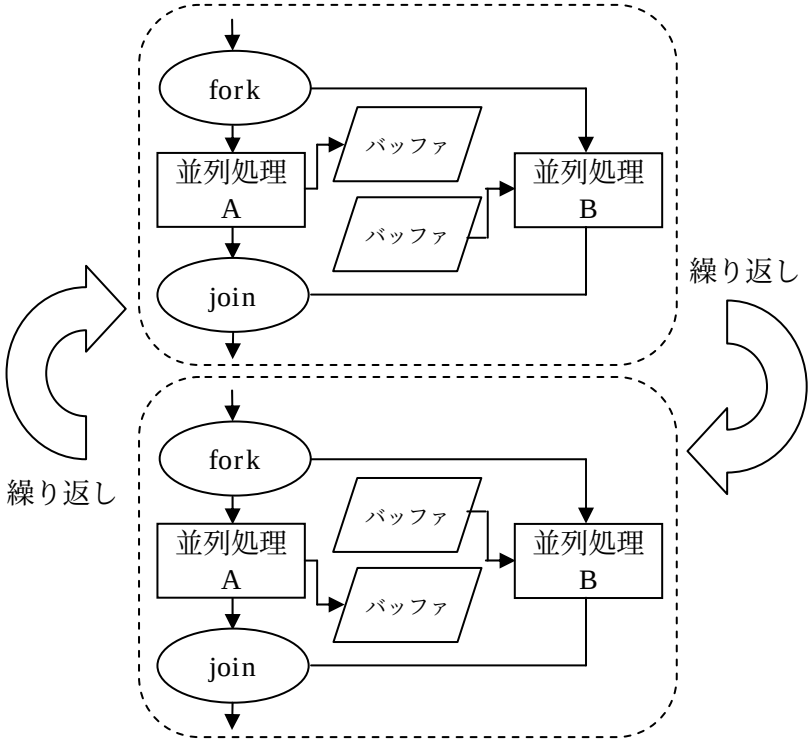

図 14　OpenMP の実行モデルにおけるパイプライン処理の動作

最適化によってパイプライン化を行ったタスク分割手法のハードウェア構成を図 15 に示す．並列処理を行うハードウェアの間にレジスタでバッファを構成し，パイプライン・バッファとする．パイプライン・バッファに対するアクセスはアービタによる調停が必要ないため，高速な処理が可能である．

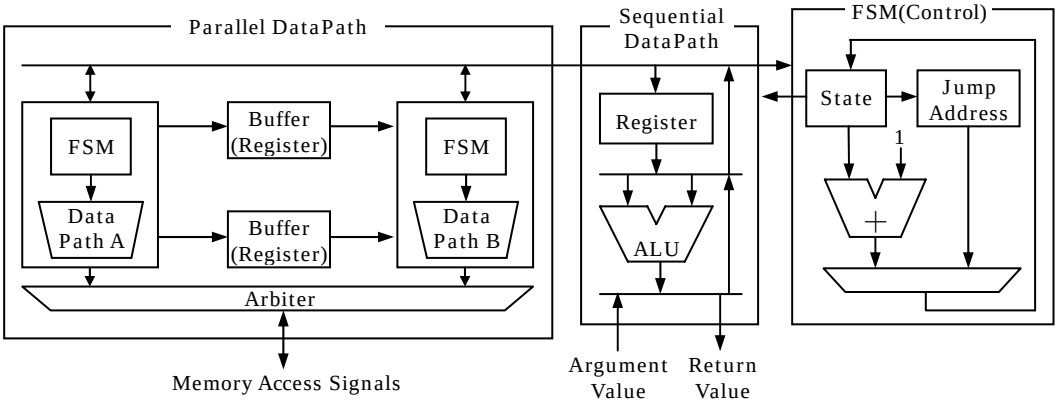

図 15　パイプライン処理のハードウェア構成

4. 動作合成システムの実装

4.1 構文・意味解析

ハードウェア動作合成系において，入力となる OpenMP の動作記述は最初に構文・意味解析によって中間表現へと変換される. OpenMP の動作記述は OpenMP の指示文とベースとなる言語を含むが，システムの実装においては制限がある.

OpenMP の指示文には以下のような制限を持つ.

(1) 並列化指示文は結合されたワークシェアリング(for，sections)文のみサポートする.
(2) マスター構文と同期構文をサポートしない.
(3) データ構文ではリダクション演算をサポートしない.
(4) ランタイムライブラリはサポートしない.

これらはハードウェアでの制御が困難であることから制限される.

ベースとなる言語は C 言語であり，以下のような制限を持つ.

(1) 配列はサポートするが，ポインタはサポートしない.
(2) ラベル文および goto 文はサポートしない.
(3) switch 文，do-while 文はサポートしない.
(4) 浮動小数点を用いることはできない.
(5) 構造体・共用体をサポートしない.

ポインタは任意のデータにアクセスするための仕組みであり，これをサポートするとハードウェア内部の変数などにアクセスする仕組みが必要になるため，並列化による性能向上が得られにくく，実現が難しい．ラベル文および goto 文も同様に任意の状態へ遷移するための仕組みであり，これはハードウェアでは実現がほぼ不可能である．ラベル文が使用できないため，switch 文は実装不可能であり，また do-while 文は while 文で代替可能であるためサポートしない.

浮動小数点および構造体・共用体はハードウェアでの実現は可能であるが，データパスや制御が複雑になる．浮動小数点の処理はハードウェアにおいては固定小数点に置き換えることが多いため，サポートしない．構造体と共用体については今後の実装でサポートする予定である.

動作記述において制限があるものは，主にハードウェアでの実現が困難か不可能な実装である．また，並列化において効果が得られにくい場合もサポートしなかった.

構文・意味解析は Java で実装し，構文解析の実装にはコンパイラ・コンパイラである JavaCC を用いた[21]．構文・意味解析の実装に必要なコードは約 2000 行であり，これは JavaCC による自動出力を含まない.

4.2 中間表現

中間表現は，抽象度の高く，一般的なコンパイラによる最適化と同様の最適化に適した高抽象度の中間表現と，資源やサイクルといった概念を含む RTL 中間表現の 2 つの中間表現を用いる．演算や制御などに対する最適化と，演算器や必要サイクル数などに対する最適化をそれぞれに適した中間表現で行うことを目的とする．

(1) 高抽象度中間表現

高抽象度の中間表現は演算や制御構造を示すための複数のクラスを含む．高抽象度中間表現に含まれるクラスの構造を図 16 に示す．

高抽象度の中間表現である内部表現クラスは複数の関数情報クラスと変数クラスを含む．内部表現クラスに含まれる関数情報クラスはソースコード中に含まれる全ての関数の情報であり，変数クラスは関数に含まれないグローバル変数の情報を保持する．

関数情報クラスは関数の制御に必要な情報と内部に含まれる変数の情報を保持する．関数の引数，返り値及び内部で用いられるローカル変数を変数クラスとして，内部で行われる処理の情報を CFG クラスとして保持している．

CFG クラスは C 言語における処理の流れを表すノードのクラスであり，分岐や逐次，Return，fork，join というノード自体の動作種類の情報を内部に含む．また，CFG クラスは他の CFG クラスが組み合わされて分岐やループなどの複雑な動作を示すため，CFG クラスに接続される CFG クラスの情報も保持している．また，CFG クラスは順次処理する式クラスを複数保持しており，制御構造の中で行われる処理の情報を示す．

式クラスと変数クラスはシンボルクラスの子クラスであり，変数や式など値を返すものを保持するクラスである．シンボルクラスは型を持つ．型は C 言語において指定された変

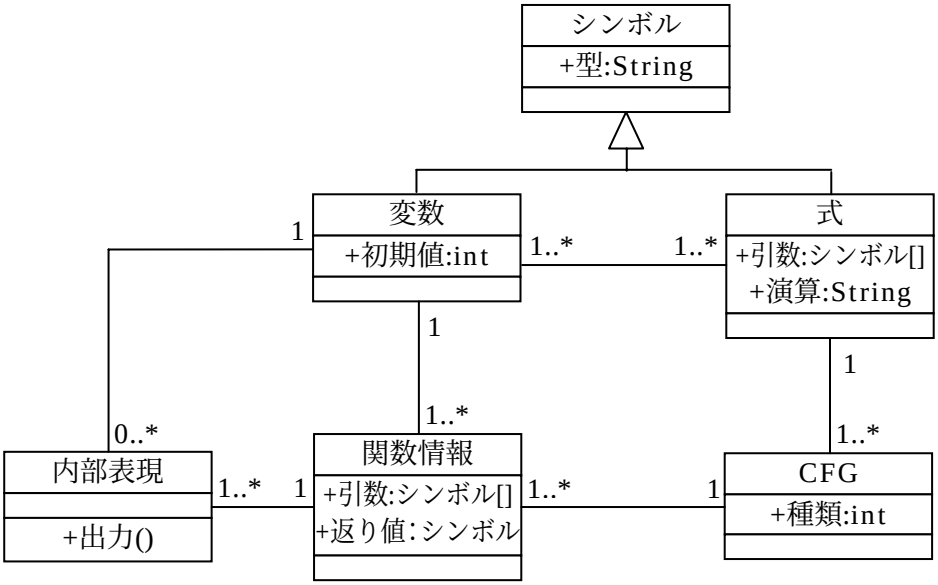

図 16　高抽象度内部表現のクラス図

数の型や，属性，配列の深さなどを示す情報である．変数クラスは変数の情報を示すクラスであり，変数の初期値の情報を格納する．初期値の値は型情報で表現できる値である．式クラスは変数クラスと異なり初期値を持つことができないが，値を返す式の処理とその引数の情報を保持する．処理は加算や乗算，関数呼び出しや配列の読み出しなどどのような処理を行うかという情報であり，引数は式の引数となる変数クラスや他の式クラスの情報である．式の引数は処理の内容によって異なる．

高抽象度の中間表現において十分に最適化を行った後，抽象度の高い中間表現を RTL へ変換する．RTL の中間表現も高抽象度の中間表現と同じように複数のクラスによって構成される．

(2) RTL 中間表現

RTL クラスはハードウェアに非常に近いレベルでの動作情報を示し，複数の State クラスから構成される．RTL クラスは関数の動作を表現するので，関数情報クラスから 1 関数につき RTL クラスが 1 つ生成される．RTL の中間表現に含まれるクラスの構造を図 17 に示す．

State クラスは RTL での遷移状態を示すクラスであり，CFG クラスから生成される．1 つの State クラスが 1 サイクルを示すため, CFG クラスから異なる処理により複数の State クラスが生成される．State クラスは内部に 1 サイクル中に行う処理の情報を Operator クラスによって含むと共に，次に遷移する状態を示す情報を含む．遷移する状態が内部の処理の結果によって異なる場合は，複数の遷移先と遷移の条件となる Operator クラスの情報を含む．

Operator クラスは処理を行う演算器の情報を示す. Operator クラスのインスタンスは加算器や乗算器, メモリアクセス用のポートなどであり, 式クラスから生成される．Operator クラスは内部に演算器の種類を判別する情報と，必要な回路規模の情報を持つ．Operator クラスはその処理対象として複数の Storage クラスを保持する．これにより例えば「加算

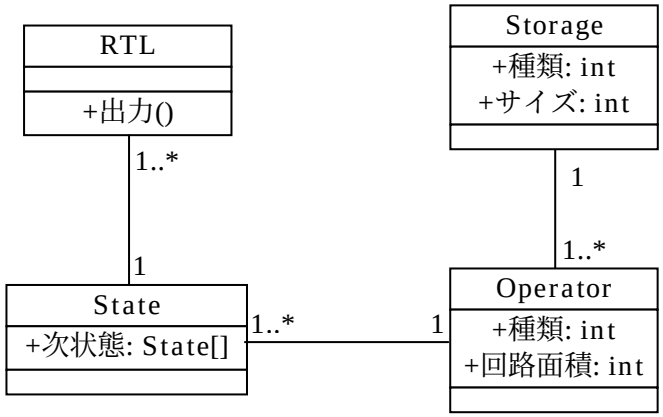

図 17　RTL 内部表現のクラス図

器 A の入力 1 にレジスタ A を，入力 2 にレジスタ B を，出力 1 にレジスタ C を接続する」という形で各クロックサイクルに実行される処理の情報を保持する．

Storage クラスは Operator クラスの処理結果や変数，定数のデータを保存するレジスタや配列の情報を示す．内部にはレジスタや配列などの種類，保存できるデータのサイズや属性などの情報を持つ．

中間表現の形式を情報の抽象度に合わせて規定することで，最適化の出力した中間表現を共通にすることができる．また抽象度に適した最適化が容易になり，また他の最適化結果に依存しない最適化が可能である．

4.3 最適化

最適化は中間表現それぞれに対して異なる処理を行う．中間表現の内容を解析し，回路面積や必要サイクル数などを改善するよう変更した中間表現を出力する．

既存のコンパイラなどで用いられてきた演算や制御に関する最適化は高抽象度の中間表現である内部表現クラスに対して行われる．これらの最適化は覗き穴最適化とも呼ばれ，動作記述中の非常に狭い範囲に対して行われる[18,19]．また，OpenMP の指示文からの並列化モデルの実現も高抽象度の中間表現における最適化で行う．OpenMP の並列化モデルの実現は設計者の指定した範囲に対して行われるため，非常に広い範囲の最適化が可能である．

内部表現クラスに対して適用可能な最適化を以下に示す．

- 内部不要式の削除

関数などの内部において，“ a=a; ” などの意味のない記述や，関数返り値など出力に影響しない不必要な式を削除する．

- 定数の畳み込み

“ const int one=1; two=2*one; ” など，中間表現で計算可能な式をあらかじめ計算することで，処理を削減する．他の最適化と組み合わせることで効果的な最適化が可能となる．

- ループ内不変式の移動

ループ内にありながらループの制御によって値が変わらない式がある場合，それをループの外側に出すことにより，処理を繰り返し行わないようにする．

- コピー伝搬

“ a=b; ” のような式があった場合，これ以降，変数 a を使用する場合は代わりに変数 b を使用することにより，代入処理が不要になる．さらに，その後変数 a が更新されなかった場合，変数 a 自体を削除可能で，レジスタなどを節約できる．

- 演算強度の削減

乗算や除算など処理に必要なコストが高い演算を，シフトと加減算などに置き換えることにより，処理に必要なコストを削減する．特に，“ a=b*3; ”のような定数演算の場合，“ a=b+b<<1 ”のように変更することで，サイクル数及び回路規模が削減可能である．

- OpenMP 実行モデル

OpenMP 指示文の情報を元に，指示文により指定された処理を，OpenMP の実行モデルに適するように並列化する．

- パイプライン並列化

タスク分割の処理が繰り返されており，しかも処理の結果に一方向の依存性しかない場合にパイプライン化が可能である．例えば，処理 A の出力を処理 B の入力に，処理 B の出力を処理 C の入力にしている場合，これら処理の結果をパイプラインバッファとなる変数に格納し，次の処理の入力とすることで，各処理を並列に処理が可能である．

RTL の中間表現である RTL クラスに対しては回路面積や必要サイクルといったハードウェアに非常に近い評価項目に対しての最適化を行う．一般的に必要サイクル数を減らすには演算器を増やすなど回路面積の増加が伴い，同様に回路面積の削減には演算に必要サイクル数が増加するなど，評価項目にはトレードオフとなる項目が多いため，設計者のハードウェア制約に記述された優先度に従って最適化を行う．

RTL クラスに対して適用可能な最適化を以下に示す．

- 演算器の共有

異なるサイクルで実行される処理で同じ種類の演算器を用いている場合，それらの処理で同じ演算器を使用することで回路規模を削減する．

- 演算器の最適化

加算と減算を異なるサイクルで実行している場合，加算器と減算器を加減算器へ統合することで回路面積の削減が可能である．また，乗算と加算が連続して実行される場合に，それを積和演算器へ割り当てることで回路面積の削減と高速化が可能である．

- 速度と回路のトレードオフ

処理においてレイテンシが重要となる場合，複数の処理を同時に実行することで必要サイクルを削減することが可能であるが, 1 サイクルで使用する演算器やレジスタなどが増加するため，回路面積が増加する．逆に回路面積を削減したい場合，処理を複数のサイクルに分割して実現することで，処理に必要な演算器を共有することが可能であるが，処理に

必要なサイクル数は増加する．必要サイクルと回路規模は相反する評価項目のため，設計者から与えられたハードウェア制約に従って調整を行う．

- 並列メモリアクセス

並列動作ハードウェアがメモリアクセスする場合，メモリアクセスが頻発すると，ストールが多発し，性能が低下する．そのため，出来る限り複数のハードウェアが同時にメモリアクセス可能なアーキテクチャを生成する必要がある．並列メモリアクセスを実現する例としてメモリバンクがある．メモリをアドレスによって複数のバンクに分割することで，異なるバンクには同時アクセスを可能にする．また，小規模なキャッシュを用いることで，キャッシュに保存されたデータへのアクセスはメモリを使用せずに済むため，結果的にメモリアクセスの回数を削減できる．

4.4 コード生成

コード生成では，RTL 中間表現から HDL へ変換する．State クラスから状態遷移機械，Storage クラスからレジスタやレジスタファイル，メモリを，Operator クラスから演算器を生成し，状態毎に処理を行うハードウェアの記述へ変換する．コード生成においては最適化などを行わず，中間表現の動作を正確に実行するハードウェア記述を生成する．

HDL は記述，及びシミュレーションが容易なことから，VerilogHDL を対象としている．

4.5 実装環境

動作合成システムは Java 言語によって実装し，必要な行数は構文・意味解析，中間表現を含めて約 6000 行である．コード生成部はツールとせず，RTL 中間表現から手作業で HDL へ変換し，それを用いて性能評価や検証を行った．

Java 言語は記述が容易で様々なプラットホームで動作が可能であるため移植が容易である．また記述や命名規則が推奨されており，複数の設計者が記述をしても可読性が高い．その結果，大規模開発に適するという特徴があるため，本システムの設計に用いた．

5. 動作合成システムの検証・評価

5.1 FIR フィルタ

FIR フィルタはディジタル信号処理における非常に一般的なフィルタであり，主に特定の周波数の信号を阻止したり，抽出する場合に用いられる．

FIR フィルタの振る舞いを示す伝達関数を図 18 に示す．

$$H(Z)=\sum_{m=0}^{M} h_m Z^{-m}$$

図 18　FIR フィルタの伝達関数

FIR フィルタは，図 18 の右辺（Z^{-m}）で示すように，出力信号を生成する際の計算に過去の入力信号列のみを用いる．同じく有名な IIR フィルタは出力信号の計算に入力信号列と，過去の自らの出力を用いるため，一種のフィードバックを含む．FIR フィルタはフィードバックを含まないため，出力信号の値が入力信号列とそれらに対する係数乗算で求めることが可能で，安定性が高い．図 18 では，m 個前の入力信号を Z^{-m} で示している．これらの入力列にフィルタ係数 h_m を乗算し，これらを足し合わせることによって出力信号を得る．入力信号及びフィルタ係数を何個用いるかを決める数は式中の M であり，次数もしくはタップ数と呼ばれる．FIR フィルタの構成法には直接形，転置形，継続形，格子形など多数の実現方法があるが，今回は図 18 の式をそのまま実現する直接形によって実現した．直接形 FIR フィルタの構成を図 19 に示す．

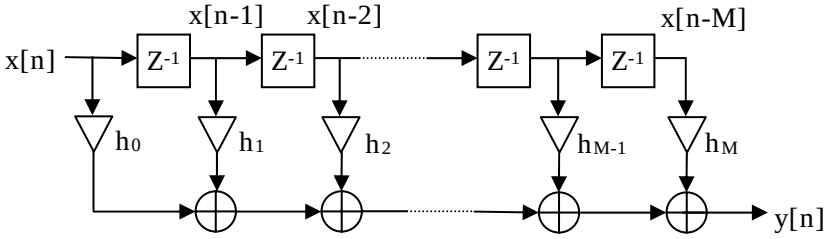

図 19　FIR フィルタの構成

各 Z^{-1} が過去の入力信号を保持し，それらに h_m を乗算することで実現する．1 回の計算毎に Z^{-1} の値が右へ更新されていき，M 個以上の過去の入力信号は破棄される．この構成をハードウェアで実現する場合，Z-1 をレジスタ，係数 h_m の乗算を定数乗算器で実現する．

5.1.1 SMP 環境での実行と並列化手法の評価

直接形の FIR フィルタを OpenMP で記述し, データ分割及びタスク分割による並列化を行った．そしてそれらをマルチプロセッサ上で実行し，並列化効果の測定を行った．今回実装し，性能検証に用いた FIR フィルタの次数は 16，入力データの数は 10000 である．しかし，最初の 15 個のデータにおいては，現在の入力と過去の入力を合わせてもデータの数が 16 に満たないため，その場合は足りない過去の入力を 0 として計算した．

データ分割による並列化では，FIR フィルタ自体を 1 つの並列動作ハードウェアとみなし，多数のデータに対して FIR フィルタを適用する場合に，複数の FIR フィルタでデータを分割して実行した．今回，2 ノード及び 4 ノードで並列化を行い，それぞれ 5000 と 2500 のデータに対して FIR フィルタを行った．

タスク分割による並列化では，FIR フィルタ内の処理を，データの移動と更新を Task1，係数の乗算を Task2，乗算結果の足し合わせを Task3 と，3 つに分割して実行した．これらの処理はデータに依存性があるため，最適化によってパイプライン化されている．

FIR フィルタのデータ分割及びタスク分割の実行モデルを図 20，図 21 に示す．

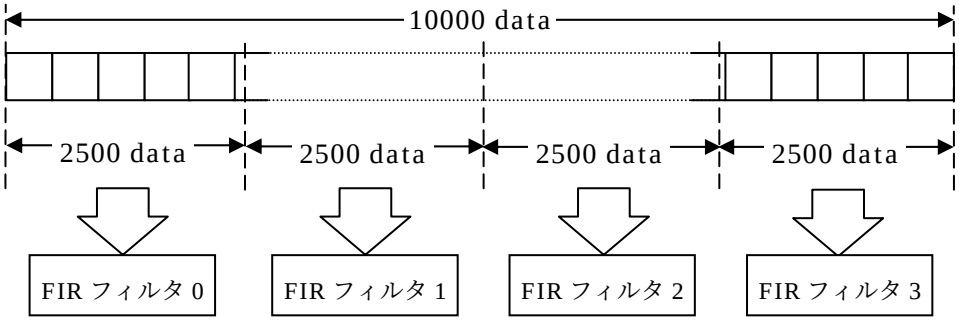

図 20　FIR フィルタのデータ分割による実行モデル

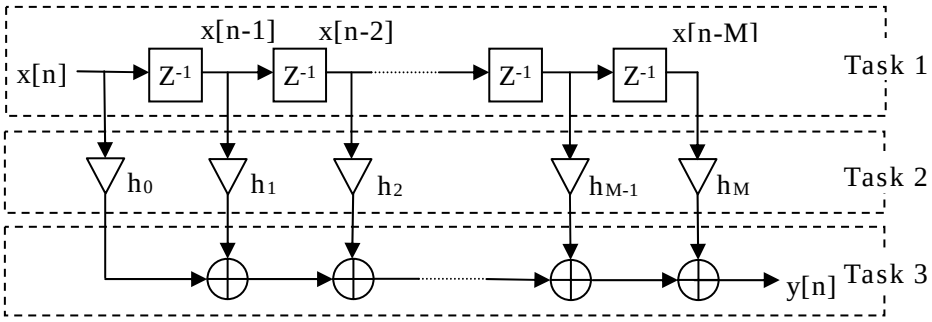

図 21　FIR フィルタのタスク分割による実行モデル

これらの構成の OpenMP 動作記述を OpenMP コンパイラでマルチスレッドプログラムに変換し，SMP 環境で実行した．OpenMP コンパイラは Intel C/C++ Compiler 9.0 を使用した．実行環境は，CPU として Intel 社 Dual Core Xeon 2.8G[Hz]を 2 個搭載した Quad Core のマルチプロセッサ PC であり，搭載メモリは 4G [Byte]，搭載 OS は Windows XP Professional SP2 である．表 1 に実行結果を示す．

表 1　次数 16 の FIR フィルタの SMP 環境での実行結果

	逐次プログラム	タスク分割 (3 ノード)	データ分割 (2 ノード)	データ分割 (4 ノード)
必要時間(ms)	12.80	29.42	9.06	7.48
速度向上比	1	0.43	1.41	1.71

データ分割では 2 ノードで 1.41 倍，4 ノードで 1.71 倍の速度向上が得られた．ノード数に比べ並列化効果が得られなかったのは，fork 処理と比べ並列実行した処理の負荷が小さかったためである．

タスク分割では逐次プログラムでの実行と比べ 0.43 倍と速度が低下した．これは，データ分割では fork 処理の内部でループによる繰り返し処理が展開されるのに対し，タスク分割ではループによる繰り返し処理の内部で fork 処理を行うため，データ分割と比べ fork の処理の割合が大きくなったためと考えられる．

fork 処理に対し並列処理の負荷を増やした場合を評価するため，より並列化効果が大きくなるように FIR フィルタの次数を増やして実行した．その結果を表 2 に示す．

表 2　次数 128 の FIR フィルタの SMP 環境での実行結果

	逐次プログラム	タスク分割 (3 ノード)	データ分割 (2 ノード)	データ分割 (4 ノード)
必要時間(ms)	71.34	162.67	41.20	31.88
速度向上比	1	0.43	1.73	2.23

次数を増やして実行すると，FIR フィルタの処理の負荷が大きくなるため，データ分割では並列化効果が向上した．これは FIR フィルタの処理の負荷が変化しても fork でのスレッド生成のコストが固定であるため，FIR フィルタの負荷が大きくなることで，fork の処理負荷が相対的に減少し，その結果 FIR フィルタの並列化効果が向上した．データ分割と比べ，タスク分割では並列化効果の向上は見られなかった．これは乗算などの負荷の重い処理では高速化されたが，データの移動や加算などは次数を増やしてもあまり負荷が大きくならないため，これらの処理がボトルネックとなったためである．

fork で行われるスレッドの生成のコストは OS やその実装により大きく異なるが，一般的に数百から数千サイクルが必要な処理であり，並列処理により得られる並列化効果が小さい場合，逆に並列化によって低速になる場合があるこのような場合，並列化の処理の範囲を大きくするなど，並列化の処理の負荷を大きくし，fork によるコストを上回る十分な並列化効果を得られるようにしなければならない．

5.1.2 動作合成ハードウェアの評価

SMP 環境で検証を行った次数 16 の FIR フィルタの OpenMP による記述をハードウェア合成系へ入力し，出力された RTL 中間表現を元に並列動作ハードウェアを作成した．FIR フィルタの構成は，図 19 の Z^{-1} をレジスタに，h_n の係数乗算を定数乗算器に，乗算結果の加算を加算器に変換した構成となる．FIR フィルタのデータ分割による構成を図 22 に，タスク分割による構成を図 23 に示す．データ分割では，並列動作ハードウェアは複数の FIR フィルタで構成される．タスク分割では並列動作ハードウェアには図 21 で示した FIR フィルタを構成する各タスクと，それを接続するパイプライン・バッファで構成される．

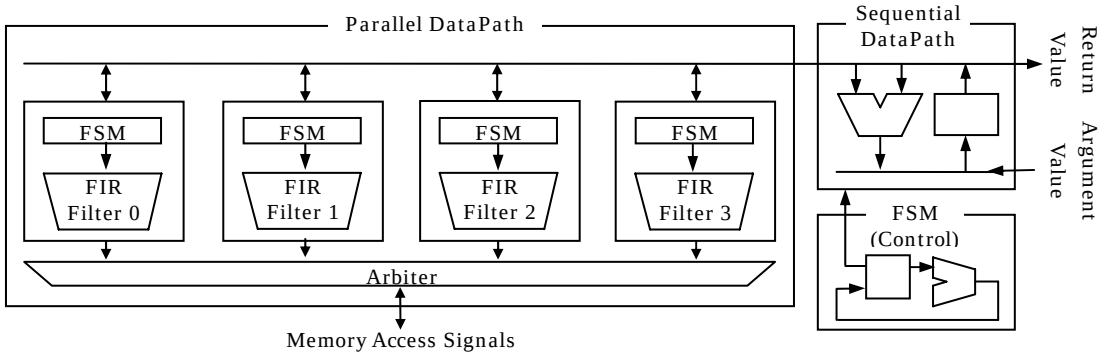

図 22　FIR フィルタのデータ分割によるハードウェア構成

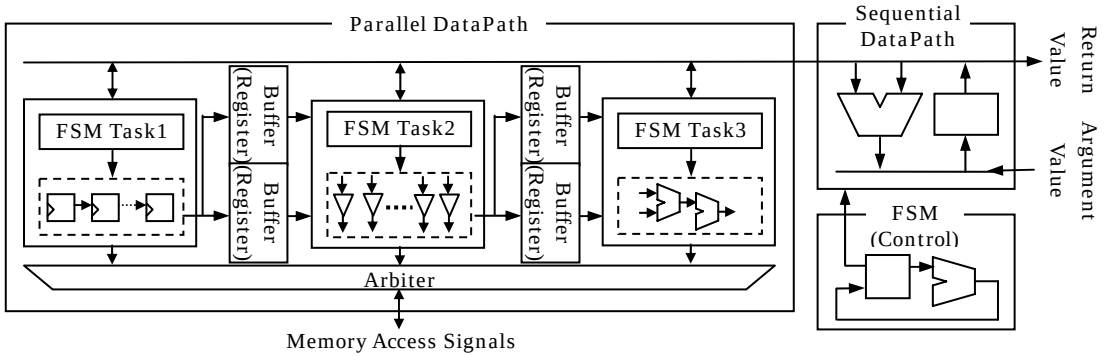

図 23　FIR フィルタのタスク分割によるハードウェア構成

並列動作ハードウェアを用いてサイクルレベルシミュレーションを行い，その性能を評価した．また，各並列動作手法に対し論理合成を行い，回路の実装に必要な回路面積を算出した．これらの評価に用いたツールは，サイクルレベルシミュレータには Menter Graphics 社の ModelSim SE 5.8c を用い，論理合成には Xilinx 社の ISE7.1 を用いた．性能評価においては，サイクルレベルシミュレーションにおいて SMP 環境と同じ結果が出力されることも確認した．表 3 にその評価結果を示す．

表 3　FIR フィルタの並列動作ハードウェアによる評価結果

	逐次プログラム	タスク分割 (3 ノード)	データ分割 (2 ノード)	データ分割 (4 ノード)
必要時間(10^6 cycle)	31.7	20.6	15.9	8.0
速度向上比	1	1.54	1.99	3.96
回路面積(Slices)	1256	2595	2655	5248
回路面積比	1	2.06	2.11	4.17

データ分割において，2 ノードで 1.99 倍，データ分割で 3.96 倍と非常に高く，理想に近い速度向上が得られた．また，タスク分割では SMP 環境の実行時には速度の低下が見られたが，並列動作ハードウェアでは 1.54 倍の並列化効果が得られている．並列動作ハードウ

ェアの動作は SMP 環境での実行と変わらないため，これは fork におけるコストが減少したからである．

SMP 環境での fork は OS によってスレッドの生成が行われ，並列動作が開始される．それに比べ，ハードウェアでは実行に応じてハードウェアを動的に生成することは難しいため，あらかじめ必要と思われるハードウェアを静的に生成しておき，それらを起動することで fork を行う．そのため，並列動作に必要な値のコピーが必要ではあるが，処理ノード自体の生成は必要がなくなり，fork に必要なコストが非常に小さくなる．このため，SMP 環境と比べて非常に高い並列化効果が得られた．

処理の開始以前に静的に並列ノードを生成しておく並列動作ハードウェアは，動的に処理ノードを生成する SMP 環境での実行と異なり，逐次処理プログラムと比べより多くの回路面積が必要となる．

データ分割は 2 ノードで 2.11 倍，4 ノードで 4.17 倍とノード数を上回る割合で回路面積が増加している．これは，単純にノード数を N 倍すると，必要な回路が N 倍されるのに加え，共有変数へのアクセスを制御するアービタや，逐次処理部分に加えられる並列動作ハードウェアの制御用のレジスタなどが必要となるためである．

タスク分割では，3 ノードで 2.06 倍の回路面積となった．タスク分割では元々1 つだった処理を分割しているため，処理に必要な回路は増加しないが，データ分割と同じようにアービタや並列動作ハードウェアを制御するためのレジスタなどが加わっている．データ分割のアービタなどに必要な回路面積と比べ，タスク分割での回路の増加が著しいが，これはタスク分割においてパイプライン化を行ったため，並列処理間のデータを保存しておくパイプライン・レジスタが必要となり，回路面積が大きく増加した．特に FIR フィルタは過去の信号列や計算の途中結果など，処理したデータを全て次の処理へ渡すため，多くのパイプライン・レジスタが必要となり，回路面積の増加の原因となった．

5.2 ウェーブレット変換

時間－周波数変換処理を行うウェーブレット変換は周波数変換処理を行うフーリエ変換と同じように，スペクトル解析に用いられる信号処理の一つであり，画像や音声の特性の解析に用いられる[20].

フーリエ変換は入力された信号列に含まれる周波数成分を解析する処理であり，ディジタル信号処理において非常に広く用いられている．フーリエ変換に入力される信号列は時間によって変化しないことが前提であり，信号列全体に含まれている周波数成分は信号列全体においてどれぐらい含まれているかという値として出力されるため，信号列のどの部分にどれだけの周波数成分が含まれているかはわからない．しかし，音声や画像などの処理においては，信号列の周波数成分が部分によって異なることが多く，信号列のどの部分にどの程度の周波数成分を含むかという情報は，その特性の解析において非常に重要である．時間窓と呼ばれる信号列に比べて比較的小さな範囲を切りだし，それをフーリエ変換することで位置情報と周波数情報の両方を得ることが出来るが，高い周波数の情報を得るためには時間窓を狭くする必要があり，低い周波数の情報を得るには時間窓を広くする必要がある．しかし時間窓を狭くすると得られる周波数の幅が狭くなるため，周波数情報とその位置の情報がトレードオフの関係になり，位置情報と周波数情報の両方を高精度で求めることはできない.

ウェーブレット変換では，ウェーブレットと呼ばれるある周波数の近傍のみを含む短い波 $\Psi(t)$を用いて周波数解析を行う．ウェーブレット変換ではウェーブレットを信号列上で移動させ，ある位置でどの程度ウェーブレットと一致するかを調べ，周波数毎の位置情報を得る．異なる周波数の情報を得るためには異なる周波数のウェーブレットが必要になるが，ウェーブレット変換では基本となるウェーブレットを拡大・縮小することにより異なる周波数のウェーブレットを得る.

基本となるウェーブレットをマザーウェーブレットと呼び，マザーウェーブレットによりことなる性質を持つ時間－周波数解析が可能である．ウェーブレットの基本となる定義を図 24 に示す．式中の a はウェーブレットの拡大・縮小のスケールを，b は位置情報を得るためのウェーブレットの移動量を表す.

$$\psi_{a,b}(t)=\frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right)$$

図 24　ウェーブレット変換の定義

本研究では最も簡単なウェーブレットを用いる Haar のウェーブレット変換を実装し, 評価を行った．Haar のウェーブレットは計算が簡単であることが大きな特徴であり，加算及び減算，シフト演算のみで実現可能である．データ数が 8 の場合のウェーブレット変換の処理の流れを図 25 に示す．Haar のウェーブレットでは計算において与えられた信号列の隣り合う 2 つの信号を加算，及び減算し，それらの結果を 2 で除算し，一定の法則によっ

て求められた位置に格納する．2 の除算は 1bit 右シフトで実現可能であるため，加算器と減算器，シフト演算器で実現する事が可能であり，ハードウェアでの処理に適している．

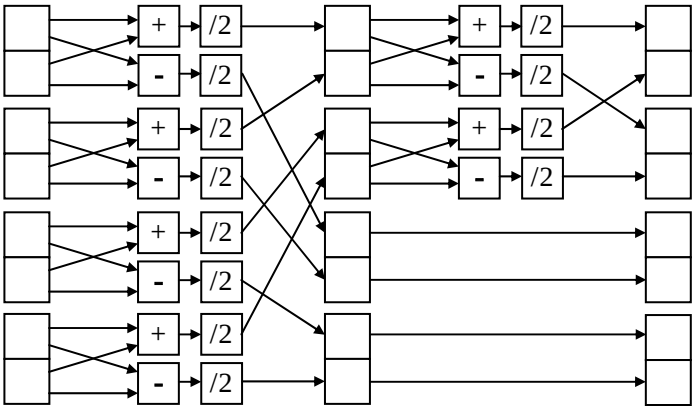

図 25　ウェーブレット変換の処理の流れ

5.2.1 SMP 環境での実行と並列化手法の評価

Haar のウェーブレット変換を OpenMP で記述し，データ分割及びタスク分割による並列化を行った．ウェーブレット変換は 256 データを 1 単位として実行し，10000 単位のデータを入力として与えた．

データ分割では，FIR フィルタと同じようにウェーブレット変換を 1 つの並列動作ハードウェアとし，2 ノード及び 4 ノードでデータを分割し，並列処理を行った．

タスク分割では，ウェーブレット変換の処理において並列で演算可能な加算と減算，そしてそれぞれの 2 の除算の処理を分割した．FIR フィルタでのタスク分割がループ単位での処理だったのに対し，ウェーブレット変換では演算単位での並列化を行った．タスク分割による処理の実行モデルを図 26 に示す．

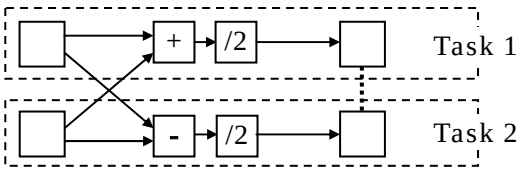

図 26　ウェーブレット変換のタスク分割による実行モデル

ウェーブレット変換の OpenMP 動作記述を OpenMP コンパイラでマルチスレッドプログラムに変換し，SMP 環境での実行した．SMP 環境は FIR フィルタと同一である．その結果を表 4 に示す．

データ分割では 2 ノードで 1.53 倍，4 ノードで 2.67 倍の速度向上が得られた．ウェーブレット変換は FIR フィルタよりも処理の負荷が大きく，より高い並列化効果が得られると考えたが，そのような結果にはならなかった．これは FIR フィルタと異なり，共有メモリへ

表 4 ウェーブレット変換の SMP 環境での実行結果

	逐次プログラム	タスク分割 (2 ノード)	データ分割 (2 ノード)	データ分割 (4 ノード)
必要時間(ms)	87.5	1803.9	57.1	32.8
速度向上比	1	0.04	1.53	2.67

のアクセスが多く，複数同時メモリアクセスの頻発によって速度が低下し，並列化効果が得られなかったためと考えられる．

ウェーブレット変換では FIR フィルタのように処理単位を 256 から変化させたり，処理の単位データの量を変化させても速度向上比は変化しなかった．つまり，スレッド生成などの fork の処理により並列化効果が低くなってしまったのではなく，アルゴリズム自体に共有メモリアクセスが多く，それがボトルネックとなったと考えられる．それを証明するように，2 ノードでは 1.53 倍と比較的高い並列化効果が得られているが，4 ノードでは 2 ノードよりも並列化効果が低減している．これは 2 ノードよりも 4 ノードの時の方が共有メモリアクセスの頻度が増し，ボトルネックの影響が顕著に表れているためである．

タスク分割では FIR フィルタよりも速度の低下が著しい．逐次処理の場合と比べ，約 25 倍もの時間が必要となっている．これは並列化を行った処理が FIR フィルタではループ処理だったものが演算処理になり，より小さくなったことで，fork の処理のコストが非常に高価になり，並列化効果が全く得られなかったことを示している．

5.2.2 動作合成ハードウェアの評価

SMP 環境で検証を行ったウェーブレット変換の OpenMP による記述をハードウェア合成系へ入力し，出力された RTL 中間表現を元に並列動作ハードウェアを作成した．

データ分割では FIR フィルタと同様にウェーブレット変換を行うハードウェアを複数並べて並列動作ハードウェアを構成している．データ分割による並列動作ハードウェアの構成を図 27 に示す．

タスク分割では図 26 に示す加算と減算，それらの結果への右 1bit シフト演算の並列化を行うため，並列動作ハードウェアは加算器と減算器，右シフト演算器を 2 つ並べた構成となる．FIR フィルタに比べ非常に小さな構成となる．タスク分割による並列動作ハードウェアの構成を図 28 に示す. データ分割では並列動作ハードウェアは複数のウェーブレット変換を行うハードウェアで構成される．タスク分割においては，並列動作ハードウェアにはウェーブレット変換に含まれる同時に演算可能な加算と減算，及び右シフトのハードウェアによって構成される．

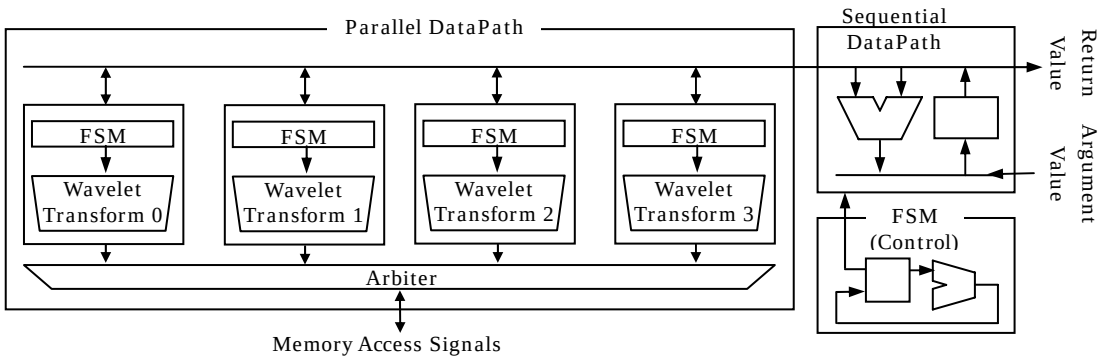

図 27　ウェーブレット変換のデータ分割によるハードウェアの構成

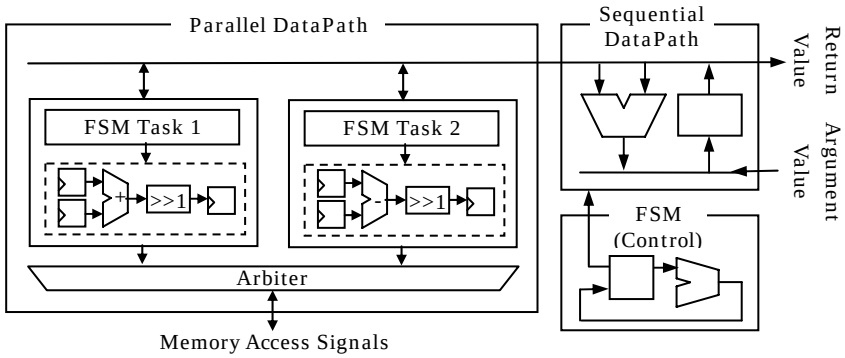

図 28　ウェーブレット変換のタスク分割によるハードウェアの構成

並列動作ハードウェアを用いてサイクルレベルシミュレーションを行い，その性能を評価した．また，各並列動作手法に対し論理合成を行い，回路の実装に必要な回路面積を算出した．評価環境及び回路面積に使用したツールは FIR フィルタと同じである．表 5 にその評価結果を示す.

表 5　ウェーブレット変換の並列動作ハードウェアによる評価結果

	逐次プログラム	タスク分割 (3 ノード)	データ分割 (2 ノード)	データ分割 (4 ノード)
必要時間(10^6 cycle)	100.69	115.99	50.38	25.21
速度向上比	1	0.87	1.99	3.99
回路面積(Slices)	21004	29508	52875	104646
回路面積比	1	1.40	2.51	4.98

データ分割では FIR フィルタと同様に理想的な速度向上が得られた．これは FIR フィルタと同様に fork における処理のコストがハードウェアでは非常に低いことが一つの原因である．しかし，SMP 環境での実行結果では，共有メモリアクセスがボトルネックとなり，並列化する処理の負荷を大きくし，fork のコストを相対的に小さくしても並列化効果は高くはならなかった．それに対してハードウェアでは非常に高い並列化効果が得られている．

データ並列においてハードウェアで SMP 環境での実行と異なり，高い並列化効果が得られたのは共有メモリへのアクセスが高速に行うことが可能だったからである．SMP 環境での実行では，共有メモリへのアクセスは専用のハードウェアがあるわけではなく，スレッドの生成と同じように OS やライブラリのルーチンコールによって行われる．これも非常に多くのサイクル数が必要となる．しかし，ハードウェアでは共有メモリへのアクセスはアービタが調停を行うため，最小のサイクル数でアクセスが可能である．そのため共有メモリへのアクセスが SMP 環境での実行ほど並列化効果の負担にならず，理想的な速度向上が得られた．

また，メモリアクセスの頻度が低かったことも理想的な速度向上が得られたもう一つの原因である．すべての並列動作ハードウェアを同時に起動すると，当然ながら共有メモリへの最初のアクセスは競合し，アービタによる調停が行われる．その結果，各並列動作ハードウェアは 1 サイクルずつ動作がずれる．これにより，N 個のハードウェアが同時に動作するとき，メモリアクセスが N サイクルに 1 回より多い頻度でなければ，以降のメモリアクセスでメモリアクセスの競合は発生しない．この動作の簡単な例を図 29 に示す．

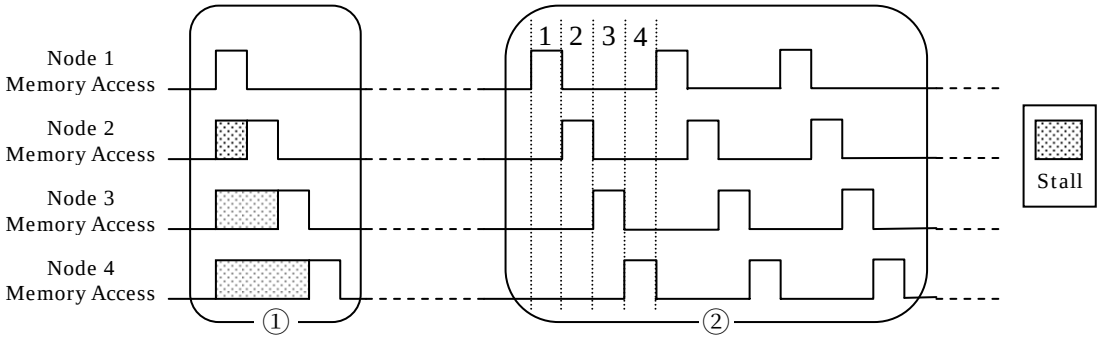

図 29　4 ノードでの共有メモリへのアクセス

図 29 では図中の①の部分では，同時にメモリアクセスが発生し，それにより各ノードにストールが出力されている．本例ではノード数の小さいノードが優先的にメモリアクセスが可能である．この場合，最初のサイクルはノード 1 がメモリアクセスし，他のノードはストールによってメモリアクセス要求を出力したまま停止している．次のサイクルではノード 2 がメモリアクセスが可能で，ノード 3 とノード 4 はストールにより停止している．全てのノードがメモリアクセスを完了したとき，各ノードは 1 サイクルずつ動作が遅れた状態になる．

図 29 の②では，連続してメモリアクセスが発生している．しかし，本来ならメモリアクセス毎にストールが発生する状況であるが，①において各ノードの動作が 1 サイクルずつずれているため，メモリアクセスのタイミングもずれ，結果的にストールが発生しない．本例のような 4 ノードの場合，各ノードのメモリアクセスが 4 サイクルに 1 回以下の頻度であれば，これ以降のメモリアクセスにおいてもストールは発生しない．

これはすべてのノードが同じ動作を行う理想的な例であり，データに依存した分岐など

があった場合，メモリアクセスの間隔はノード毎に異なる．しかし，最適化によって，メモリアクセスの頻度をノード数と同じサイクル数に 1 回程度にできれば，ストールが発生する状況を大幅に削減することが可能であることがわかった．ウェーブレット変換のハードウェアではメモリアクセスが 5 サイクルに 1 回の頻度で行われていたため，ストールが発生したのは最初のサイクルのみであった．

また，ストールが発生しにくいメモリアクセスの頻度はアービタの構造によって異なり，同時に 2 つのノードがメモリアクセス可能な場合，頻度はノード数の半分のサイクルに 1 回でも良い．

ストールが発生しにくいメモリアクセスの頻度により，最適化において最も効率の良い処理のサイクル数を決めることが可能である．無理に演算器などの資源を消費してサイクル数を削減しても，共有メモリへのアクセスが頻発しては，結果的にストールが多発し，処理に必要なサイクル数が削減できない．メモリアクセスの頻度を考慮することにより，回路面積を増加しても必要サイクル数が削減できないという状況を防ぐことができる．

タスク分割による並列化の評価結果では，FIR フィルタと異なり速度向上が得られず，逆に悪化した．FIR フィルタでは SMP 環境での実行では速度が低下してもハードウェアでは速度向上が得られていたが，ウェーブレット変換では SMP 環境での実行と同じくハードウェアでも速度が低下した．

これは FIR フィルタとウェーブレット変換のタスク分割で並列化した処理の粒度が異なるから発生した違いである．FIR フィルタではループ単位で並列化を行ったため，SMP 環境では速度が低下したが，fork 処理のコストの違いによりハードウェアでは速度向上が得られた．ウェーブレット変換では演算単位という非常に粒度の小さい処理の並列化であり，fork 処理のコストが小さいハードウェアでも，並列動作ハードウェアに対する変数の値のコピーなど並列動作の準備のコストが並列処理により得られた利益よりも大きくなり，速度向上が得られなかったと考えられる．

回路面積においては，タスク分割では演算器を 2 つにしただけで 1.4 倍もの回路規模になった．これは並列動作のためのアービタや，各並列動作ハードウェア内のレジスタなどにより回路規模が増加したためである．データ分割では同様にアービタなどの回路が加わるためにノード数を N 倍した場合に回路規模は N 倍以上になった．

5.3 動作合成システムの評価

ハードウェア動作合成系を用いることで，データ分割により理想的な速度向上が得られるハードウェアを生成することが可能であった．しかし，タスク分割では処理の粒度にある程度の大きさがなければ速度向上が得られなかった．

OpenMP は元々，大規模な処理の並列化のための API であり，小さな処理の並列化には向いていない．そのため合成された並列動作ハードウェアが小さな処理の並列化が効率よく行えないことは問題ではなく，演算や制御レベルの最適化によって並列化を行うべき問題である．粒度の小さな処理の並列化に対し，ハードウェアにのみ有効な特別な並列化を行うことは可能であるが，SMP 環境での実行結果をハードウェアの性能評価に用いることができなくなる．これは本動作合成システムの目的と反する．

今回の FIR フィルタ及びウェーブレット変換を用いた評価において，OpenMP の SMP 環境での実行の特徴がハードウェアでも見られ，SMP 環境での実行結果や評価結果を元にハードウェアでの並列化手法の検討が十分に可能であることがわかった．しかし，fork 処理や共有メモリアクセスのコストは SMP 環境ではサイクル数の増加に現れるが，ハードウェアでは回路規模の増加に繋がる．これらの違いを考慮したアルゴリズムや並列化手法が必要となる．

また OpenMP の実行モデルのハードウェア化において，アービタなど共有メモリアクセスのためのアーキテクチャにより並列化が大きな影響を受けることがわかった．中間表現における最適化において，回路面積や必要サイクルなどの最適化する際，ハードウェア制約などにアービタなどの仕様を与えることにより，より効率的な回路の生成が可能であると考えられる．

最後に SMP 環境での実行とハードウェアによるサイクルレベルシミュレーションに必要な時間を表 6 と表 7 に示す．

表 6　FIR フィルタのシミュレーション時間

単位：ms

	逐次プログラム	タスク分割 (3 ノード)	データ分割 (2 ノード)	データ分割 (4 ノード)
SW	71.34	162.67	41.20	31.88
HW	3968	4375	4672	4938

表 7　ウェーブレット変換のシミュレーション時間

単位：ms

	逐次プログラム	タスク分割 (2 ノード)	データ分割 (2 ノード)	データ分割 (4 ノード)
SW	87.5	1803.9	57.1	32.8
HW	652719	842047	2333859	4239344

表中の SW が SMP 環境での実行に必要な時間であり, HW がハードウェアによるサイクルレベルシミュレーションでの必要時間である．FIR フィルタとウェーブレット変換の双

方において，SMP 環境での実行はハードウェアによるサイクルレベルシミュレーションと比べ数十から十万倍程度高速に実行できる．

しかも，ハードウェアによるサイクルレベルシミュレーションでは回路の並列化を行うと必要な回路面積が増加し，シミュレーションに必要な時間が増加しているが，SMP 環境での実行では並列化により必要時間が減少している．これはサイクルレベルシミュレーションでは並列化による回路面積の増加により，サイクルレベルでのイベントが増加し，処理が増加したからである．それに対し SMP 環境での実行では並列化により並列処理ノードに処理を分担させ，必要時間が削減できる．

高性能・高機能な回路の作成には処理の並列化は必須である．並列処理を多く含む回路は設計の難しさは当然ながら，シミュレーションに必要な時間が非常に多く，設計初期での検討が難しい．しかし，本システムを用いることにより，並列処理を多く含めば含むほど高速なシミュレーションが可能であり，SMP 環境において並列動作ハードウェアのアルゴリズムや並列化手法の検討が容易になる．

設計生産性を向上させる手法として，本システムが実現した OpenMP を用いたハードウェア動作合成手法は非常に有効であると評価できる．

6. おわりに

本研究では，LSI の設計生産性の危機を解決する技術として，並列プログラミング言語である OpenMP からのハードウェア動作合成手法を提案し，その評価を行った．評価においては実際に OpenMP からハードウェアを自動で生成するシステムを構築し，FIR フィルタとウェーブレット変換というアプリケーションを対象にその性能評価を行った．

OpenMP 記述からハードウェアを合成することで，大量のデータに対する繰り返し処理においては理想的な速度向上が得られることがわかった．しかし，ハードウェアによる並列化は SMP 環境でのソフトウェアでの並列処理とは異なり，処理の開始以前に回路を生成しておく必要がある．動作中に並列処理のための準備が必要ないので，速度向上は得られやすいが，回路面積が速度向上比より多く増加する．

今後の課題は，回路面積削減の最適化を行うことと，システムとしての完成させることである．OpenMP では並列処理の手順ではなく構造を記述することが可能であり，既存のコンパイラにおける局所的な最適化だけではなく，プログラムの構造を考慮した大規模な最適化が可能である．

近年，LSI を構成するトランジスタの数は著しく増加し，非常に多機能かつ高性能な ASIC や，複数のコアを含んだプロセッサが一般的になりつつある．だが同時に過度に増大したトランジスタが複雑化を招き，LSI 設計者の手に余るようになっている．膨大なトランジスタを利用した並列処理環境が広がる中で，それを上手く利用し，さらに膨大なトランジスタで構成される回路を効率的に設計する手法が求められている．

本研究で提案した動作合成手法が新たな回路設計の手法として用いられ，上記の問題を解決する手段となることを望む．

謝辞

本研究の機会を与えてくださり，貴重な助言，ご指導をいただきました山崎勝弘教授，小柳滋教授に感謝いたします.

また，動作合成に関して様々なご意見やご指摘をいただいた名古屋大学の冨山宏之助教授, C 言語からの高位合成ツールについてご意見をいただいた関西学院大学の石浦菜岐佐教授，財団法人京都高度技術研究所ソフトウェア研究室室長の神原弘之氏，ならびに CCAP を開発されている関西学院大学の学生の皆様に感謝いたします.

本研究に関して貴重なご意見をいただきました高性能計算研究室の皆様に感謝致します.

参考文献

[1] 森江善之, 冨山宏行, 村上和彰:“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, 2003.

[2] 鈴木, 亀井, 富田, 森下, 山田, 久保:“C 言語によるサイクル精度でのハードウェア/ソフトウェア協調検証手法”, シャープ技法第 92 号, pp.78-83, 2005.

[3] 西口健一, 石浦菜岐佐, 西村啓成, 神原弘之, 冨山宏之, 高務祐哲, 小谷学: "ソフトウェア互換ハードウェアを合成する高位合成システム CCAP における変数と関数の扱い" 電子情報通信学会技術研究報告, VLD2005-79, ICD2005-174, DC2005-56, pp. 19-24.

[4] 岩田, 田中, 山崎:“C 言語による有限ステートマシンベースのプロセッサ生成”, 信学技報,VLD2001-07,ICD2001-162,FTS2001-64,pp33-38, 2001.

[5] 松田昭信, 南谷崇:“高位合成手法を用いた C ベース設計による LSI 開発事例”, 情報処理学会第 67 回全国大会論文集分冊 1, pp.99-100,2005.

[6] 松浦努, 内田順平, 宮岡祐一郎, 戸川望, 柳澤政生, 大附辰夫,“ネットワークプロセッサ合成システム”, 電子情報通信学会技術研究報告, VLD2003-145, pp.55-60,2003.

[7] Yuichi Nakamura, Kouhei Hosokawa, Ichiro Kuroda, Ko Yoshikawa, Takeshi Yoshimura:“A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication”,Annual ACM IEEE Design Automation Conference, pp.299-304, 2004.

[8] Spec C：http://www.specc.gr.jp/

[9] Handel-C：http://www.celoxica.co.jp/products/system_tools.asp/

[10] OpenMP : http://www.openmp.org/.

[11] Intel C/C++ Compiler：http://www.intel.com/

[12] PGI Compiler：http://www.pgroup.com/

[13] Omni OpenMP Compiler Project：http://phase.hpcc.jp/Omni/home.html

[14] 上平祥嗣:“並列アルゴリズムクラスに基づくハードウェア自動生成”,立命館大学大学院理工学研究科修士論文,2000.

[15] 松井誠二:“並列プログラムからのハードウェア自動生成システムの検討”, 立命館大学大学院理工学研究科修士論文,2002.

[16] David A.Patterson,John L.Hennessy,成田光彰訳:“コンピュータの構成と設計”,第 2 版,上,日経 BP 社,1999.

[17] David A.Patterson,John L.Hennessy,成田光彰訳:“コンピュータの構成と設計”,第 2 版,下,日経 BP 社,1999.

[18] 中田育夫:コンパイラの構成と最適化,朝倉書店,1999.

[19] 今城哲二,布広永示, 岩澤京子,千葉雄司:“コンパイラとバーチャルマシン”オーム社,2004

[20] 三谷政昭: “ やり直しのための信号数学一DFT, FFT, DCT の基礎と信号処理応用 ”, CQ 出版,2004.
[21] 五月女健治: “ JavaCC コンパイラ・コンパイラ for Java ”,テクノプレス,2003.