修 士 論 文

動的リコンフィギャラブルプロセッサにおける入出力と演算のオーバーラップを用いたループネストの高速化に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

荒木 光一

2008 年 3 月

修 士 論 文

動的リコンフィギャラブルプロセッサにおける入出力と演算のオーバーラップを用いたループネストの高速化に関する研究

指導教官　井口 寧 准教授

審査委員主査　井口寧 准教授
審査委員　松澤照男 教授
審査委員　田中清史 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

610004 荒木 光一

提出年月: 2008 年 2 月

概 要

リコンフィギャラブルデバイスは、任意の回路を何度でも構成することができるデバイスである。近年のリコンフィギャラブルデバイスは、集積度の向上により大規模な回路や多数の回路を実現することが可能となった。これにより、従来の CPU ではボトルネックとなるループネストの処理をリコンフィギャラブルデバイスで並列処理することによって高速に処理を行うことが可能となった。しかし、リコンフィギャラブルデバイスの I/O ビット幅は小さいので、並列処理を行うために必要なデータを 1 クロックで入力することは不可能である。従って、リコンフィギャラブルデバイスで並列処理するためには、演算で必要なデータを複数クロックで入力しなければならない。出力に関してもデータ入力と同じように複数クロックでデータを出力する必要がある。この結果、入出力データが多い処理をリコンフィギャラブルデバイスで行った場合、全実行時間においてデータ入力時間がボトルネックとなる可能性がある。そこで、本研究ではリコンフィギャラブルデバイスの一種である動的リコンフィギャラブルプロセッサを用いてデータ入出力と演算処理をオーバーラップさせる手法を提案する。

本研究では、マルチコンテキスト型動的リコンフィギャラブルプロセッサである NEC エレクトロニクス社の Dynamically Reconfigurable Proseccer(DRP) を対象デバイスとする。DRP は回路情報であるコンテキストを 16 個保存することが可能である。しかし、ある処理が要求するコンテキスト数が 16 個を超えた場合、実行中にホストメモリから保存できなかったコンテキストをダウンロードする必要がある。この結果、ダウンロード時間がボトルネックとなり高速に処理することができない可能性がある。本研究では、コンテキストを削減するために、データ駆動方式によるコンテキストの削減を提案する。

画像処理のラプラシアン・フィルタ処理で実験を行った。その結果、本研究の提案手法の実行クロック数は、リコンフィギャラブルデバイスによる従来手法と比較して最大 55%に削減することができた。実行時間では、Pentium4 2.80GHz と比較して実行時間を 150%向上させたが、本研究の提案手法では、最大 230%向上させることができた。コンテキスト数の削減に関しては、従来手法の 50%のコンテキスト数に削減することができた。

目 次

第 1 章　序論 **1**
1.1 研究背景 1
1.2 研究目的 4
1.3 本論文の構成 6

第 2 章　リコンフィギャラブルデバイスの利用 **7**
2.1 はじめに 7
2.2 リコンフィギャラブルデバイス 7
2.2.1 Feild Programmable Gate Array (FPGA) 7
2.2.2 FPGA の問題点 8
2.2.3 動的部分再構成 10
2.2.4 マルチコンテキスト方式 10
2.2.5 動的リコンフィギャラブルプロセッサ 11
2.3 関連研究 12
2.4 まとめ 13

第 3 章　動的リコンフィギャラブルプロセッサを利用した提案手法 **15**
3.1 はじめに 15
3.2 Dynamically Riconfigurable Processor (DRP) 15
3.2.1 DRP の概要 15
3.2.2 Tile アーキテクチャ 16
3.2.3 Processing Element アーキテクチャ 18
3.2.4 DRP-1 アーキテクチャ 19
3.2.5 DRP コンパイラ 19
3.3 問題点の整理 21
3.4 提案手法 22
3.4.1 はじめに 22
3.4.2 DRP によるデータ入出力と演算処理のオーバーラップ処理法 . . . 23
3.4.3 データ駆動型方式によるコンテキストの削減方法 33

第 4 章　実験・評価 **40**
4.1 はじめに 40

4.2 実装環境 40
4.3 実験 40
4.3.1 実験環境 41
4.3.2 実験プログラムと実装 42
4.4 評価 46
4.4.1 入出力と演算のオーバーラップ 46
4.4.2 実行速度の比較 52
4.4.3 コンテキスト数の比較 54
4.5 考察 54
4.6 まとめ 55

第 5 章 まとめ **56**
5.1 本研究の目的 56
5.2 提案手法 56
5.3 実験結果 57
5.4 本研究の貢献 57
5.5 今後の課題 58

図 目 次

1.1 リコンフィギャラブルデバイスによるコプロセッサ 2
1.2 プログラムのループ部のハードウェア化 3
1.3 リコンフィギャラブルデバイスと CPU の実行法の比較 3
1.4 入力と処理のオーバーラップによる処理法 5
1.5 DRP のコンフィギュレーションデータ生成法 5

2.1 Vertex-2 の構造 [9] 8
2.2 Vertex-2 CLB の構造 [9] 9
2.3 Virtex-2 スライスの構成 [9] 9
2.4 全体再構成と動的部分再構成 10
2.5 FPGA のマルチコンテキスト方式 [14] 11
2.6 動的リコンフィギャラブルプロセッサのマルチコンテキスト方式 12

3.1 マルチプロセス処理 16
3.2 Tile の構造 [2] 17
3.3 Proccessing Element(PE) の構成 [14] 18
3.4 DRP-1 の構造 [2] 20
3.5 DRP コンパイラのコンパイルフロー 21
3.6 提案手法のフェーズ 22
3.7 Tile 単位の処理法 23
3.8 全 PT へ同時に入力 (パターン 1) 24
3.9 各 PT へタイミングをずらして入力 (パターン 2) 24
3.10 いくつかの PT へ同時に入力 (パターン 3) 25
3.11 2PT の構成 27
3.12 2PT の DRP-1 上の構成 27
3.13 3PT の構成 28
3.14 3PT の DRP-1 上の構成 28
3.15 4PT の構成 29
3.16 4PT の DRP-1 上の構成 30
3.17 5PT の構成 31
3.18 5PT の DRP-1 上の構成 31
3.19 DRP におけるデータ駆動型の処理法 34

3.20 コンテキスト削減アルゴリズムの概要 34
3.21 コンテキスト削減アルゴリズム 37
3.22 Priority 関数の例 37
3.23 CreateContext 関数のアルゴリズム 39

4.1 実験環境 42
4.2 実験環境の概要 43
4.3 従来手法のデータの取り込み 44
4.4 提案手法のデータの取り込み 45
4.5 従来手法のデータ入力 47
4.6 従来手法の演算処理 47
4.7 従来手法のデータ出力 48
4.8 2PT の動作 48
4.9 3PT の動作 49
4.10 4PT の動作 50
4.11 クロック削減率 51
4.12 ラプラシアン・フィルタの実行時間の比較 53
4.13 ラプラシアン・フィルタにおける従来手法と提案手法の動作周波数の比較 . 53

表 目 次

3.1 DRP-1 の詳細 16
3.2 1PT の Tile 数 26
3.3 2PT のマルチプロセスによる VMEM 使用箇所 27
3.4 3PT のマルチプロセスによる VMEM 使用箇所 29
3.5 4PT のマルチプロセスによる VMEM 使用箇所 30
3.6 5PT のマルチプロセスによる VMEM 使用箇所 32
3.7 演算器コスト 35
3.8 演算の実行優先度 36
3.9 各 PT 数の 1PT における PE 数 38

4.1 実装環境 41
4.2 動作検証ツール 41
4.3 実験で利用したソフトウェア 44
4.4 従来手法の実行クロックの内訳 46
4.5 入出力と演算のオーバーラップによるクロックの削減率 51
4.6 実行時間の比較 52
4.7 コンテキスト数の比較 54

第1章　序論

1.1　研究背景

リコンフィギャラブルデバイスは、プログラムで任意の回路を構成することができるデバイスである。また、1 度回路を構成したら回路を変更することが不可能な Application Specific Integrated Circuit(ASIC) とは異なり、何度でも回路を再構成することが可能である。現在のリコンフィギャラブルデバイスは、VLSI の集積率向上により大規模化しているので、様々なアプリケーションをハードウェア化することが可能となっている。この様な背景からリコンフィギャラブルデバイスは、画像・音声処理 [1][2]、暗号処理 [3][4][5]、組み込みシステム [6] など様々な分野で注目されている。

再構成可能という特長を活かしてリコンフィギャラブルデバイスを図 1.1 のようなコプロセッサとすれば、CPU でボトルネックとなる処理をリコンフィギャラブルデバイスでハードウェア化するによって高速化できる可能性がある。プログラム中の for 命令や while 命令のようなループ命令は、CPU による処理ではボトルネックとなる。ループ命令がプログラム中でたかだか数行であっても、ループカウントが莫大な回数であれば、その部分の実行時間がプログラム全体の実行時間の大部分を占めてしまう。従って、図 1.2 のようにループの部分をリコンフィギャラブルデバイスで処理することで高速化できる可能性がある。図 1.3 で示すようにループ部分を CPU で処理する場合、ループ内の命令文をループカウント数だけ実行するので、実行時間が増加してしまう。一方、リコンフィギャラブルデバイス上にループカウント数分の演算器を並列に構成した場合、動作周波数は CPU に比べて非常に低いが、1 クロックで実行することができるので、CPU よりも高速に実行できる可能性がある。従って、ボトルネックとなるループ内の命令文は、リコンフィギャラブルデバイスでハードウェア化して処理を行い、逐次的な命令文は、CPU で処理を行うハイブリッドなシステムを構成したのならば、実行時間の短縮を望むことができる。

しかし、リコンフィギャラブルデバイスで並列処理を行うには、問題点がある。リコンフィギャラブルデバイスの大規模化により多数の演算回路をハードウェア化することが可能となったが、リコンフィギャラブルデバイスのデータ入出力する I/O ピン数は、それにともなって増加しているわけではない。この結果、リコンフィギャラブルデバイス上に多数の演算器を実現したものの、データの入出力を 1 クロックで行うことができず、全ての演算器を利用することができない。この問題を解決するために入力データを内部のメモリに一度保存してからデータを全演算器に転送することが考えられる。しかし、入出力データが多い処理をこの手法で実行した場合、入出力時間がボトルネックとなる可能性があ

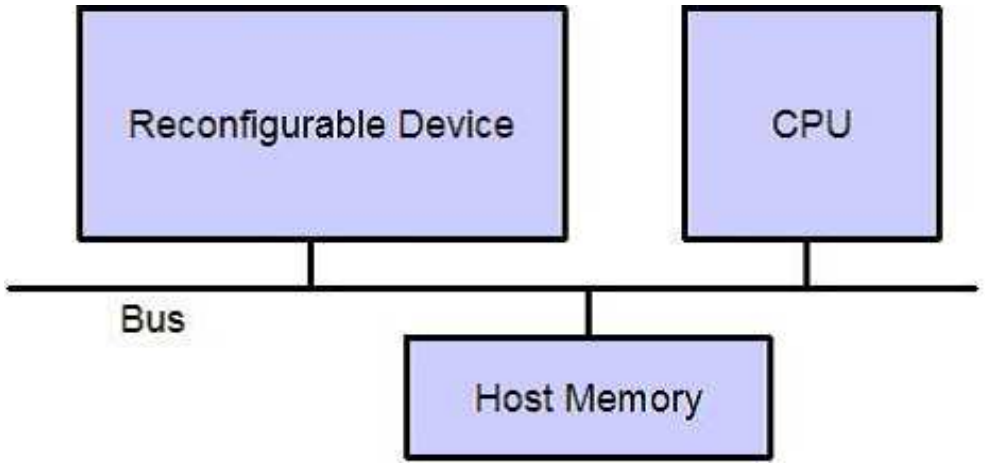

図 1.1: リコンフィギャラブルデバイスによるコプロセッサ

る。従って、リコンフィギャラブルデバイスで並列処理を行うには、データ入出力時間を考慮する必要がある。

本研究では、近年登場したリコンフィギャラブルデバイスであるマルチコンテキスト型動的リコンフィギャラブルプロセッサを対象デバイスとする。Feild Programmable Gate Array(FPGA) のような従来のリコンフィギャラブルデバイスは、再構成するために回路情報であるコンフィギュレーションデータをホストメモリからデバイス内にダウンロードしなければならないという問題を抱えている。そして、近年のリコンフィギャラブルデバイスは、大規模化に伴ってコンフィギュレーションデータのサイズも増大しているため、ダウンロード時間が非常に長い。本研究の対象デバイスは、デバイス内にコンフィギュレーションデータを複数保存することが可能なので、再構成を瞬時に行うことができる。また、従来のリコンフィギャラブルデバイスのコンフィギュレーションデータよりデータサイズが小さいので、ホストメモリからのダウンロード時間も短縮することできる。[7] しかし、デバイス内に保存可能なコンフィギュレーションデータ数は限られているので、最近の動画処理などのようにタスクの処理が複雑化しているアプリケーションで要求されるコンフィギュレーションデータ数を DRP に全て保存できないことが考えられる。[8] この結果、実行中にコンフィギュレーションデータのダウンロードが発生し、本来の能力を発揮できない可能性がある。また、内部のコンフィギュレーションデータ数を削減することで、複数のアプリケーションのコンフィギュレーションデータを保存することが可能となる。従って、マルチコンテキスト型動的リコンフィギャラブルデバイスにおいて内部に保存するコンフィギュレーションデータ数を削減することは、大きな課題である。

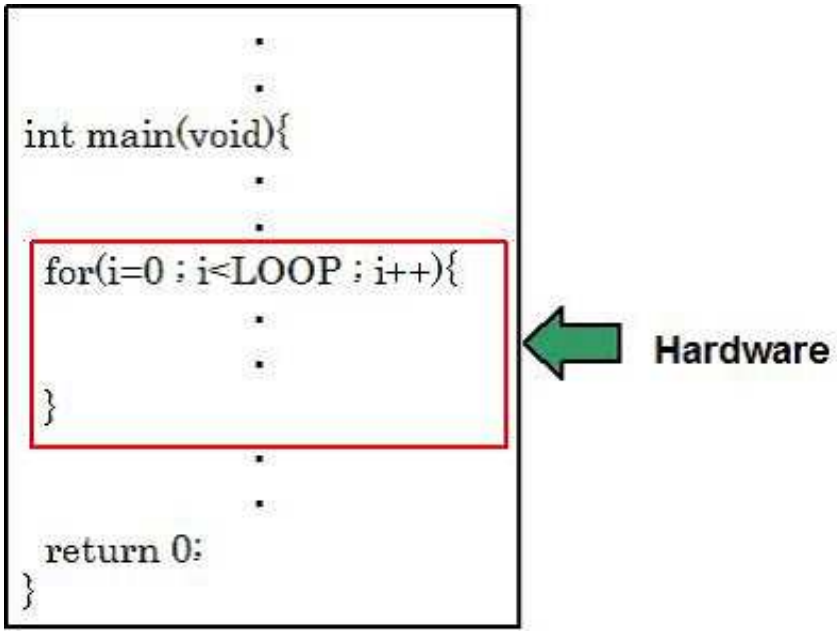

図 1.2: プログラムのループ部のハードウェア化

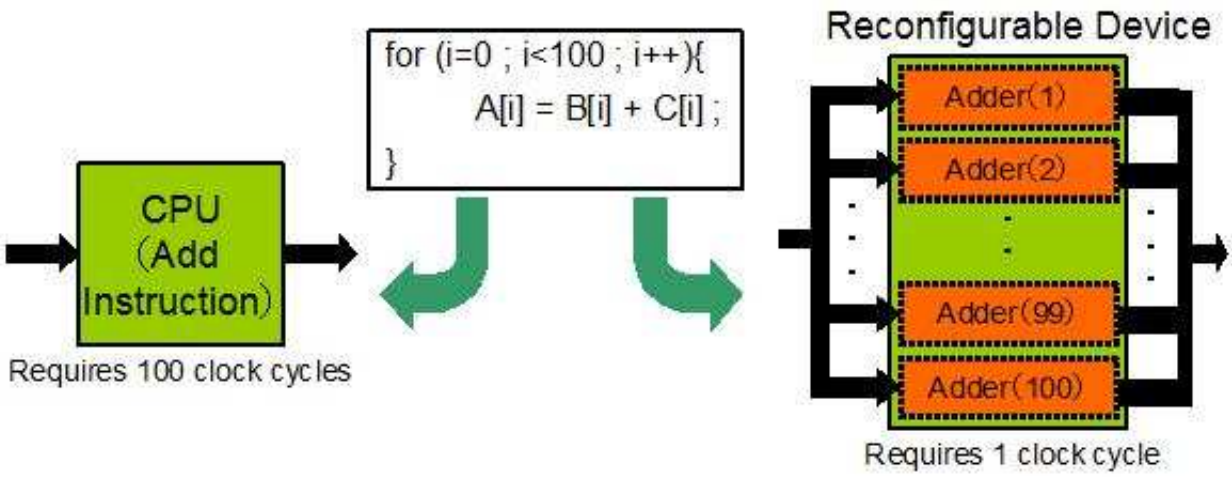

図 1.3: リコンフィギャラブルデバイスと CPU の実行法の比較

1.2 研究目的

本研究では、以下の目的を対象として 2 つの手法を提案する。

- 動的リコンフィギャラブルプロセッサにおけるデータ入出力を考慮したループ処理
- コンテキスト型リコンフィギャラブルプロセッサ内に保存するコンフィギュレーションデータ数の削減

そして、対象とするコンテキスト型動的リコンフィギャラブルプロセッサは、NEC エレクトロニクス社の Dynamically Reconfigurable Proseccer (DRP) とする。DRP は、再構成を行う基本セル Processing Element(PE) が 8 ビットの Arithmetic and Logic Unit(ALU) で構成されているため、ストリーム処理のようなデジタル演算処理に適している。PE は、コンテキストと呼ばれるコンフィギュレーションデータを 16 個保存できるため再構成を瞬時に実行できる。これにより、その面積以上の回路を実現する回路の仮想化が可能である。また、DRP には Tile と呼ばれるコアが複数あり、各 Tile は、他の Tile に依存することなく独自に再構成を行う動的部分再構成が可能である。回路を生成するためのプログラミング言語は、DRP 専用に拡張した C 言語であるので、Register Transfer Level(RTL) 言語に比べて簡単にアルゴリズムを回路にすることができる。

1 つ目の提案は、各 Tile にループ内の処理を行う回路を生成し、各 Tile のデータ入力のタイミングをずらすことでデータ入出力と演算処理をオーバーラップさせる手法である。これにより、全実行クロック数を削減する。DRP の入力・出力ビット幅は各 64 ビットと非常に小さいので、入出力データ数が多い処理やループカウント数が大きい場合、全実行時間においてデータ入出力時間を無視することができない。DRP は、Tile 単位で動的部分再構成が可能なので、回路の仮想化を利用する場合や 1 つのコンフィギュレーションデータで複数クロックで処理を行う場合では、図 1.4 のように Tile1 が処理を行っている間に Tile2 のデータ入力を行うことでデータ入出力を隠蔽することが可能である。

2 つ目の提案は、演算処理を行う Tile に 1 クロックで入力可能なデータからデータ駆動型のコンテキストを生成することである。DRP 専用のコンパイラは、図 1.5 のように高級言語コンパイラから生成される Control Data-Flow Graph(CDFG) を分割することで複数のコンテキストを生成する。従って、CDFG を細かくするれば、コンテキスト数は増加するが、クリティカルパスを減少させて動作周波数を向上させることができる。逆に、CDFG を粗く分割すれば、コンテキスト数を削減できるが、動作周波数が低下してしまう。データ駆動方式は、演算に必要なデータがそろうと演算を開始することが可能であり、CDFG のデータ並列性を引き出す可能性がある。本研究では 1 クロックで入力されたデータで実行可能な演算子を CDFG 中から発見し、それらを 1 つのコンテキストでハードウェア化する。ここの結果、粗く CDFG を分割するため、DRP 内に保存するコンテキスト数を削減することができる。コンテキストを削減することで動作周波数の低下が考えられるが、1 つ目の提案で実行クロック数を削減できるので、高速に処理を行うことが可能だと考えられる。

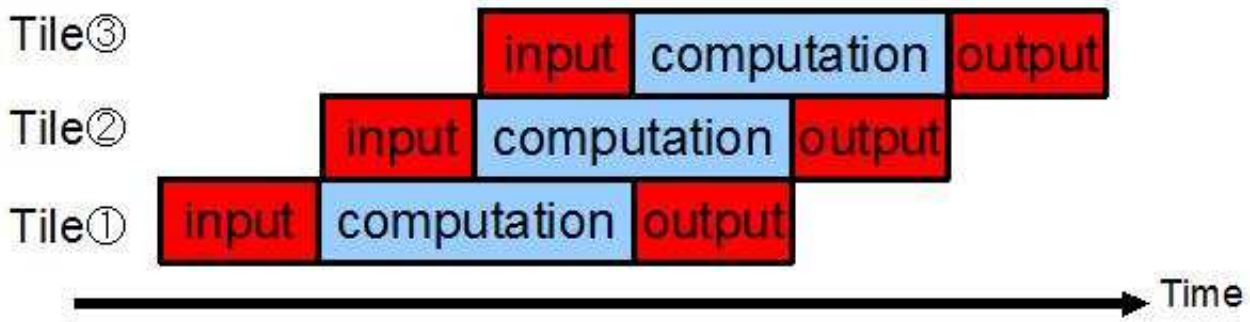

図 1.4: 入力と処理のオーバーラップによる処理法

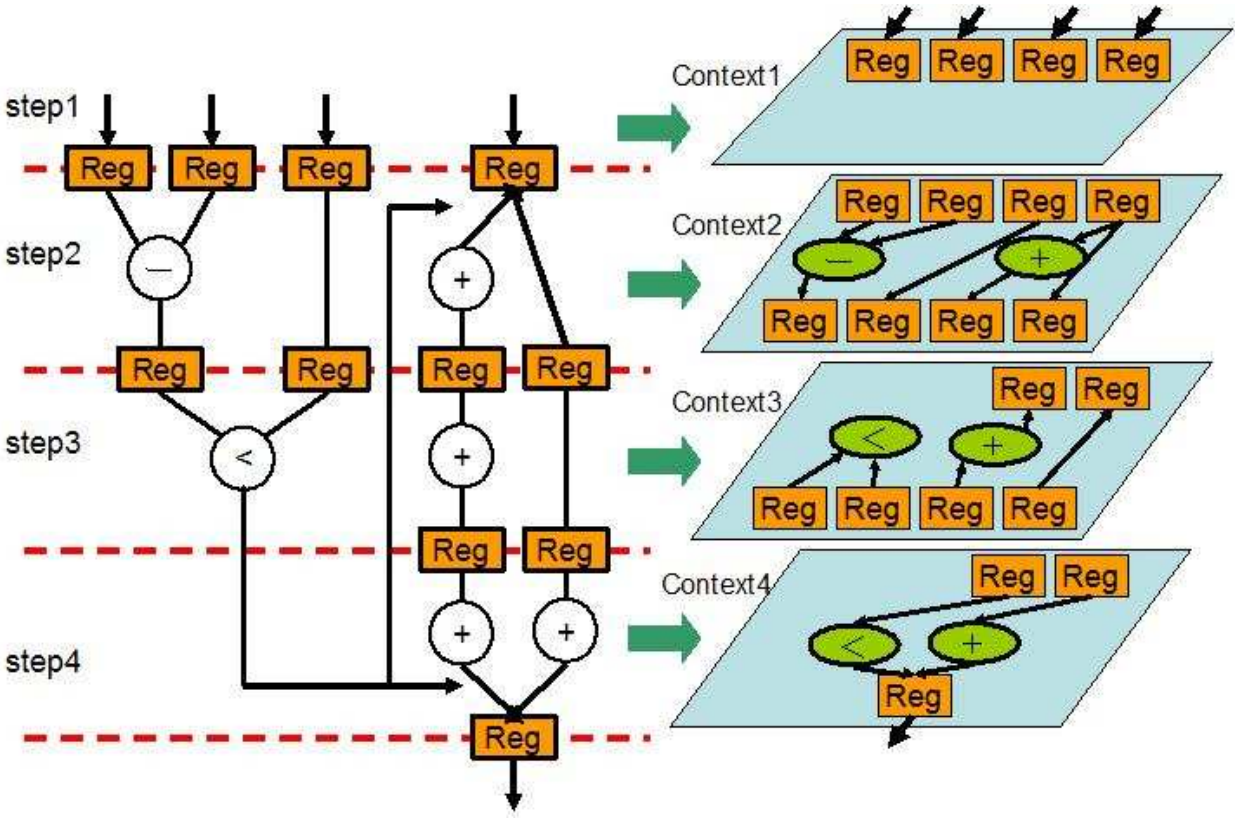

図 1.5: DRP のコンフィギュレーションデータ生成法

1.3　本論文の構成

本論文は、全5章で構成されており各章は以下のようになっている。

- 第1章：

  研究背景と本研究の目的について述べる。

- 第2章：

  既存のリコンフィギャラブルデバイスの種類と特徴を述べる。また、リコンフィギャラブルデバイスを用いた関連研究についても述べる。

- 第3章：

  本研究で対象デバイスとなるDRPの詳細について述べ、DRPでのデータ入出力と演算のオーバーラップ処理法とDRP内に保存するコンフィギュレーションデータ削減法の2つの手法について述べる。

- 第4章：

  第3章で述べた提案手法の実装法・実験・評価・考察を述べる。

- 第5章：

  本研究のまとめを述べる。

第2章　リコンフィギャラブルデバイスの利用

2.1　はじめに

現在、Feild Programmable Gate Array (FPGA) や動的リコンフィギャラブルプロセッサなど様々なリコンフィギャラブルデバイスが既存している。そして、それらを利点を活かして、様々なアプリケーションや CPU ではボトルネックとなる処理をリコンフィギャラブルデバイスで高速処理する研究が多くある。

本章では、FPGA の構成・特徴と問題点を述べ、System-onChip(SoC) で FPGA を利用する際の問題点を解決するために開発された動的リコンフィギャラブルプロセッサについて述べる。リコンフィギャラブルデバイスを用いた並列処理やダウンロード時間の短縮に関する関連研究についても述べる。

2.2　リコンフィギャラブルデバイス

2.2.1　Feild Programmable Gate Array (FPGA)

1970 年代にフューズ方式でプログラム可能なリコンフィギャラブルデバイスが初めて開発された。1980 年代には、SRAM ベースの Look-Up Table(LUT) に基づく FPGA や複数の AND-OR アレイ構造を組み合わせた Complex Programmable Logic Device(CPLD) が登場しことで、リコンフィギャラブルデバイスは、注目されるようになった。その後の 1990 年代後半に FPGA の集積度や性能が向上したため、現在のリコンフィギャラブルデバイスは、FPGA が中心となっている。

FPGA は、LUT を利用して論理回路を構成するため、ビットレベルで細かな論理回路を生成することができる。従って、演算回路や制御回路など様々な種類の論理回路を無駄なく生成することが可能である。図 2.1 は、代表的な FPGA である Xilinx 社 Virtex-2 の構成を示している。FPGA は、再構成可能な基本セル Configurable Logic Block (CLB) がアレイ状に並べられており、これらをどのように構成するのかをプログラムで明記することで目的の論理回路を生成することが可能である。CLB は、図 2.2 で示したようにスイッチマトリックスと 4 つのスライス構成されている。スイッチマトリックスは、CLB 間の配線接続を司っているので、ある CLB のデータ入出力は、この部分の構成で決定される。

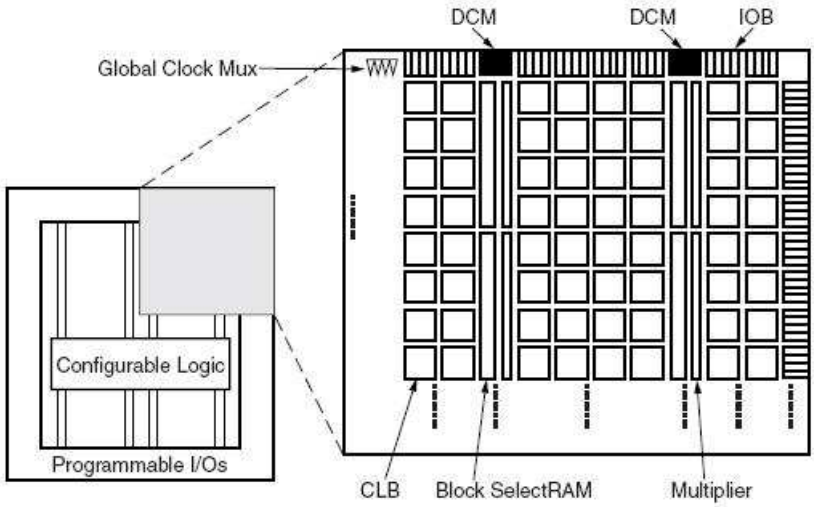

図 2.1: Vertex-2 の構造 [9]

図 2.3 は、CLB 中のスライスの構成を示しており、2 つの 4 入力 LUT や 2 つのレジスタなどで構成されている。スライス中の LUT に真理値表を保持することで、意図する論理回路を実現する。

2.2.2 FPGA の問題点

FPGA には、いくつかの問題がある。1.1 でも述べたように論理回路を生成ためのダウンロード時間が非常に長いということである。FPGA の再構成は、スイッチマトリックスとスライス中の LUT の SRAM を全て書き換えるので、リコンフィギャラブルデバイスが大規模になればコンフィギュレーションデータのサイズも大きくなり、ダウンロード時間が長くなる。この問題に対して、ベンダは、1 ビット毎コンフィギュレーションデータを FPGA にダウンロードする JTAG ダウンロード方式以外に [10] のように複数ビット毎ダウンロードする新しい方式を開発している。また、2.2.3 で後述する動的部分再構成が可能な FPGA を発表している。しかし、FPGA での動的部分再構成を設計・実行は、設計者に相応の知識と技術がないと実行することが困難である。[11] ダウンロード時間を短縮する研究としては、Zhiyuan ら [12] のようにコンフィギュレーションデータを圧縮する方法や Krishna ら [13] のように論理的にコンフィギュレーションデータを解析しデータサイズを最小化する方法など様々な手法で研究が行われている。

他の問題点として、メディア処理などで利用される演算器を FPGA で実現した場合、Digital Signal Processor(DSP) のような特定のアプリケーションに特化したハードウェアと比べて数倍の面積を要求してしまうことである。そして、面積を広く要求するために、動作速度も低下してしまう。[14]

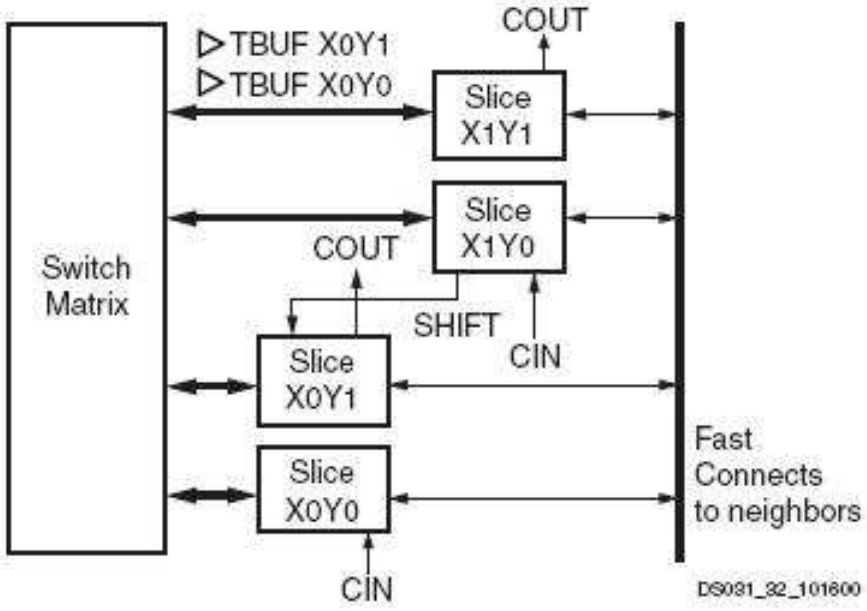

図 2.2: Vertex-2 CLB の構造 [9]

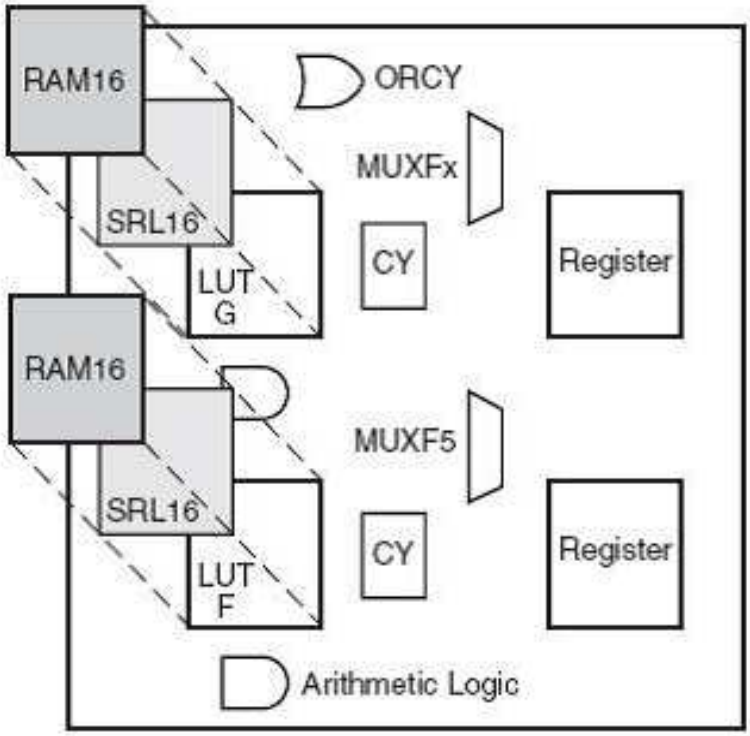

図 2.3: Virtex-2 スライスの構成 [9]

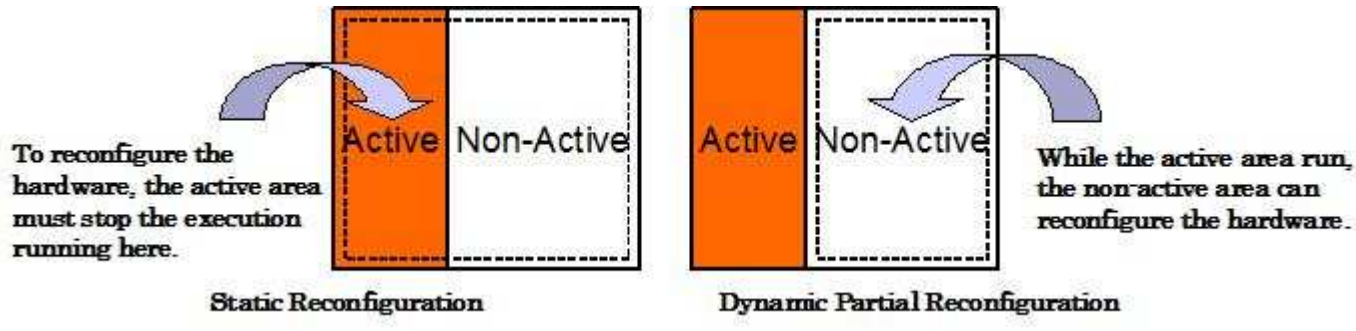

図 2.4: 全体再構成と動的部分再構成

2.2.3 動的部分再構成

動的部分再構成とは、デバイスで稼動していない部分のみを部分的に再構成する手法である。図 2.4 は、従来の再構成手法である全体再構成と動的部分再構成の仕組みを示している。図 2.4(a) の全体再構成は、常にデバイス全体を再構成するので、未稼働の回路を再構成するためには、稼動中の回路を停止した後に再構成を行わなければならない。一方、図 2.4(b) の動的部分再構成は、稼動中の回路を停止することなく未稼働部分のみを再構成可能である。従って、コンフィギュレーションデータは、未稼働部分のみになるため、データサイズを削減することができ、ダウンロード時間を減少させることができる。また、稼動中の回路のバックグラウンドで再構成を行うので、ダウンロード時間を隠蔽することも可能である。

しかし、動的部分再構成を行うためには、様々なことを考慮する必要がある。部分再構成が可能なサイズは、デバイス毎に指定されているので、設計段階で回路の配置を考慮しなければならない。また、どのタイミングでどの回路を生成するのかを決定するスケジューリングについても考慮しなければならい。回路間のデータの受け渡しについても考慮する必要がある。このような背景から FPGA で、動的部分再構成を実現することは、非常に困難である。

2.2.4 マルチコンテキスト方式

ダウンロード時間を短縮する方法で、動的部分再構成以外の方法としてマルチコンテキスト方式がある。マルチコンテキスト方式とは、図 2.5 で示すように複数のコンフィギュレーションデータを保持可能なメモリを FPGA の傍に搭載することで再構成時間を大幅に短縮する手法である。一般的に、この方式でコンテキストとは、メモリ内に保持されているコンフィギュレーションデータのことであり、FPGA を再構成することをコンテキストスイッチと呼ぶ。

マルチプレクサで生成したいコンテキストを指定することで、高速に回路を FPGA 上

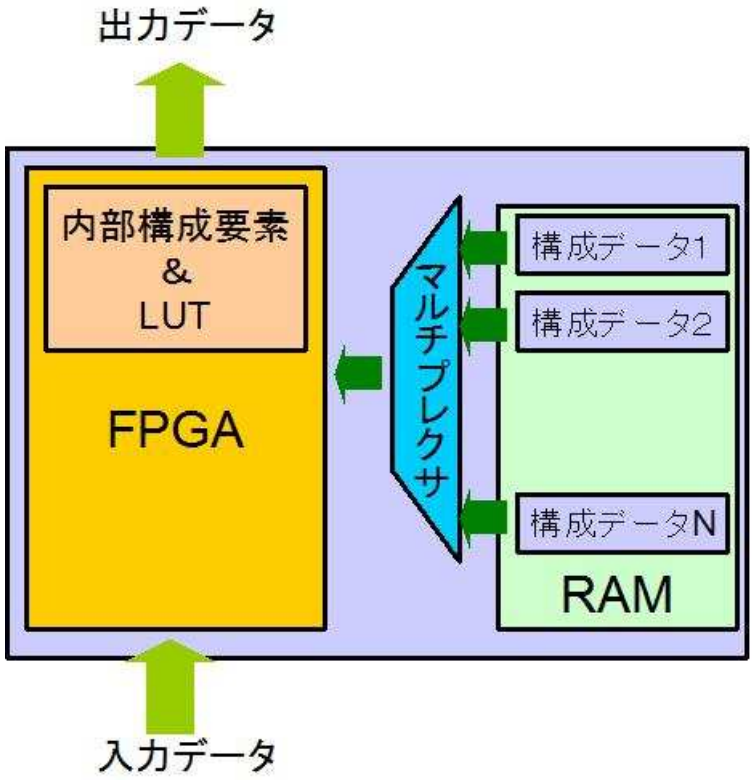

図 2.5: FPGA のマルチコンテキスト方式 [14]

に実現することが可能である。従って、リアルタイム処理でもダウンロード時間がボトルネックとなることがない。また、あるアプリケーションをハードウェア化したときに FPGA の面積より大きくなり 2 つのコンテキストが要求されても、2 つ目のコンテキストを高速にコンテテキストスイッチが可能である。これにより実行時間で再構成がボトルネックとならず、回路の仮想化ができる。

2.2.5 動的リコンフィギャラブルプロセッサ

近年、SoC におけるアプリケーション専用のハードウェア部の代替として注目されている動的リコンフィギャラブルプロセッサが登場している。具体的には、NEC エレクトロニクス社の Dynamically Reconfigurable Processor(DRP)[15] やアイピーフレックス社の DAPDNA[16] などがそれに当たる。最近の様々な製品には、SoC が搭載されていおり、MPEG や JPEG などのコード圧縮や復号、暗号化されたコード、エラー修正用コードなど強力な演算能力を要求する処理を専用のハードウェアで高速処理している。LUT ベースの FPGA を SoC に搭載した場合、各処理に必要とされる演算器を実現すると専用ハードウェアの数倍の面積を要求してしまう。これにより、動作速度が低下してしまう。この解決する

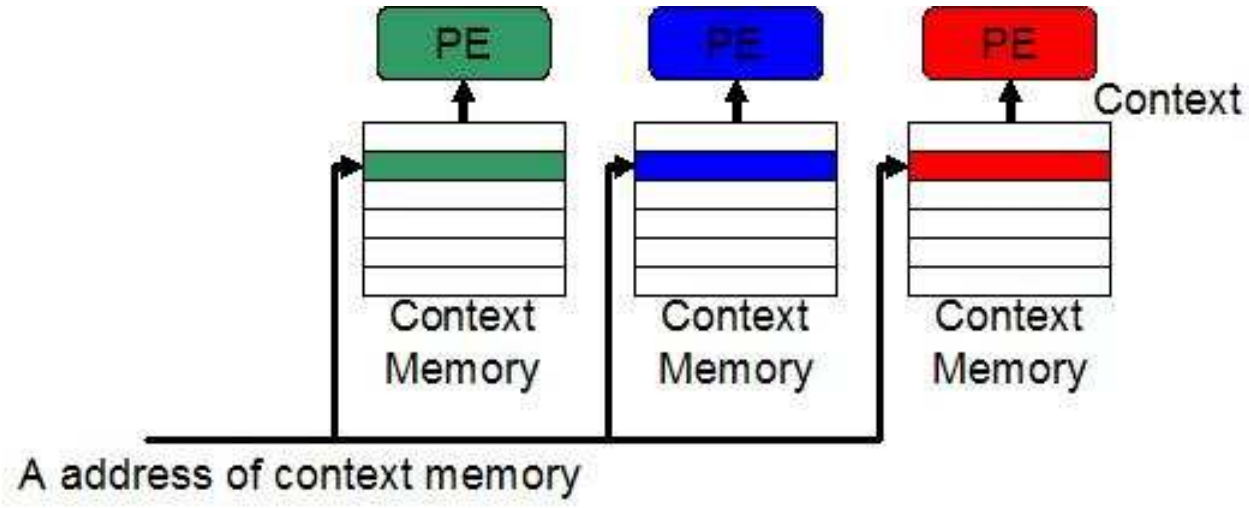

図 2.6: 動的リコンフィギャラブルプロセッサのマルチコンテキスト方式

ために動的リコンフィギャラブルプロセッサは、基本セルとなる Processing Element(PE) を Arithmetic and Logic Unit(ALU) ベースで構成している。基本セルが ALU ベースであるため、LUT ベースの FPGA のように回路の細かい部分を設計することは不可能だが、演算ビットが 4 ビットー32 ビットであるため、対象アプリケーションのビット幅とうまく適合した場合、高速でかつ面積効率を向上させることができる。[14]

再構成時間の問題に関しては、2.2.4 で述べたマルチコンテキスト方式を採用している。FPGA のマルチコンテキスト方式とは異なり、動的リコンフィギャラブルプロセッサのマルチコンテキスト方式は、各 PE に複数のコンテキストを保持している。AUL ベースの PE は、ローカルに加算命令やシフト命令などを持っているので、動的リコンフィギャラブルプロセッサのコンテキストの内容は、FPGA のような回路構成ではなく、各 PE が実行する命令である。図 2.6 のようコンテキストが保存されているコンテキストメモリに対して生成したいコンテキストのアドレスを指定することで回路を実現する。

2.3 関連研究

Narasimhan ら [17] は、FPGA を用いたループネストの並列処理に関する研究を行った。彼らは、ループ内の並列性の解析をステートメントレベルで行い、実行中の再構成回数の最小化と回路を再利用するための部分再構成でダウンロード時間を最小限に抑えた。また、FPGA の面積を最大限に利用して並列度を上げることも行った。その結果、livermore Loops や Perfect Benchmark などベンチマークのループ処理を CPU よりも高速に処理することが可能であることを示した。しかし、実験に使用した FPGA は、非常に小規模なものであるため、彼らのアルゴリズムにはデータ入出力について考慮されていない。従って、現在の大規模なリコンフィギャラブルデバイスに彼らのアルゴリズムを適応しても面

積を最大に利用して高並列度の回路構成を導き出すことができるが、データ入出力がボトルネックとなる可能性がある。井口 [18] は、C レベルのプログラミング言語から並列性を導き出し、FPGA で並列処理を行うための手法について議論した。高級言語から CPU 処理ではボトルネックとなるループを検出後、ループ内のステートメントのデータ依存、面積効率や実行時間を解析し、FPGA で最適な並列処理の実行方法を提案した。結果として、従来のコンパイラより FPGA 上に多く演算器を構成することができた。しかしながら、Narasimhan らの研究と同様にデータ入出力のボトルネックが発生してしまう可能性が考えられる。また、これら 2 つの研究は、FPGA を利用しているので、再構成時のダウンロード時間が発生してしまう。このため、ダウンロード時間が長ければ、全実行時間でこの時間が支配的となりボトルネックとなる可能性がある。

鈴木ら [1] は、ストリームアプリケーションなどでボトルネックとなる処理を DRP で実行した。彼らは、エッジ近傍合成機能付き α ブレンダ・RC6・JPEG の離散コサイン変換 (DCT)・MP3 デコーダの変形逆離散コサイン変換 (IMDCT)・離散ウェブレット変換 (DWT)・Viterbe デコーダを実装した。処理に応じて適切な設計を行うことで CPU や DSP より高スループットを実現することを示した。しかし、エッジ近傍合成機能付き α ブレンダは、連続的にデータを出力していないので、次のデータ出力までに数クロック掛かってしまう。また、DCT、IMDCT は、1 度付属メモリに演算に必要な入力データを全て取り込んだ後に演算を行い、演算結果を再度付属メモリに戻してから出力を行っているため、データ入出力のボトルネックが生じている。天野ら [19] は、これまでの DRP の研究からデータ入出力時間が全実行時間の 30ー50%近くになり、データ入出力がボトルネックになっていることを指摘した。この問題を解決するために、単一チップ内で処理を行う場合、付属メモリの制御と演算を行うコアの部分の制御を分離することでデータ入出力時間を隠蔽することを提案した。また、複数の DRP をデータ入出力制御機構と接続するシステムも提案した。複数の DRP で処理を行う場合、2 つのパターンが考えられる。各タスクを DRP に割り当て、データを DRP 間で転送する手法と各 DRP で全タスクを処理する手法である。後者の手法では、デバイス内に全コンテキストを保存することができず、実行中に残りのコンテキストをダウンロードする必要がある可能性がある。彼らのシミュレーションの結果、現在の DRP では、前者の手法の方がアドヴァンスがあると報告されている。鈴木ら [8] は、各 PE に物理コンテキストと論理コンテキストの対応表を搭載し、同じコンテキストをもつ物理コンテキストを複数の論理コンテキストで共有する分散対応表方式を提案した。この提案により、DRP の PE 内に保存するコンテキスト数を最大で 40%削減することに成功した。しかし、彼らの手法は、PE 内に保存するコンテキスト間の共通性を利用しているため、共通性が少ない場合には効果的ではない。

2.4 まとめ

動的リコンフィギャラブルプロセッサは、マルチコンテキスト方式を用いているためチップ内にコンテキストを保存でき、再構成を瞬時に行うことが可能である。基本セル

が、ALU ベースであるので、LUT ベースで構成されている FPGA と比べて演算速度が速い。このことから本研究では、動的リコンフィギャラブルプロセッサである NEC エレクトロニクス社の DRP を利用する。また、DRP は、動的部分再構成が容易に実現できることも大きなメリットである。

これまでの DRP の研究で、CPU ではボトルネックとなる処理を高速化に処理することが可能であることが分かっている。しかし、天野ら [19] が述べたようにデータ入出力時間がボトルネックとなる処理がある。彼らの研究では、この問題に対しての複数の DRP でデータ入出力と演算処理をオーバーラップさせる提案を行っているが、1 つの DRP における手法については述べられていない。本研究では、動的部分再構成を利用してデータ入出力と演算処理をオーバーラップさせる研究を行う。

DRP の実装では、高速処理を行うために動作周波数とコンテキスト数のバランスがポイントとなる。1.2 でも述べたように動作周波数が向上すれば、コンテキスト数が増加し、逆にコンテキスト数を削減すれば、動作周波数は低下してしまう。これまでの DRP 研究では、コンテキストメモリ内のデータの共通性からコンテキスト数を削減する手法などが報告されているが、データ入出力と演算をオーバーラップさせることによる動作周波数とコンテキスト数のバランスについては、報告されていない。従って、本研究では、このことについても研究を行う。

第3章　動的リコンフィギャラブルプロセッサを利用した提案手法

3.1　はじめに

本研究で対象とするデバイスをマルチコンテキスト型動的リコンフィギャラブルデバイスの Dynamically Riconfigurable Processor(DRP) とする。そして、DRP によるデータ入出力と演算処理のオーバーラップ処理法とデータ駆動型方式によるコンテキスト数の削減を行う。

本章では、DRP のアーキテクチャと専用コンパイラについて説明し、本研究で提案する 2 つの手法について述べる。

3.2　Dynamically Riconfigurable Processor (DRP)

3.2.1　DRP の概要

DRP は、ストリーム処理を対象とした NEC エレクトロニクス社のマルチコンテキスト型動的リコンフィギャラブルデバイスである。DRP の基本セルの Processing Element(PE) は、ALU を基本としており、Tile と呼ばれる DRP のコアにアレイ状に配置されている。各 PE には 16 個のコンテキストを保存することができ、瞬時にコンテキストスイッチが可能である。DRP のプロトタイプである DRP-1 は、Tile を 4 * 2 の配置で 8 つ持ち、各 Tile は独自にコンテキストスイッチを行う State Transition Controller(STC) を持っているため Tile 単位で動的部分再構成ができる。また、DRP のプログラム言語である Behavioral Design Language(BDL) は、C 言語を拡張した言語なので、C 言語のアルゴリズムを容易に回路にすることができる。表 3.1 は、DRP-1 の詳細である。

動作原理は、初めに DRP コンパイラで BDL から Finite State Machine(FSM) とデータパスを生成し、STC に FSM を、PE アレイにデータパスを割り当てる。各 PE は、データパス中の演算などを実現するので、Tile 上にはデータパスの一部、又は、全体が構成されることとなる。コンテキストスイッチは、STC が FSM をトレースして各 PE のコンテキストメモリに命令ポインタを発行することで PE の命令を変化させ、Tile 上のデータパスを変化させる。

マルチコンテキストによる瞬時のコンテキストスイッチと Tile 単位による動的部分再

表 3.1: DRP-1 の詳細

クロック周波数	11-133MHz
アレイサイズ	8 タイル
PE 数	512
メモリサイズ	2Kb 2port Memory * 80, 64Kb 1port Memory * 32
PE に保存可能なコンテキスト数	16

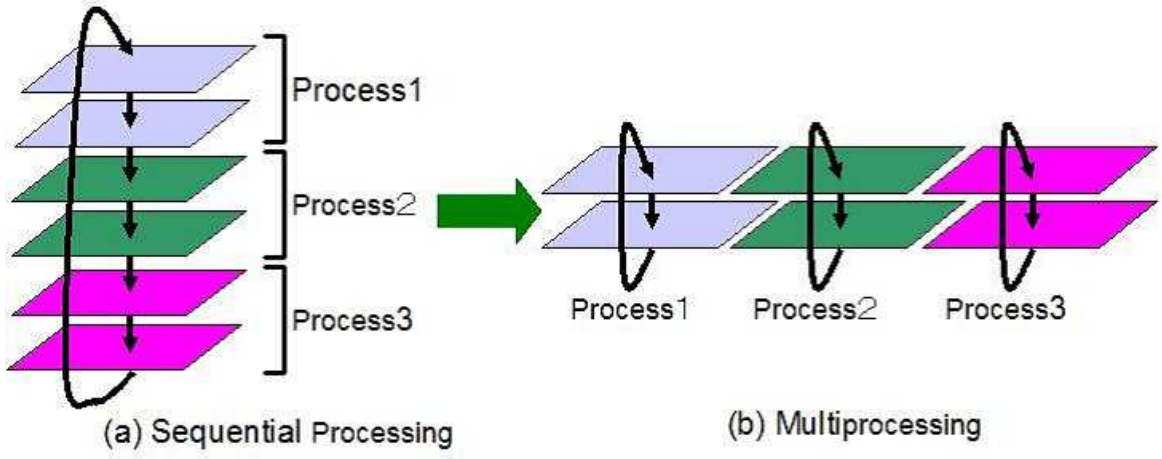

図 3.1: マルチプロセス処理

構成を利用することで、マルチプロセス処理を行うことが可能である。図 3.1 は、3 つのプロセスを実行するためのマルチプロセス処理を表している。各プロセスが 2 つのコンテキストを要求し、1 コンテキスト当たり 1 クロックで処理できると仮定する。図 3.1(a) のように全体再構成を行い逐次処理を行った場合では、データ入力から出力まで 6 クロックサイクル必要とする。従って、6 クロック毎にデータが出力される。図 3.1(b) のように Tile 毎にプロセスを分割しマルチプロセス処理を行った場合、処理開始から最初の出力まで 6 クロックサイクルかかる。しかし、その後からは 2 クロックサイクル毎にデータを出力することが可能なので、全体再構成の逐次処理よりもスループットを向上させることができる。

3.2.2 Tile アーキテクチャ

Tile の構成を図 3.2 に示す。Tile は以下の要素で構成されている。

- Processing Element (PE)

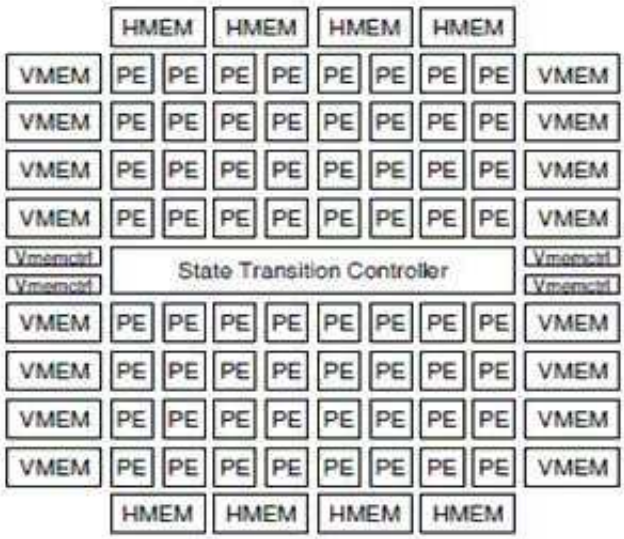

図 3.2: Tile の構造 [2]

- Vertical Memory (VMEM)
- Horizontal Memoty (HMEM)
- State Transition Controller (STC)
- VMEM Controller

基本セルとなる PE は、8 * 8 の 2 次元配列状に 64 個配置されており、その中央部に STC が配置されいる。メモリは、Tile を囲むように配置されており、左右に VMEM が各 8 個、上下に HMEM が各 4 個配置されている。VMEM Controller は、VMEM 列の中央部に配置されている。

VMEM は、8 ビット幅の深さ 256 ワードの 2-port メモリである。8 ビット以上のデータを保存する場合、又は、256 ワード以上のデータを保存する場合は、複数の VMEM を利用して実現する。例えば、32 ビットのデータを 256 ワード保存する場合は、4 つの VMEM を利用して実現する。HMEM は、8 ビット幅の深さ 8K ワードの 1-port メモリである。このメモリも VMEM 同様に 8 ビット以上のデータや 8K ワード以上のデータを保存するときは、複数の HMEM を利用して実現する。マルチプロセス処理をする場合、Tile 間のデータ転送は、VMEM を使用しなければならない制約があるので、VMEM Controller は、VMEM のデータのリード・ライト信号とアドレス指定などの制御を行う。STC は、Tile 内の全 PE と接続されており、コンテキストスイッチの実行時 PE 内のコンテキストメモリにメモリアドレスを転送する。

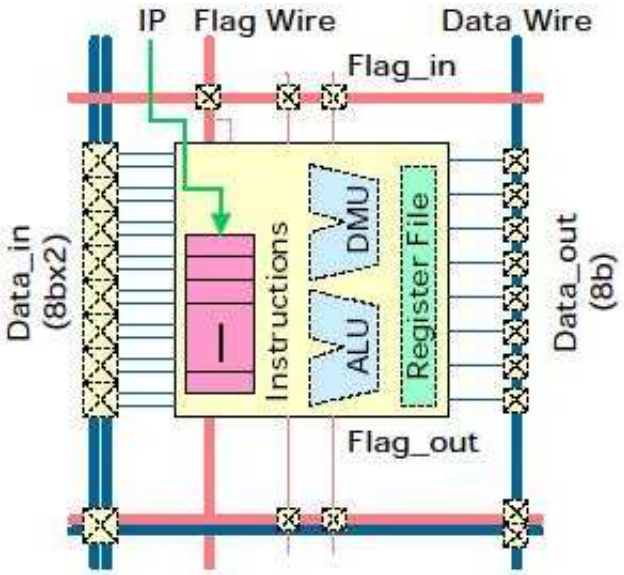

図 3.3: Proccessing Element(PE) の構成 [14]

3.2.3 Processing Element アーキテクチャ

Tile の基本セルとなる PE の構成を図 3.3 に示す。PE は以下の要素で構成されている。

- Arithmetic Logic Unit (ALU)
- Data Manipulation Unit (DMU)
- Instruction Unit (IU)
- Flip-Flop Unit (FFU)
- Register File Unit (RFU)
- Bus Selector

PE は、8 ビット演算を扱うユニットである。8 ビット以上の演算は、複数の PE を利用して実行する。ALU では、加算命令や減算命令などを行い、DMU では、シフト命令やマスク命令などを行う。FFU は、8 ビットのフリップフロップであり、RFU は、8 ビット-16 ワードのレジスタである。両方ともコンテキストスイッチが起きても記憶しているデータが変化することはない。Bus Selector は、PE 内の演算ユニット、FFU や RFU と PE を囲んでいるバスの入出力接続を指定するユニットである。IU は、16 個のコンテキストを保存することができるコンテキストメモリであり、Tile の STC から送られてきたアドレスのコンテキスト内の命令を ALU や DMU が実行する。コンテキスト内には、演算ユニットの命令の他に Bus Selector の制御命令も含まれている。

3.2.4 DRP-1 アーキテクチャ

DRP のプロトタイプである DRP-1 の構成を図 3.4 で示す。DRP-1 は、以下の要素で構成されている。

- Tile
- Center State Transition Controller (CSTC)
- Multiplier (MUL)
- PCI Controller (PCIC)
- Memory Controller (MC)
- PLL
- Configuration Read Data Selector (CRDS)
- Initialize Controller (INIT)

DRP のコアとなる Tile は、アレイ状 (4 * 2) に 8 つ、VMEM が 5 列で 80 個、HMEM が Tile 郡の上下に各 16 個ずつ配置されている。CSTC は、全 Tile のコンテキストスイッチを一括管理を行う。各 Tile は、独自に STC を持っているため他の Tile に依存することなくコンテキストスイッチを行うことが可能である。MUL は、8 つの Tile を囲むように配置されており、Tile から離れている MUL を利用することも可能である。

3.2.5 DRP コンパイラ

DRP コンパイラは、DRP 専用のコンパイラであり、大きく分けてフロントエンド合成とバックエンド合成に分かれる。図 3.5 は、DRP コンパイラのコンパイルフローである。フロントエンド合成部の動作合成は、BDL を入力として FSM とデータパスの 2 種類の Register Transfer Level(RTL) を中間コードとして出力する。動作合成は、プログラムのデータフローを分割してコンテキストを生成する。分割は、データの依存性、リソースの制約とクロック制約を満たし、かつ、並列性の最大化、処理リサイクル数の最小化を目的に実行される。[21] バックエンド合成部は、マッピング、Place & Route とオブジェクト生成で構成されており、フロントエンド合成部から出力された RTL を入力として DRP オブジェクトコードである STC コードと PE アレイコードを出力する。マッピングは、STC コードの生成、PE の命令マッピング、演算粒度フィッティング、論理の最適化などを行う。Place & Route は、マッピングから出力されたネットリストを基にして PE 命令を適切な位置に割り当て、データ転送路を決定する。与えられた PE アレイ内でクロック制約を満たす命令配置とデータ転送路を求めることが目的である。オブジェクトコード生成は、DRP にダウンロード可能なコードに変換する。

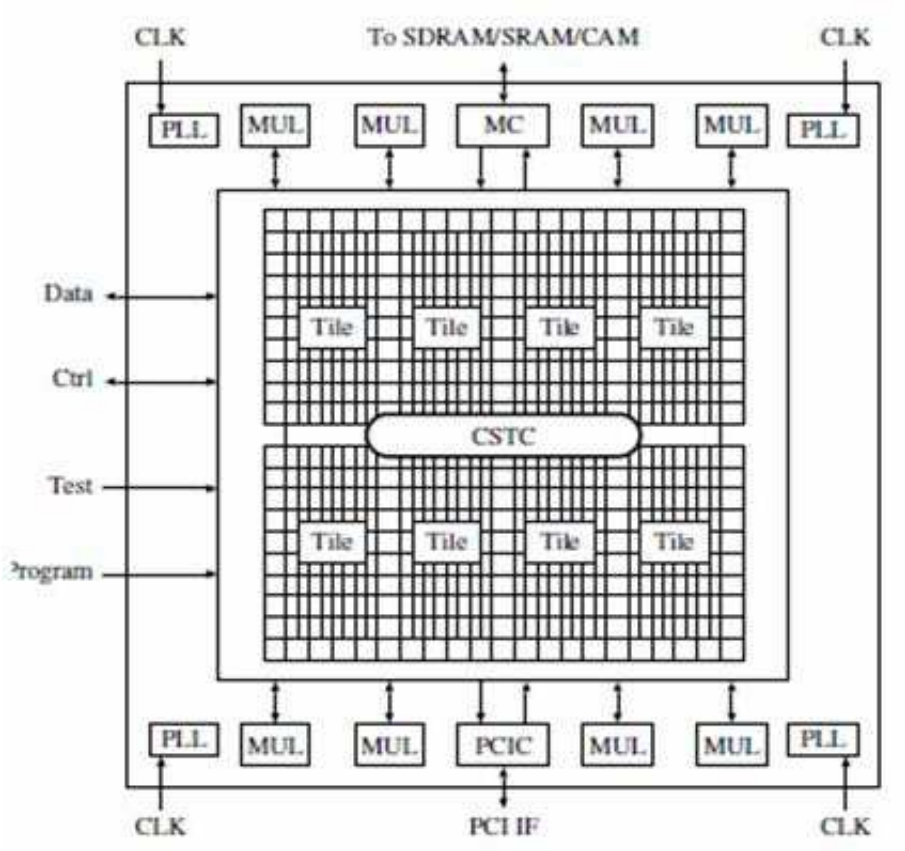

図 3.4: DRP-1 の構造 [2]

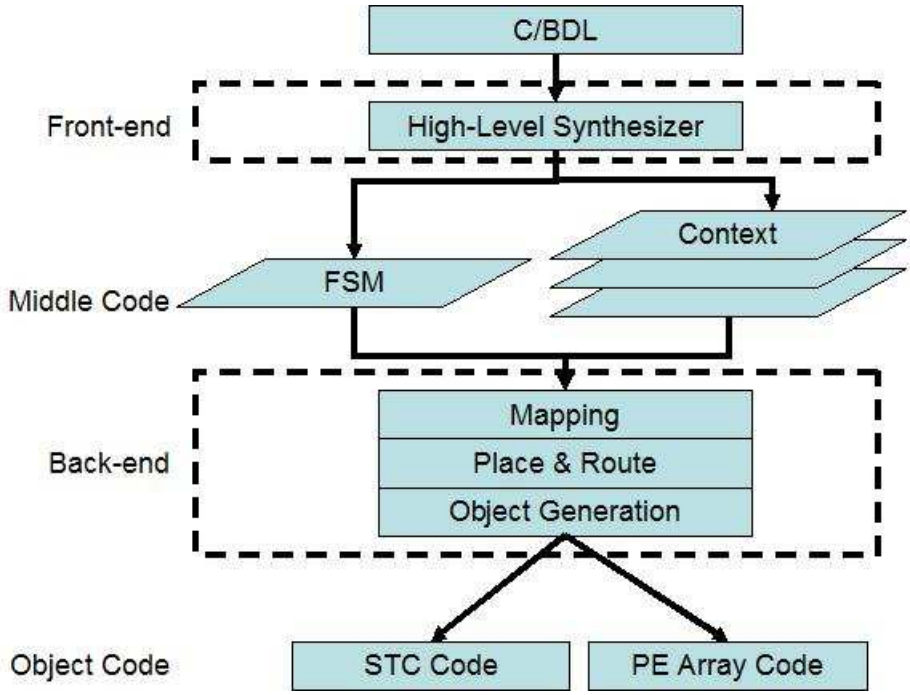

図 3.5: DRP コンパイラのコンパイルフロー

DRP コンパイラには、データフローグラフを自動的に分割する自動スケジューリングモードと分割箇所を明示的に指示できる手動スケジューリングモードがある。自動スケジューリングモードは、上記で説明した処理を行う。一方、手動スケジューリングモードは、フロントエンドのデータフローの分割を指定できるので、PE に保存するコンテキスト数を意図的に決定できる。

3.3 問題点の整理

リコンフィギャラブルデバイスで並列処理を行うには、同時に入力できるデータ数が重要である。現在のリコンフィギャラブルデバイスの大規模化により I/O ボトルネックが考えられる。デバイスの大規模化に伴って付属している I/O の数が増加しているわけではないためである。従って、リコンフィギャラブルデバイスで並列処理を行うためには、複数クロックでデータを入力しなければならない。データ出力に関しても同じことが言える。この結果、データ入出力の多い処理では、入出力の実行時間と実行クロック数がボトルネックとなる。

本研究では DRP を利用することで、瞬時にコンテキストスイッチを行うことができるので、回路の仮想化が可能となったが、DRP 内に保存できるコンテキスト数が 16 個であり、今後登場する複雑な処理を実行するにはこの数では足りなくなる可能性がある。16 個を超えるコンテキスト数を要求する処理の場合、実行途中でホストメモリからコンテキス

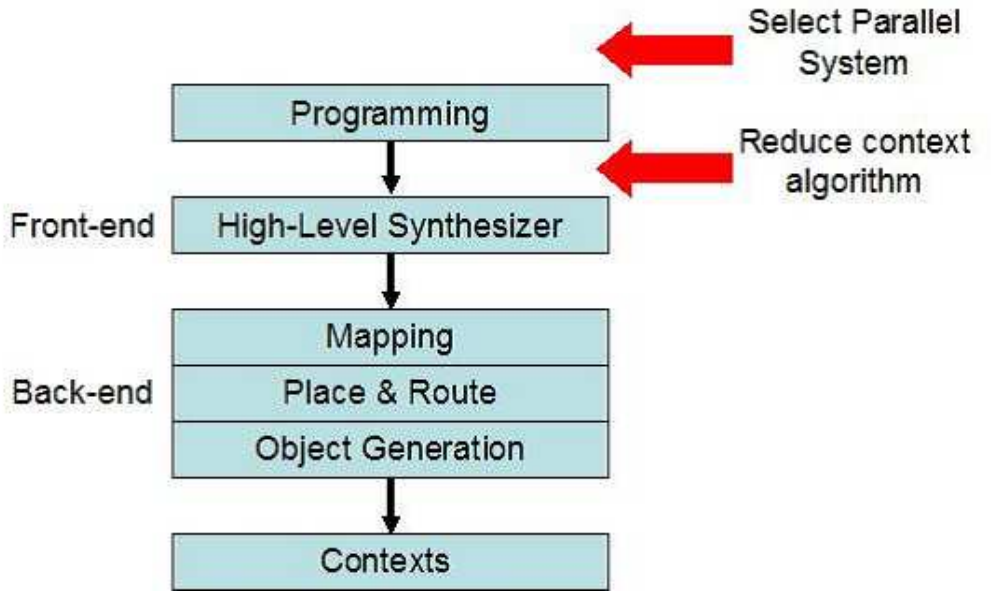

図 3.6: 提案手法のフェーズ

トをダウンロードする必要となり、この時間がボトルネックとなる。従って、処理を行うためのコンテキスト数を削減することは、DRP での課題である。

3.4 提案手法

3.4.1 はじめに

本研究は、DRP によるデータ入出力と演算処理のオーバーラップ処理法とデータ駆動型方式によるコンテキスト数の削減アルゴリズムを提案する。図 3.6 のように、DRP によるデータ入出力と演算処理のオーバーラップ処理法のシステム選択は、プログラミングを行う前の設計時に行う。DRP へのデータ入力方式により処理の方法が変わるので、設計するアプリケーションに適したオーバーラップ処理法を決定する。2 つ目の提案手法であるデータ駆動型方式によるコンテキストの削減アルゴリズムは、プログラミングからコンテキストの作成までのフェーズ中で DRP コンパイラのフロントエンド合成の前に位置づけされる。

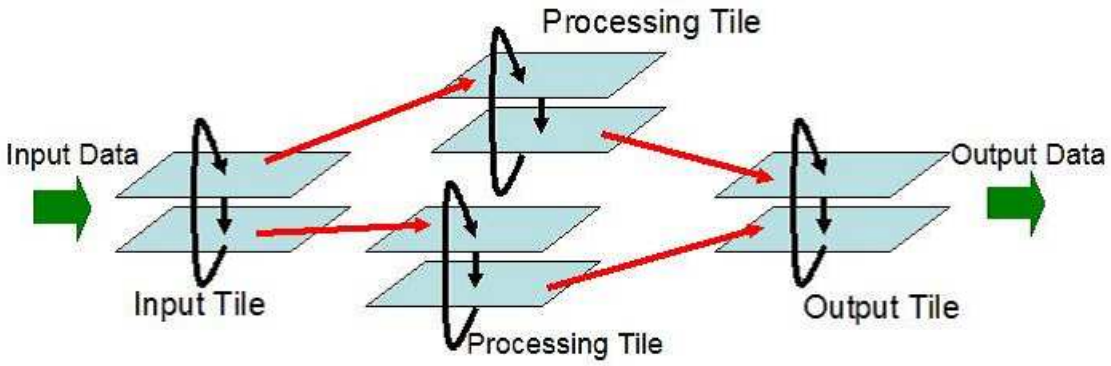

図 3.7: Tile 単位の処理法

3.4.2 DRP によるデータ入出力と演算処理のオーバーラップ処理法

はじめに

DRP-1 は、多くの PE が搭載されているので、PE 単位で並列処理を行えば、高並列度の回路を構成することが可能である。しかし、DRP-1 の入力ビット幅が 64 ビットしかないので、全入力データを VMEM や HMEM に保存してから演算を開始する手法が考えられる。しかし、入力データの保存に多くのクロック数を要してしまう。また、VMEM が 2 ポートメモリ、HMEM が 1 ポートメモリなので、並列度がメモリのポート数を越えている場合にも面積効率が低下してしまう。そこで、本研究では、図 3.7 のようにマルチプロセス機能を利用して Tile 単位で処理を行い、データ入出力と演算処理をオーバーラップさせる手法を提案する。Tile 単位で処理した場合、ある時間における処理数は非常に低いものとなるが、入力ビット幅とのバランスは、PE 単位の並列処理と比べ良くなり、面積効率を向上させることが可能である。また、動的部分再構成は、Tile 単位で行われるので、他の Tile に依存することなく独立して処理をすることができる。マルチプロセス機能の Tile 間のデータ転送が最大 64 ビットなので、提案手法も Tile 間のデータ転送を 64 ビットで行う。オーバーラップで処理する場合、ホストメモリから入力したデータを各 Tile に分配しなければならない。また、各 Tile からの演算結果をまとめて DRP-1 の外に出力しなければならい。従って、DRP-1 上の 8 つある Tile の内 2 つの Tile は、データの入出力を管理するための Tile となり、残りの 6Tile が演算処理するための Tile となる。図 3.7 は、簡易化したシステムモデルを示している。Input Tile が、各 Tile へ入力データを分配し、Output Tile が各 Tile から出力された演算結果を出力するための Tile である。Processing Tile(PT) は、演算処理するための Tile である。

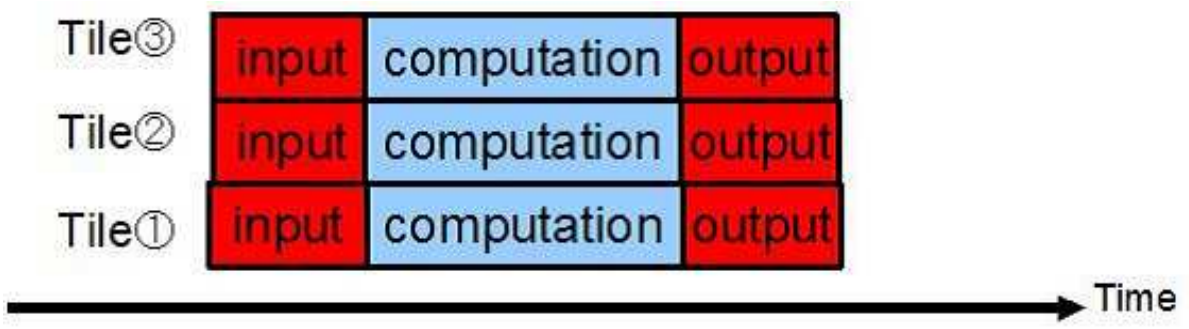

図 3.8: 全 PT へ同時に入力 (パターン 1)

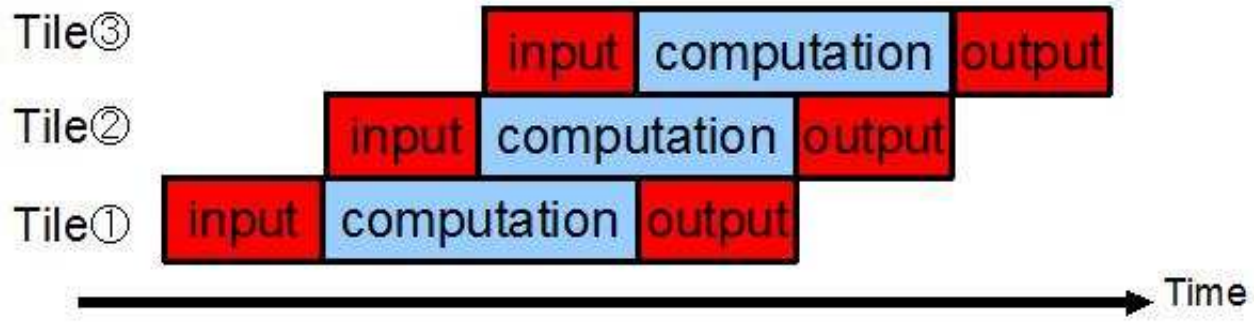

図 3.9: 各 PT へタイミングをずらして入力 (パターン 2)

Processing Tile 数の算出

DRP-1 の実行方法は、演算処理で要求するのデータ性質によって 3 パターン考えられる。1 つ目は、図 3.8 のように各 PT へ同時に入力するパターンである。(パターン 1) この場合、各 PT の処理は、出力のタイミングが同時なので、1 クロックで出力可能なデータ数を考慮する必要がある。出力データが 1 クロックで出力不可能な数とき、出力データのストールが発生してしまう。2 つ目は、各 PT の入力データが異なる場合、図 3.9 のように入力のタイミングがずれてしまうパターンである。(パターン 2) この場合、データを同時に出力しないので、DRP で実現可能な PT 数の回路を利用可能である。3 つ目のパターンとして、全 PT のデータを入力できないが、数 PT 分のデータを 1 クロックで入力可能なときは、パターン 1 と同様に出力データ数を考慮しなければならない。(パターン 3)

パターン 1 とパターン 3 のために 1 クロックで出力可能なデータ数から PT 数を算出しなければならない。DRP-1 のデータ出力ビットは、64 ビットなので、出力データビットから 1 クロックで同時に出力可能なデータの数は、式 (3.1) で算出することができる。

$$output\_data = \left\lfloor \frac{drp\_output\_bit}{output\_bit} \right\rfloor \tag{3.1}$$

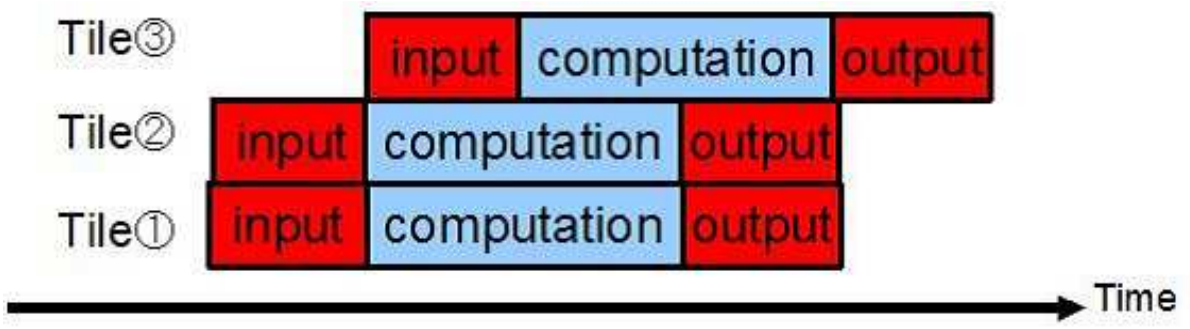

図 3.10: いくつかの PT へ同時に入力 (パターン 3)

- output_data：1 クロックで出力可能なデータの数
- drp_output_bit：データ出力ビット数 (64)[1]
- output_bit：PT から出力されるデータのビット数

PT の Tile 数が 6 つなので、DRP で実現可能な PT 数は、6PT が可能と考えられるが、マルチプロセル処理による制約のため 6PT を実現することはできない。マルチプロセス処理のデータ転送では、以下の 3 つの制約を満たさなければならない。

1. 異なる処理を行う Tile 間のデータ転送に VMEM を利用しなければならない。
2. VMEM は、Tile 間を 1 対 1 でしか接続することができない。
3. 基本的に VMEM の左側が転送データの入力、右側が出力を行う。

式 (3.2) は、VMEM の制約を考慮した PT 数を求める式である。

$$parallelism = \frac{\frac{num\_VMEM}{M}}{N} \tag{3.2}$$

- parallelism：実装可能な PT 数
- num_VMEM：VMEM の数 (80)
- M：64 ビットのデータを転送するために必要な VMEM の数 (8)
- N：入力 Tile、出力 Tile との接続数 (2)

本研究のシステムモデルで Tile 間のデータ転送は、64 ビットで行っているため、8 つの VMEM を利用することでデータ転送が可能となる。DRP-1 上には 80 個の VMEM が搭載されているので、データ転送の VMEM の集合が 10 個あると考えられる。そして、各 PT は、入力 Tile と出力 Tile とのみ接続されているので、1Tile につき 2 つの VMEM 集合を

[1]() は、実際に DRP-1 上に配置されている数

表 3.2: 1PT の Tile 数

PT 数	1PT で利用可能な Tile
2	3
3	2
4	1
5	1

利用することとなる。従って、式 (3.2) より DRP-1 で実現できる最大 PT 数は 5PT である。逆に、6PT を実現するためには、96 個の VMEM を搭載しなければならない。

次に、各 PT で利用できる Tile 数を求めなければならない。各 PT の Tile 数は、式 (3.3) で求めることができる。

$$available\_tile = \left\lfloor \frac{num\_tile}{num\_output\_data} \right\rfloor \tag{3.3}$$

- available_tile : 1PT の Tile 数
- num_tile : PT で利用する Tile 数 (6)
- output_data : 1 クロックで出力可能なデータ数

表 3.2 は、式 (3.3) から求められる各 PT の Tile の数を示している。2PT は、PT のための Tile を全て利用しているが、他の PT 数では、全て利用していない。1PT のみ異なる Tile 数にした場合、面積の制約によりコンテキスト数が異なる可能性がある。そして、1PT のみ実行クロックが異なり、システム全体の同期が取れなくなる。従って、全 PT は、同 Tile 数にしなければならない。

システムモデル

3.4.2 で求めた PT 数と各 PT 数の Tile 数を基にシステムモデルと DRP-1 上での構成を図 3.11ー3.18 に示す。図中の矢印は、データの流れを示している。表 3.3ー3.6 は、各 PT 数のマルチプロセス処理で利用される VMEM の位置を示している。

2PT の場合、1PT につき 3Tile 利用することができるので、図 3.11 のようにルート 1(Tile2、Tile4、Tile6) とルート 2(Tile3、Tile5、Tile7) の 2 ルートで構成する。Tile1 は、入力データを各ルートに分配を行い、Tile8 が各ルートからの演算結果を出力する。図 3.12 は、DRP-1 への実装とデータの流れを表している。Tile 間のデータ転送のためにルート 1 の入力側が Tile2 の左側の VMEM、出力側が Tile7 の右側の VMEM を利用している。ルート 2 は、入力側が Tile3 の左側の VMEM、出力側が Tile7 の右側を利用している。4、5PT では、演算処理を実行する Tile が 4 つなので、入出力の Tile が 4 つとなる。この場合は、マルチプロセス機能の 3 つ目の制約に沿うよう Tile を入力 Tile と出力 Tile に振り分ける。

図 3.11: 2PT の構成

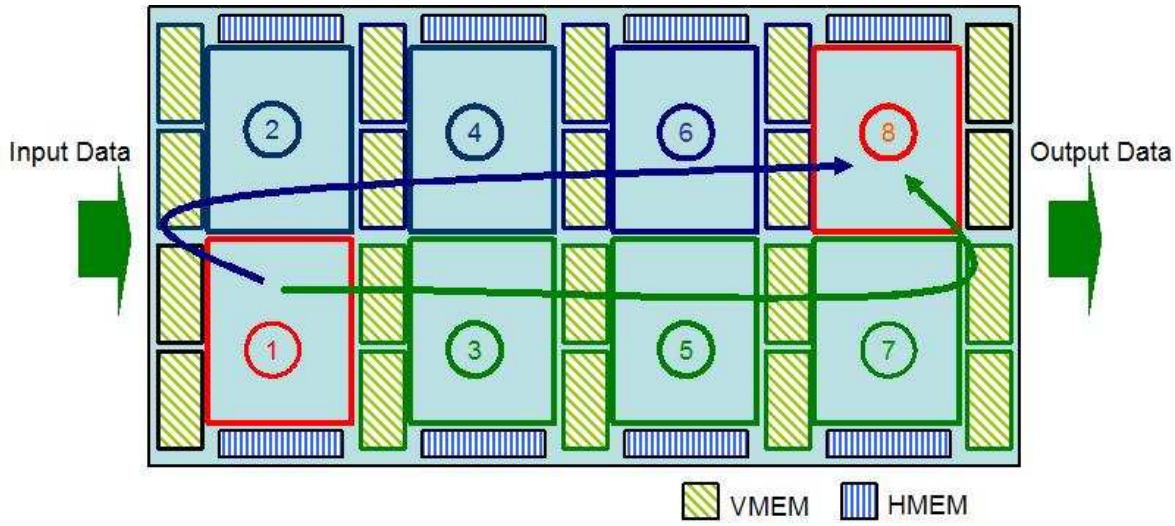

図 3.12: 2PT の DRP-1 上の構成

表 3.3: 2PT のマルチプロセスによる VMEM 使用箇所

ルート	入出力	VMEM 使用箇所
1(Tile2,4,6)	入力	Tile2 の左
	出力	Tile6 の右
2(Tile3,5,7)	入力	Tile3 の左
	出力	Tile7 の右

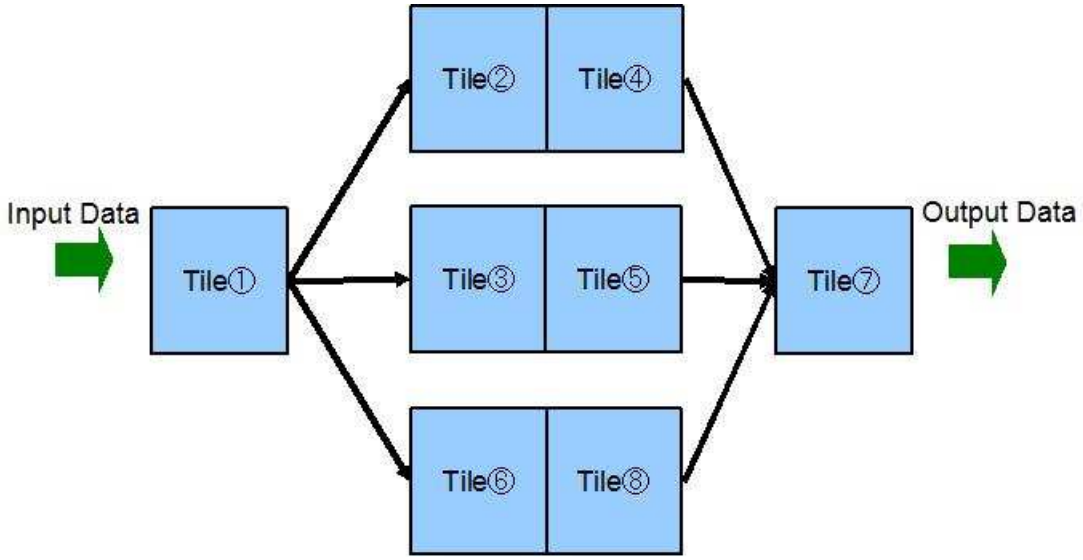

図 3.13: 3PT の構成

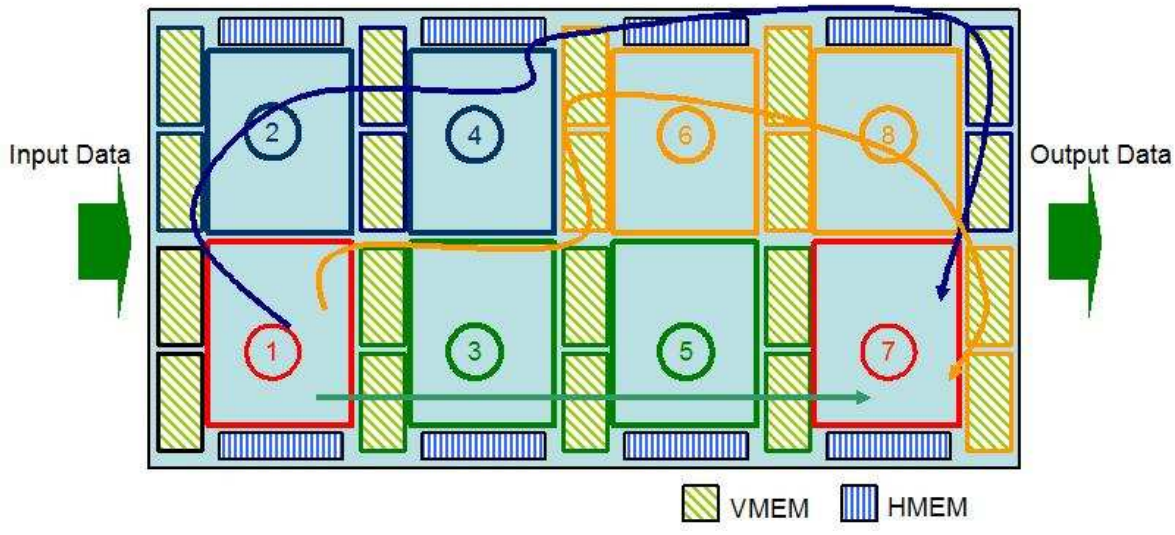

図 3.14: 3PT の DRP-1 上の構成

表 3.4: 3PT のマルチプロセスによる VMEM 使用箇所

ルート	入出力	VMEM 使用箇所
1(Tile2,4)	入力	Tile2 の左
	出力	Tile8 の右
2(Tile3,5)	入力	Tile3 の左
	出力	Tile5 の右
3(Tile6,8)	入力	Tile6 の左
	出力	Tile7 の右

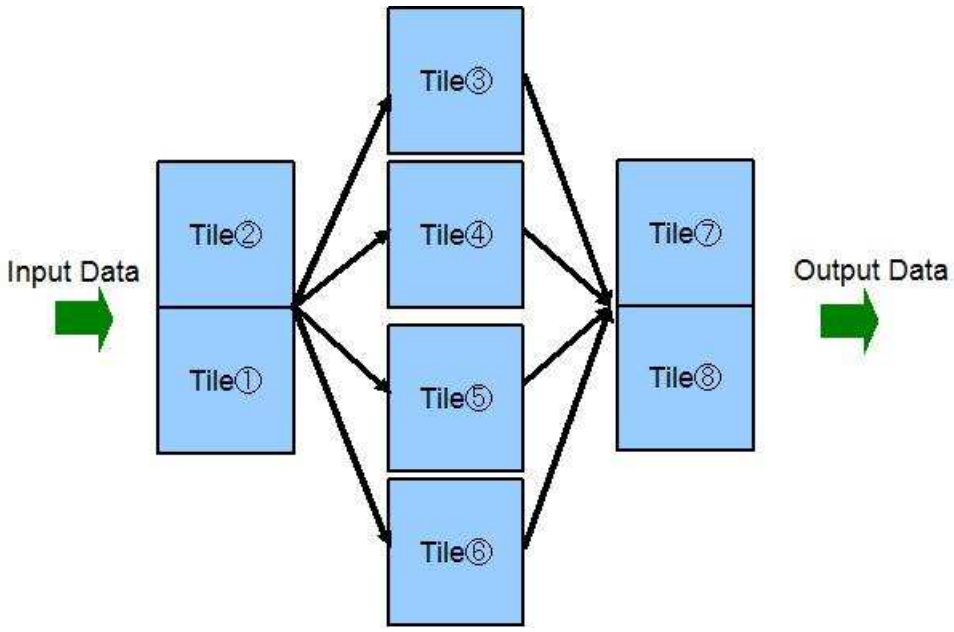

図 3.15: 4PT の構成

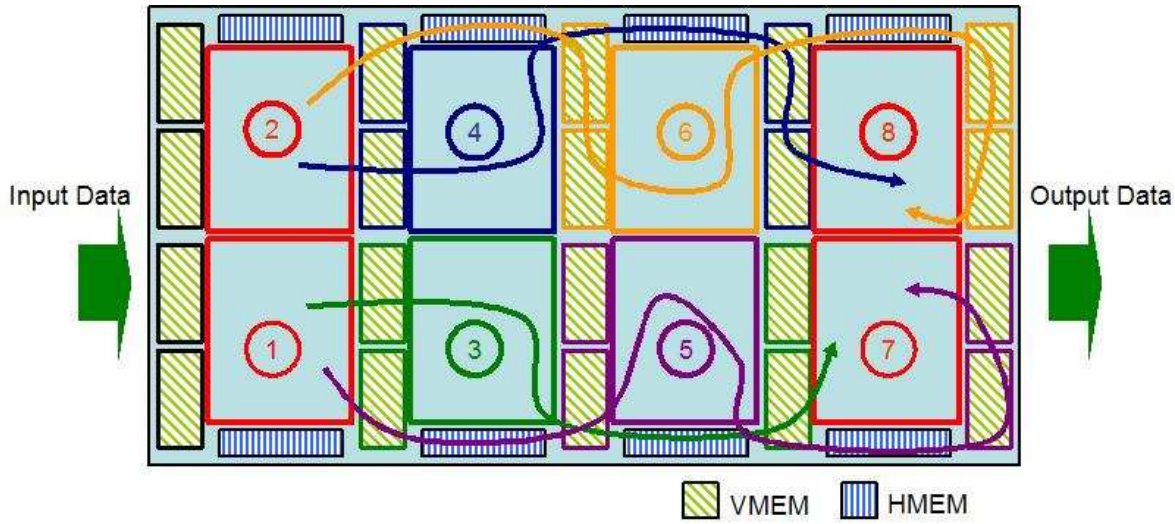

図 3.16: 4PT の DRP-1 上の構成

表 3.5: 4PT のマルチプロセスによる VMEM 使用箇所

ルート	入出力	VMEM 使用箇所
1(Tile3)	入力	Tile3 の左
	出力	Tile5 の右
2(Tile4)	入力	Tile4 の左
	出力	Tile6 の右
3(Tile5)	入力	Tile5 の左
	出力	Tile7 の右
4(Tile6)	入力	Tile6 の左
	出力	Tile8 の右

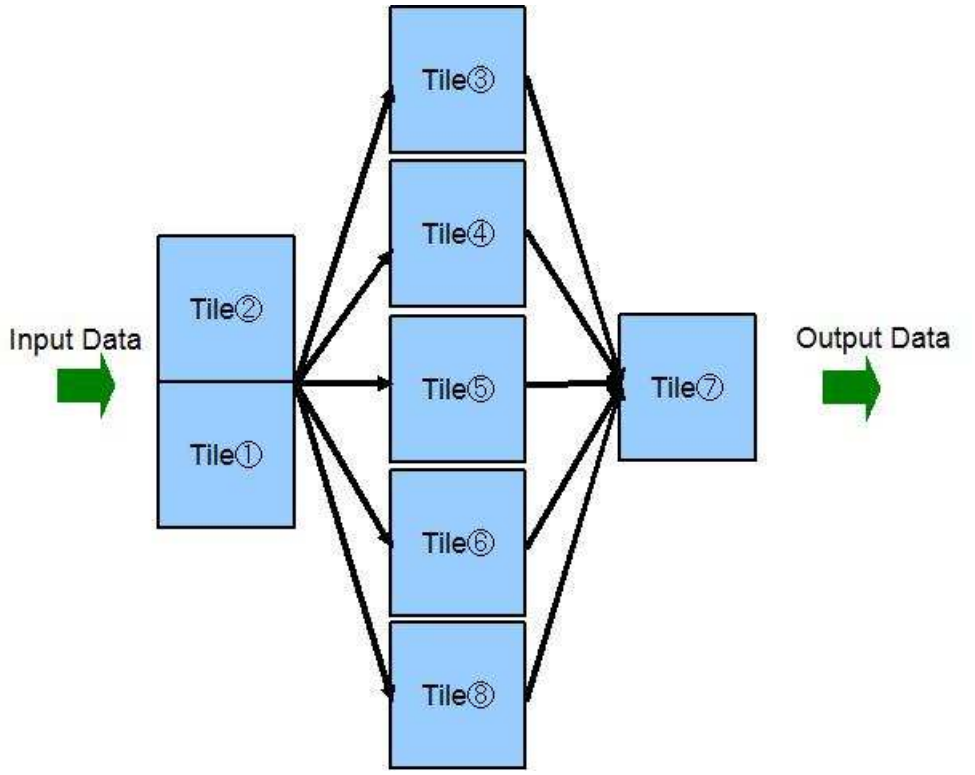

図 3.17: 5PT の構成

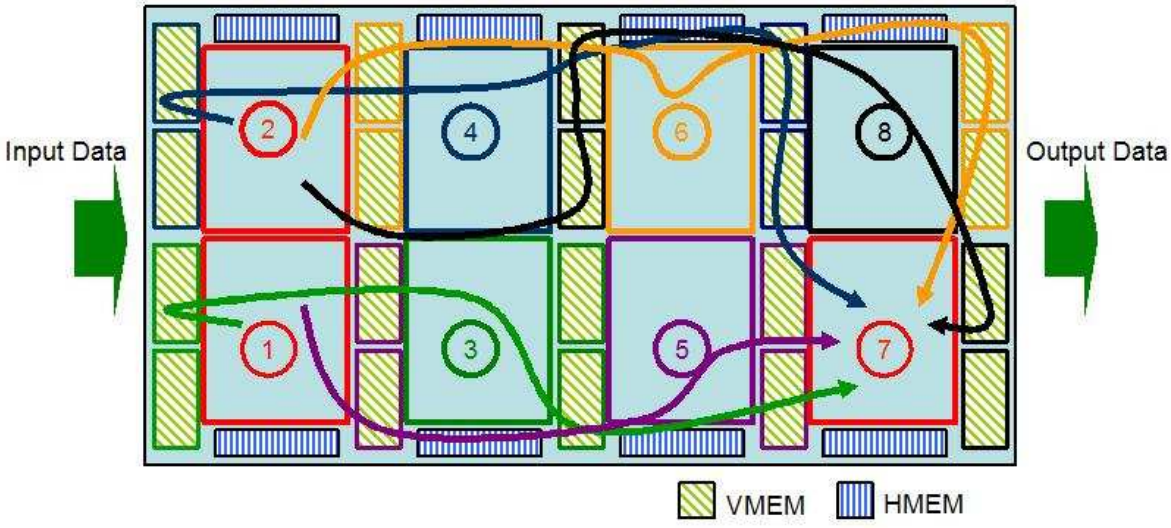

図 3.18: 5PT の DRP-1 上の構成

表 3.6: 5PT のマルチプロセスによる VMEM 使用箇所

ルート	入出力	VMEM 使用箇所
1(Tile3)	入力	Tile1 の左
	出力	Tile3 の右
2(Tile4)	入力	Tile2 の左
	出力	Tile6 の右
3(Tile5)	入力	Tile3 の左
	出力	Tile5 の右
4(Tile6)	入力	Tile4 の左
	出力	Tile8 の右
5(Tile8)	入力	Tile6 の左
	出力	Tile7 の右

3.4.3 データ駆動型方式によるコンテキストの削減方法

アルゴリズムの概要

データ駆動型のコンテキストを生成してDRPのPE内に保存するコンテキスト数を削減する。図3.19は、DRPにおけるデータ駆動型の簡易図である。図中のデータフローグラフを提案アルゴリズムで解析し、図中の右側に示したコンテキストを生成す津。DRPでデータ駆動型のコンテキストを生成するために考慮する点は、以下の3点である。

- 1クロックで1PTに入力可能なデータ数
- 1コンテキストのPE数
- 演算器の実行クロックと要求するコンテキスト数

1クロックで転送されてくるデータビットが64ビットなので、入力されるデータ数は、非常に少ない。従って、処理を行うために必要な入力データ数を1クロックで入力できない場合は、図中のdata5とdata6のように新しいコンテキストで入力しなければならない。1PTのPEは限りがあるので、実現する回路が要求するPE数が1PT当たりのPEを超えてしまった場合には、新しいコンテキストを生成しなければならない。演算器によっては、1コンテキストで複数クロック要求するもの、2クロックで2コンテキストを要求するものがあるので、この点も考慮しなければならない。

図3.20は、コンテキスト削減のアルゴリズムである。初めに、入力データを入力して、1PTに1クロックで入力可能なデータ数を算出する。次のフェーズでData Flow Graph(DFG)と演算優先度を入力してDFG中の演算子の実行順序を決定する。最後のフェーズでは演算器のコストを入力して、PE数、1クロックで入力可能なデータ数などを基にしてコンテキストを生成する。

演算器コストと実行優先度

アルゴリズムの入力となる面積コストを表3.7に示す。コストは、DRPコンパイラの手動スケジューリングモードで各演算器を生成し、構成結果のレポートから得た。PE中のALUに加算命令と減算命令があるので、加算器と減算器は、1コンテキストに実装でき1クロックで実行可能である。また、各命令は、8ビット演算可能なので、32ビット演算の場合4つのPEを利用している。乗算器は、DRP-1上にある8つ乗算DSPを利用する。DRPの制約上、乗算DSPを利用するには、乗算DSPの入力と演算にに各1クロック必要であるので、乗算DSPの出力までに2クロック必要となる。更に、2クロックは、各1コンテキストでなければならず2コンテキストを要求する。通常、DRPで定数との乗算は、乗算器を利用せず加算器などで構成するが、定数の値によってPEの使用数が変化するので、定数との乗算も乗算DSPを利用する。除算器は、2のべき乗の除算ではDMUのシフト命令を利用し、他はPEとDMUで独自で生成した。独自に作成した除算器は、1

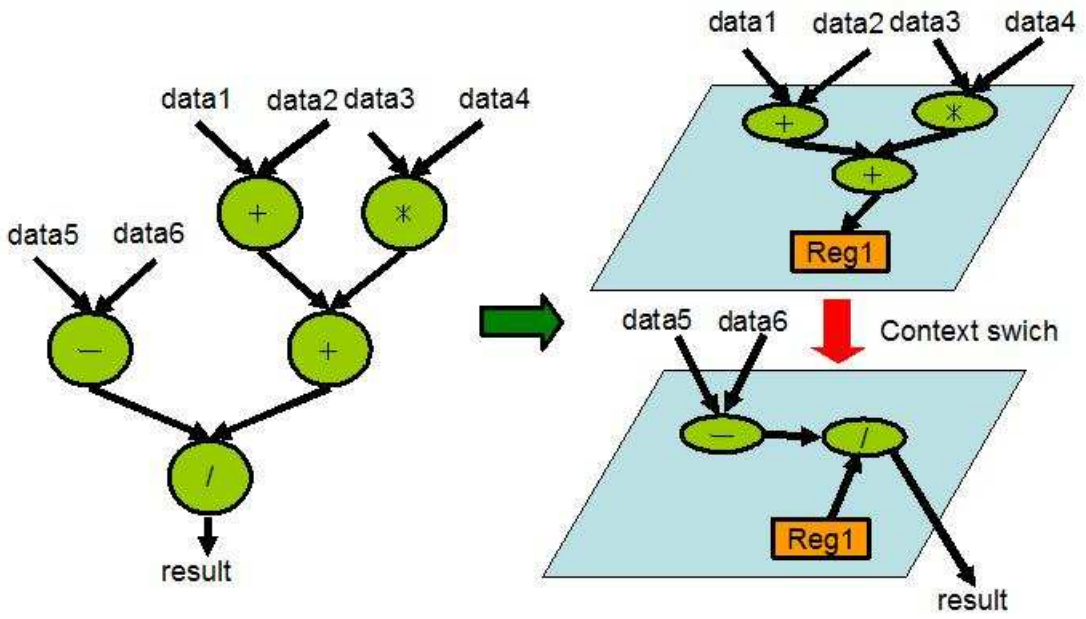

図 3.19: DRP におけるデータ駆動型の処理法

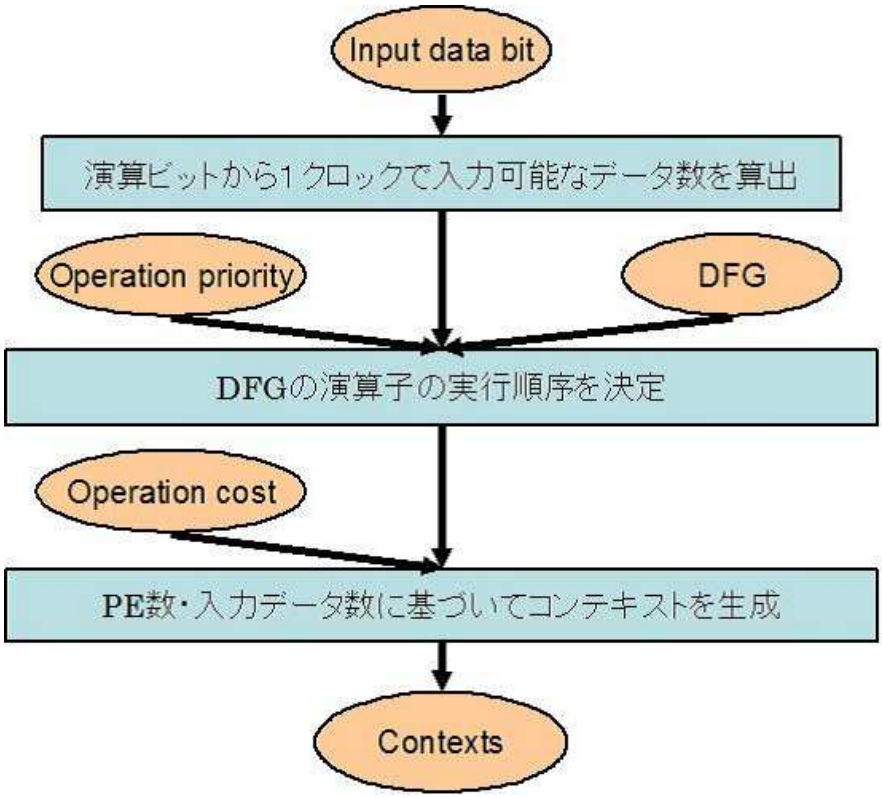

図 3.20: コンテキスト削減アルゴリズムの概要

表 3.7: 演算器コスト

演算器	演算ビット	PE 数	実行クロック	コンテキスト数
加算器	8	1	1	1
	16	2	1	1
	32	4	1	1
減算器	8	1	1	1
	16	2	1	1
	32	4	1	1
乗算器	8	0	2	2
	16	0	2	2
	32	0	2	2
除算器	8	7	9	1
	16	11	17	1
	32	19	33	1
シフト除算	8	2	1	1
	16	3	1	1
	32	5	1	1

コンテキストで複数クロックで実行である。シフト命令による除算は、PE 内の DMU を利用するので 1 クロックで実行可能である。

表 3.8 は、各演算器の実行優先度であり、DFG 中の演算の実行順序の決定に利用する。DFG の同じ深さに複数の演算がある場合、実行優先度が高いものから実行する。乗算器は、2 クロックで実行可能だが 2 つのコンテキスト数を要求するので、結果出力待ちの間に他の処理ができるように最も先に実行する。残りの演算器は、全て 1 コンテキストで実行可能なので、実行クロックが最も多い除算器を 2 番目の実行優先度にした。加算器、減算器とシフト除算は、コンテキスト数と実行クロックが全く同じなので、PE の数が多い加算器と減算器を次の優先度にした。

回路構成生成アルゴリズム

図 3.21 は、データ駆動型のコンテキストを生成するためのアルゴリズムである。アルゴリズムの入力は、

- 入力データビット数
- PT 数
- DFG
- 演算器コストテーブル

表 3.8: 演算の実行優先度

実行優先度	演算
1	乗算
2	除算
3	加算
3	減算
4	シフト除算

- 演算の実行優先度テーブル

である。1ー5 行目のデータ入力の後、DataPerClock 関数で入力データビット数と DRP-1 の入力ビット幅から 1 クロックで入力可能なデータ数を計算する。(6 行目)DataPerClock 関数内の式は、式 (3.4) である。

$$input\_data = \left\lfloor \frac{input\_bit}{ope\_bit} \right\rfloor \tag{3.4}$$

- output_data : 1 クロックで入力可能なデータの数
- output_bit : DRP-1 の入力ビット幅
- ope_bit : 入力データビット数

続いて、7 行目の TotalPe 関数で PT 数と 1PT の PE 数テーブルから 1PT の PE 数 (max_PE) を求める。8 行目の Priority 関数で DFG 中の演算に実行順序を決定する。Priority 関数の入力は、DFG と演算の実行順序の 2 項目である。図 3.22 は、priority 関数の例であり、図中の () が実行順序を示している。初めに、データが入力と同時に実行可能な step1 の 4 つ演算器は、演算器優先度テーブルから実行順序が付けられる。同優先度の演算器には、同じ優先度を付ける。次に step1 の演算結果と新しい入力データから演算を行う step2 の演算器に対して優先度を付けていく。9ー13 行目で、CreateContext 関数で i 番目のコンテキスト (context[i]) の回路構成を決定する。この while 文は、演算に実行順序が付属された DFG(dfg_added_priority) の演算がなくなるまで行う。

CreateContext 関数のアルゴリズム

図 3.23 は、回路構成生成アルゴリズムの 11 行目の create 関数のアルゴリズムを示している。create_group 関数の入力は、

- 優先度が付属された DFG
- 1 クロックで入力可能なデータ数
- 演算器コスト

```
1: input_bit ← bit number of input data;
2: num_par ← parallelism;
3: dfg ← DFG;
4: tab_ope_cost ← operator cost;
5: tab_ope_pri ← operator priority;
6: input_data = DataPerClock(input_bit);
7: max_PE = TotalPe(num_par);
8: dfg_added_priority = Priority(dfg, tab_ope_pri);
9: i = 0;
10: do{
11:       context[i] = CreateContext(dfg_added_priority,
                                     input_data , tab_ope_cost, max_PE);
12:       i ++;
13: }while(dfg_added_operator == NULL)
```

図 3.21: コンテキスト削減アルゴリズム

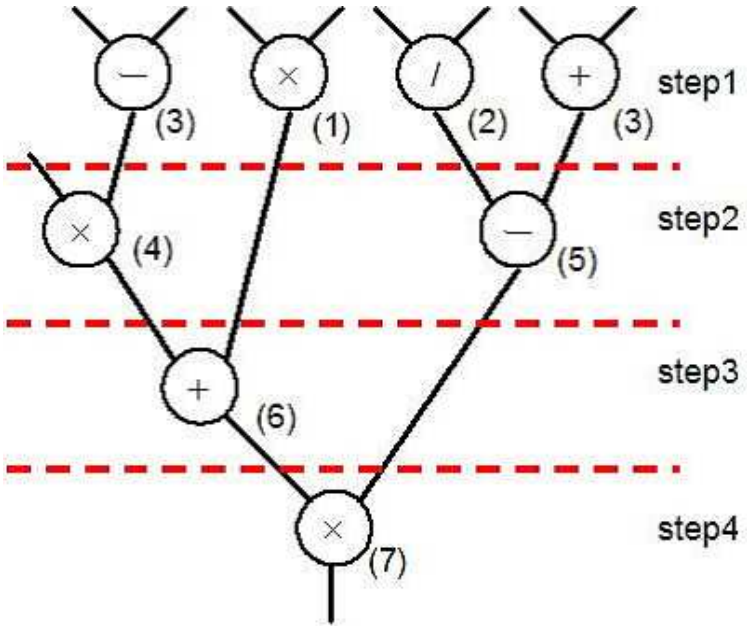

図 3.22: Priority 関数の例

表 3.9: 各 PT 数の 1PT における PE 数

PT 数	PE 数
2	192
3	128
4	64
5	64

- 1PT 当たりの PE 数

である。初めに各変数の初期化を 3,4 行目で行い、5ー17 行目の while 文を実行する。6 行目では、現在の入力データから実行可能な演算器があるかを確認する。もし、実行可能な演算器がある場合は、7ー10 行目のステートメントを実行し、逆の場合は 1 つの回路構成が決定し 15 行目の retrun で回路構成生成アルゴリズムに戻る。7 行目は、max 関数で優先度が付属された DFG(dfg_added_priority) から優先度が高い演算器を選出し、演算器コストから選出された演算器の PE 数を operaor_cost 関数で取得する。取得した PE 数を現在のコンテキスト内の PE 数 (current_PE) に足し、8 行目で 1PT の PE 数を超えているかどうかを確認する。表 3.9 は、各演算ビットの 1PT 当たりの PE の数である。PE 数が超えていない場合は、選出された演算器をコンテキスト内の回路集合 (group[]) に加え、優先度が付属された DFG から削除する。もし、現在の PE 数が 1PT 当たりの PE 数を超えてしまった場合は、13 行目の return で基のアルゴリズムに戻る。

```
1: create_group(dfg_added_priority, input_data, ope_cost, max_PE)
2: {
3:   group[] ← NULL;
4:   current_PE = 0;
5:  while(1){
6:    if(search(dfg_added_priority input_data)== true){
7:      current_PE += operator_cost(max(dfg_added_priority), ope_cost);
8:       If(current_PE < max_PE){
9:          group[] ← max(dfg_added_priority);
10:          remove(dfg_added_priority, max(dfg_added_priority));
11:       }
12:       else
13:           return group[];
14:      else
15:         return group[];
16:   }
17:}
```

図 3.23: CreateContext 関数のアルゴリズム

第4章　実験・評価

4.1　はじめに

本章では、前章で述べた提案手法の実験と評価について述べる。また、実験したプログラムの実装方法についても述べる。

4.2　実装環境

表 4.1 に、実装環境を示す。DRP の推奨 OS は、Vine Linux3.1 であるが、2.4 系のカーネルであれば Vine Linux3.2 でも稼動する。Cronus2 は、DRP-1 を搭載している評価ボードである。ホスト PC との接続インターフェイスは、PCI バスと HyperTransport であるが、HyperTransport は、標準構成ではサポートをしていないので、本研究では、PCI バスでホスト PC と接続した。DRP コンパイラは、C 言語を DRP 用に拡張した Behavioral Design Language(BDL) 専用のコンパイラであり、統合開発環境の Musketeer に搭載されている。表 4.2 に本研究で設計した回路の動作検証に使用した検証ツールを示す。動作検証は、Musketeer 上で実行可能である。BDL 検証は、記述したアルゴリズムが正確なものかをソフトウェア実行で検証する。DRP 用に拡張されたステートメントに関しては、専用のヘッダーファイルを使用することでソフトウェアで実行できる。Verilog 検証は、DRP コンパイラから出力された Verilog HDL を基に波形シミュレーションをすることで回路の動作検証を行う。

4.3　実験

提案手法を評価するために以下の項目について実験を行った。

- 実行クロック数の削減率
- 実行時間の比較
- コンテキストの削減数

従来手法は、天野ら [19] が述べている DRP-1 上での典型的なストリーム処理法とした。従来手法の流れは、

表 4.1: 実装環境

ホスト CPU	Pentium4 2.80CGHz
ホストメモリ	1.0GB
OS	Vine Linux 3.2
カーネル	2.4.27-0v17smp
ボード	Cronus2
開発環境	Musketeer
開発言語	Behavioral Design Language
コンパイラ	DRP コンパイラ

表 4.2: 動作検証ツール

BDL 検証	gcc 3.3.2
Verilog 検証	Modelsim SE 6.1e

1. データを複数クロックで入力し、内部メモリに保存する。
2. 内部メモリから並列にデータを取り出し演算を実行し、演算結果を再度内部メモリに保存する。
3. 内部メモリから演算結果を連続的に出力

であり、3 が終了したら 1 に戻り、処理を繰り返す。評価に利用する従来手法は、DRP コンパイラの自動スケジューリングモードで論理合成を行い、動作周波数とコンテキスト数のトレードオフを解析するための機能である Musketeer の反復合成機能を利用して最も高速に実行できるものとした。提案手法の実装は、コンテキストの分割を明示的に行うことが可能な手動スケジューリングモードで行った。理論上 5 つの Tile でオーバーラップをさせることができるが、配線が困難であるため DRP コンパイラの Place & Route で常にエラーを吐き出して実現することができなかったので、2ー4Tile によるオーバーラップで評価を行った。

4.3.1 実験環境

図 4.1 は、ホスト PC と DRP 評価ボードの Cronus2 である。実験は、実装環境で述べた PC と同じ PC で行った。Cronus2 は、図 4.2 のように PCI バスでホスト PC と接続されている。本研究の実験のために利用したソフトウェアを表 4.3 に示した。ソフトウェアの実行時間を測定するために gcc3.2.2 を利用し、コンパイル時に gcc の最適化オプションである-O3 を付属した。DRP の実行時間の測定のために実行クロックと DRP の動作周波

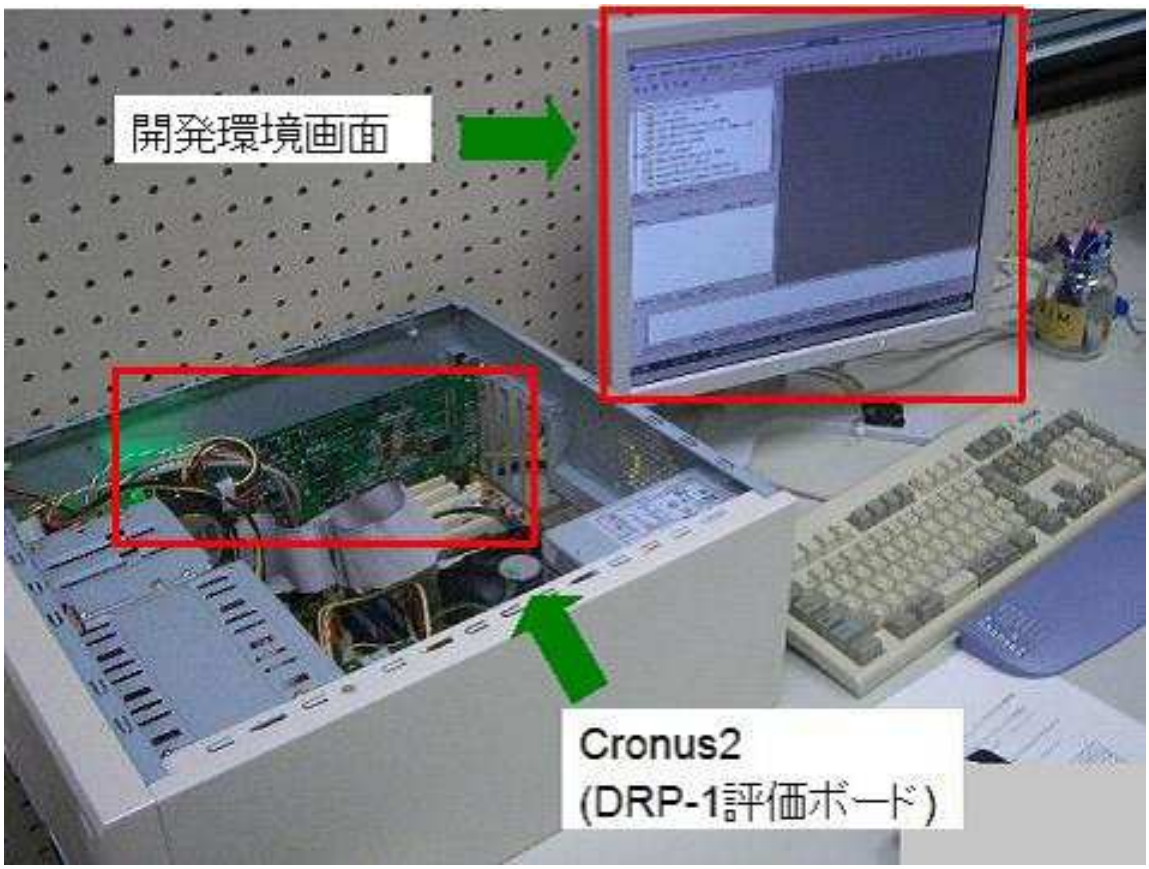

図 4.1: 実験環境

数を測定する必要があるので、実行クロックは ModelSim SE 6.1e でカウントし、DRP の動作周波数は DRP コンパイラで Place & Route 後のものを利用した。コンテキスト数は、DRP の動作周波数と同様に DRP コンパイラを用いてフロント合成後のものをカウントした。

4.3.2 実験プログラムと実装

実験するために利用したプログラムは、画像処理のラプラシアン・フィルタである。ラプラシアン・フィルタは、与えられた画像を 2 次微分することによって、緩やかな濃度勾配でも濃度が変化するエッジ部分が強調されるエッジ検出フィルタである。式 (4.1) は、ラプラシアン・フィルタのプログラム式である。式 (4.1) の演算結果は、0 を中心とした正負の値となるので、プログラム中ではシステムの最高濃度の中間濃度値である 127 を加えている。また、最高濃度値 255 を超えた場合は、演算結果値を 255、負の値の場合は、演算結果値を 0 と濃度補完も行っている。入力データは、簡易化のために 1002*1002 サイズのグレイ画像を想定し、入力データのビット数は 8 ビットとした。

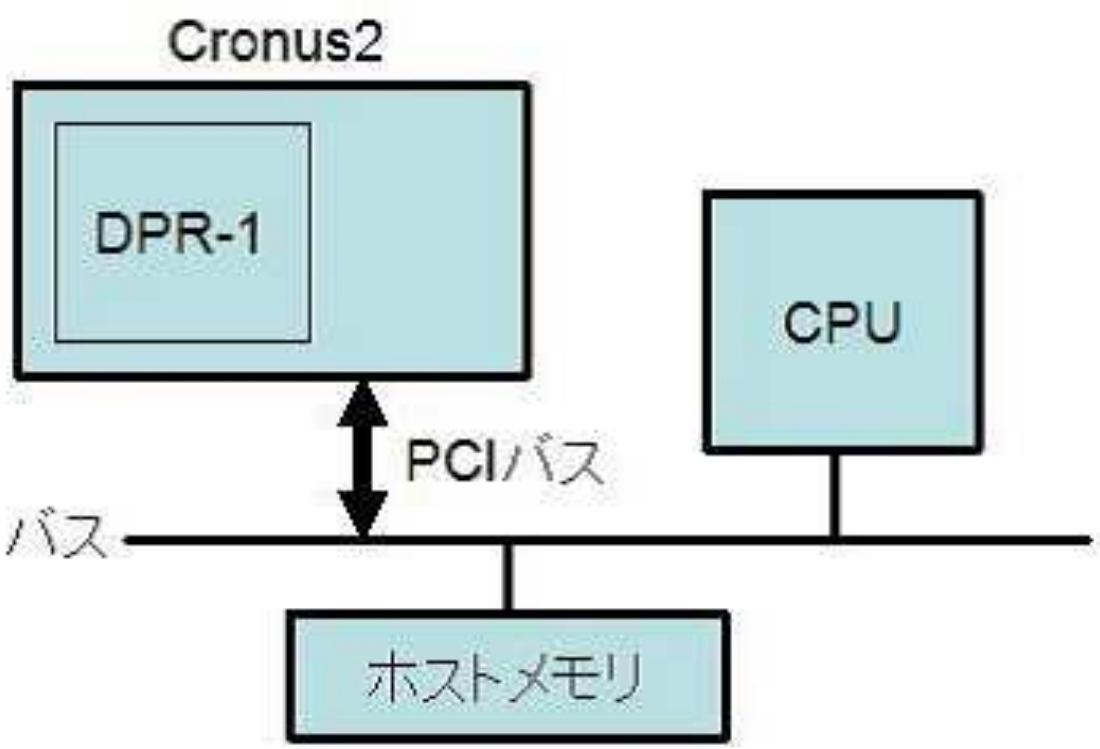

図 4.2: 実験環境の概要

$$f[i,j]_{newimage} = f[i-1,j] + f[i+1,j] + f[i,j-1] + f[i,j+1] - 4f[i,j] \qquad (4.1)$$

従来手法の実装

従来手法の実装は以下流れで行った。

1. 図 4.3 のように画像の 1002 × 3 のデータを読み込み、VMEM に保存する。
2. VMEM から入力データを読み込み、8 並列で演算処理を行う。乗算は、DRP-1 上にある乗算器を利用した。演算結果を再び VMEM に保存する。
3. 8 つの演算結果を 1 クロックで同時に出力する。

1 回目のデータの出力が完了後、図 4.3 の緑色で囲まれている範囲のデータをを読み込む。この流れを 1000 回行う。

表 4.3: 実験で利用したソフトウェア

ソフトウェアコンパイラ	gcc3.2.2
ソフトウェアの実行時間の測定	gprof2.14.90
DRP の実行クロックの測定	Modelsim SE 6.1e
DRP の動作周波数の測定・コンテキスト数	DRP コンパイラ

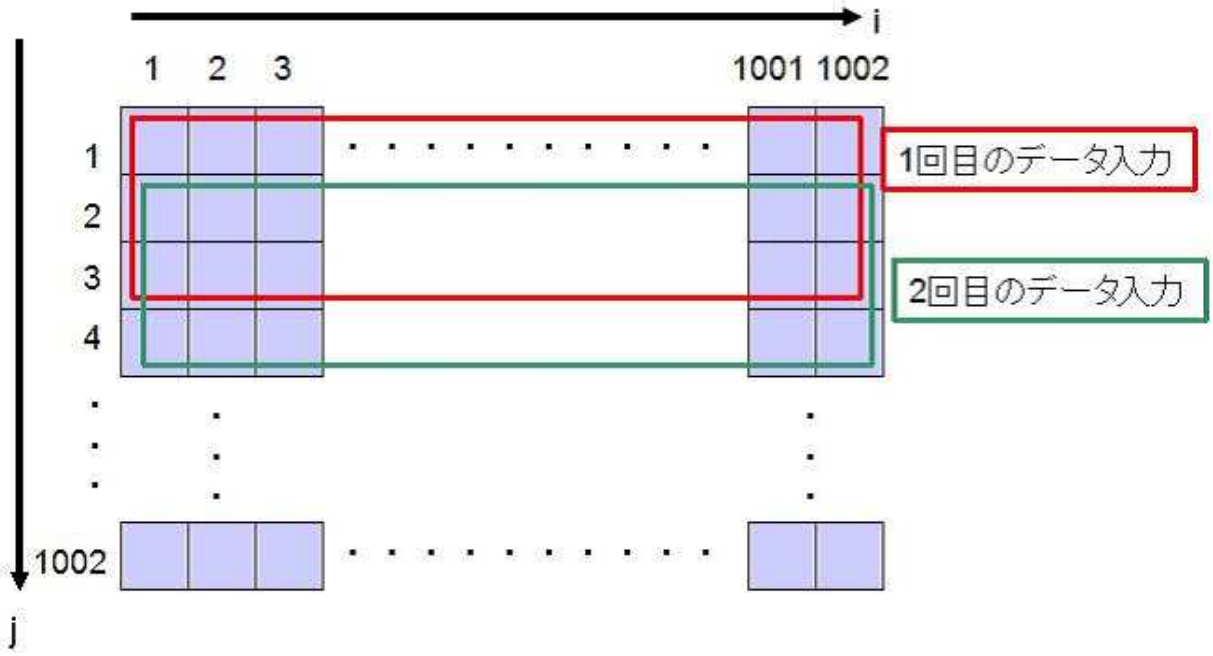

図 4.3: 従来手法のデータの取り込み

提案手法の実装

提案手法の実装は以下流れで行った。

1. 図 4.4 の赤色で囲まれた範囲のデータを読み込み、処理を行う Tile にデータを転送し、演算処理を行う。このときに、次の演算に必要なデータを Input Tile で保存する。
2. 図 4.4 の緑色で囲まれた範囲のデータを読み込み、処理を行う Tile にデータを転送する。また、直前に演算を開始した Tile に残りのデータを転送する。
3. 各 Tile の処理が終わり次第、Output Tile に演算結果を転送し、データ出力を行う。

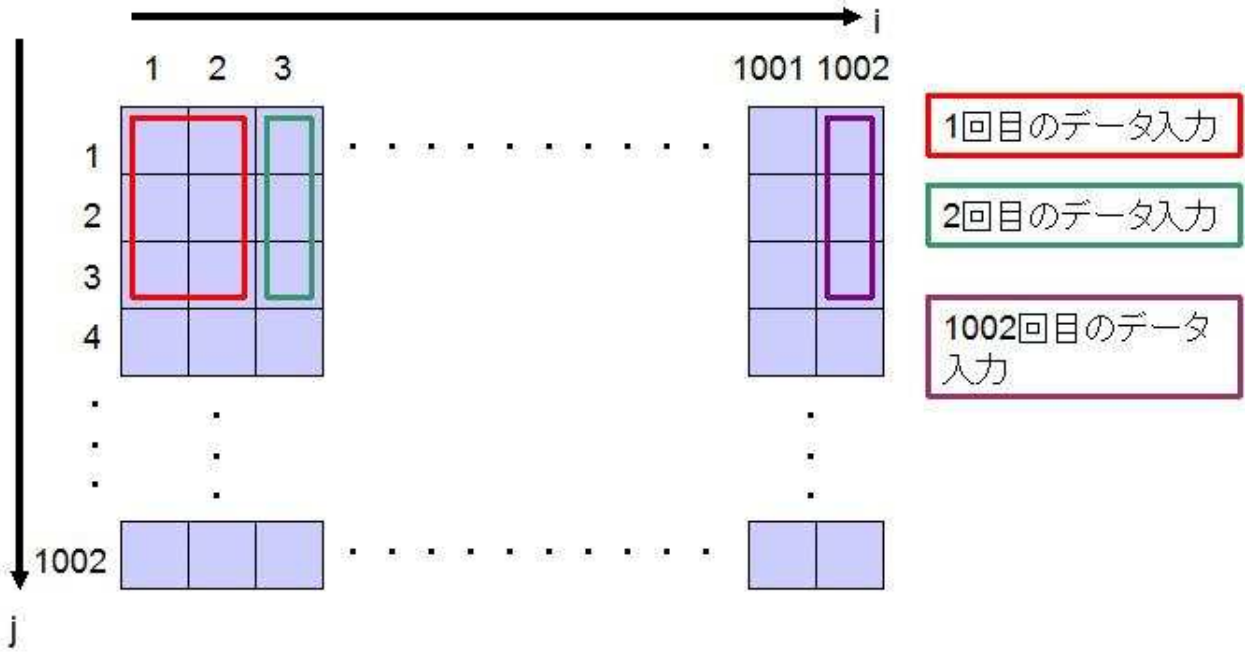

図 4.4: 提案手法のデータの取り込み

表 4.4: 従来手法の実行クロックの内訳

処理	実行クロック	割合 (%)
input	1006	44.5
computation	1001	44.3
output	251	11.1

4.4 評価

4.4.1 入出力と演算のオーバーラップ

図 4.5ー4.10 は、ラプラシアン・フィルタの従来手法と提案手法の動作を示している。従来手法は、図 4.5ー図 4.7 で示したように数クロックの初期化後にデータ入力が開始され、演算処理・出力処理を行っている。表 4.4 で示したように入出力の実行クロックが全実行クロックの 50%以上を占めていることがわかる。一方、提案手法は、図 4.8ー4.10 のように初期化後に 1 クロックずつ入力タイミングをずらして各 PT に入力データを転送している。この結果、入出力と演算のオーバーラップが実現できていることが分かる。

Output Tile では、連続出力するためにループ命令を利用しているので、ループカウントの判定を行うために 1 クロックは必須ある。従って、複数のデータを連続出力後に 1 クロックの待ち時間が発生する。ラプラシアン・フィルタの 3PT(図 4.9) と 4PT(図 4.10) では 1 クロックの待ち時間後に再び連続的にデータを出力しているが、2PT(図 4.8) では、2 データを出力後 2 クロック待ち時間が発生している。これは、2 データ出力後の時点で各ルートの演算が完了しておらず Output Tile がデータ待ちをしているからである。このことから、Output Tile のデータ待ち時間をなくすためには、処理の実行クロックと PT 数のトレードオフを考慮する必要があることが分かる。

表 4.5 は、入出力と演算をオーバーラップさせることによって削減したクロック数と削減率を示している。提案手法は、最大で従来手法の実行クロックの 55%削減することができた。しかし、4.11 のように PT 数が向上するにつれて削減率の伸びが低くなっていく。これは、1 クロックで入力データを全て入力できる状況に近くなっていくためである。従って、一般的な並列処理を実行できた場合が最も最適な処理を行っていることであると言える。

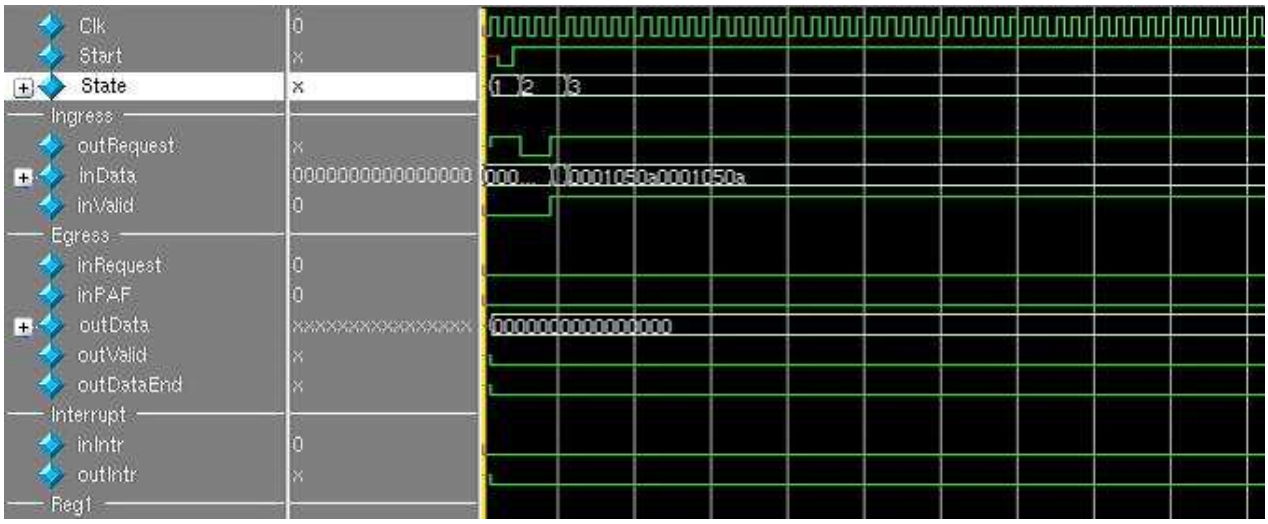

図 4.5: 従来手法のデータ入力

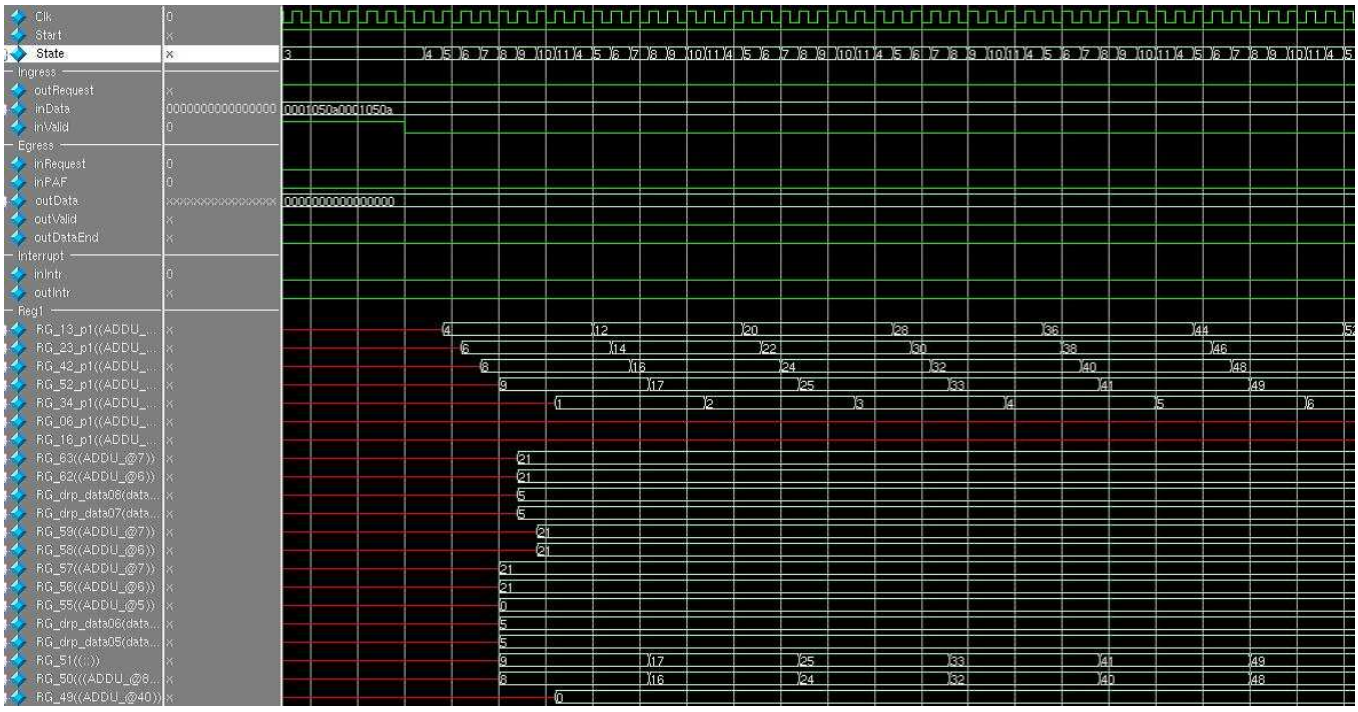

図 4.6: 従来手法の演算処理

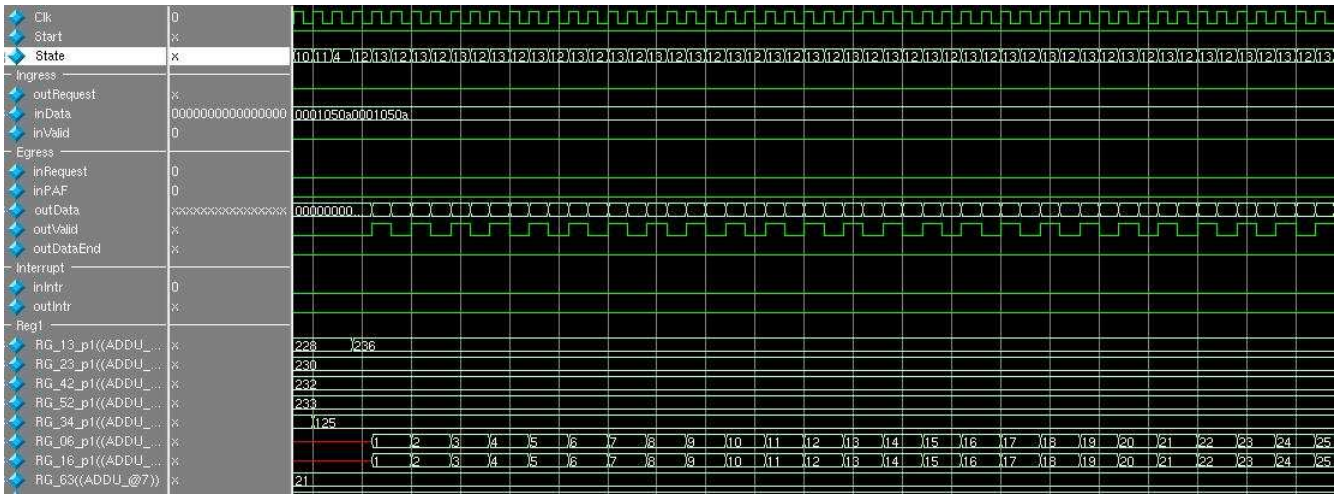

図 4.7: 従来手法のデータ出力

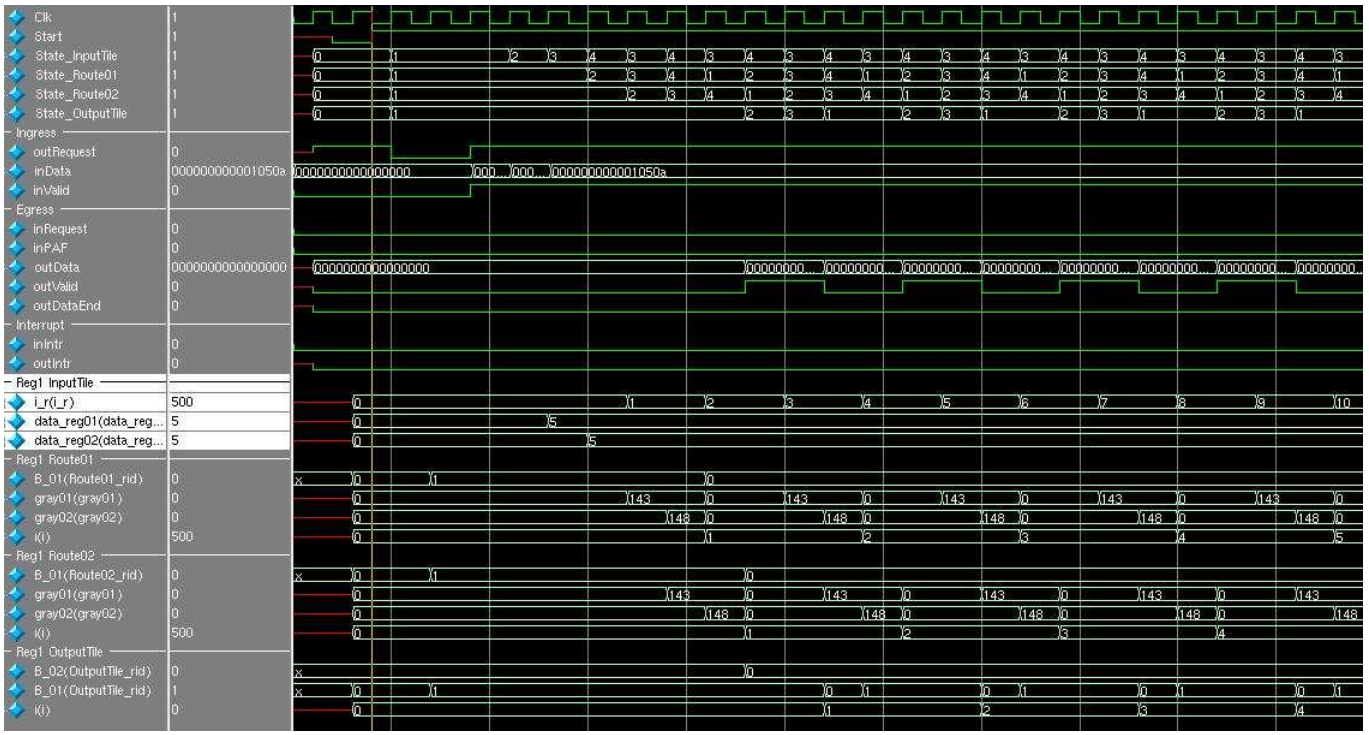

図 4.8: 2PT の動作

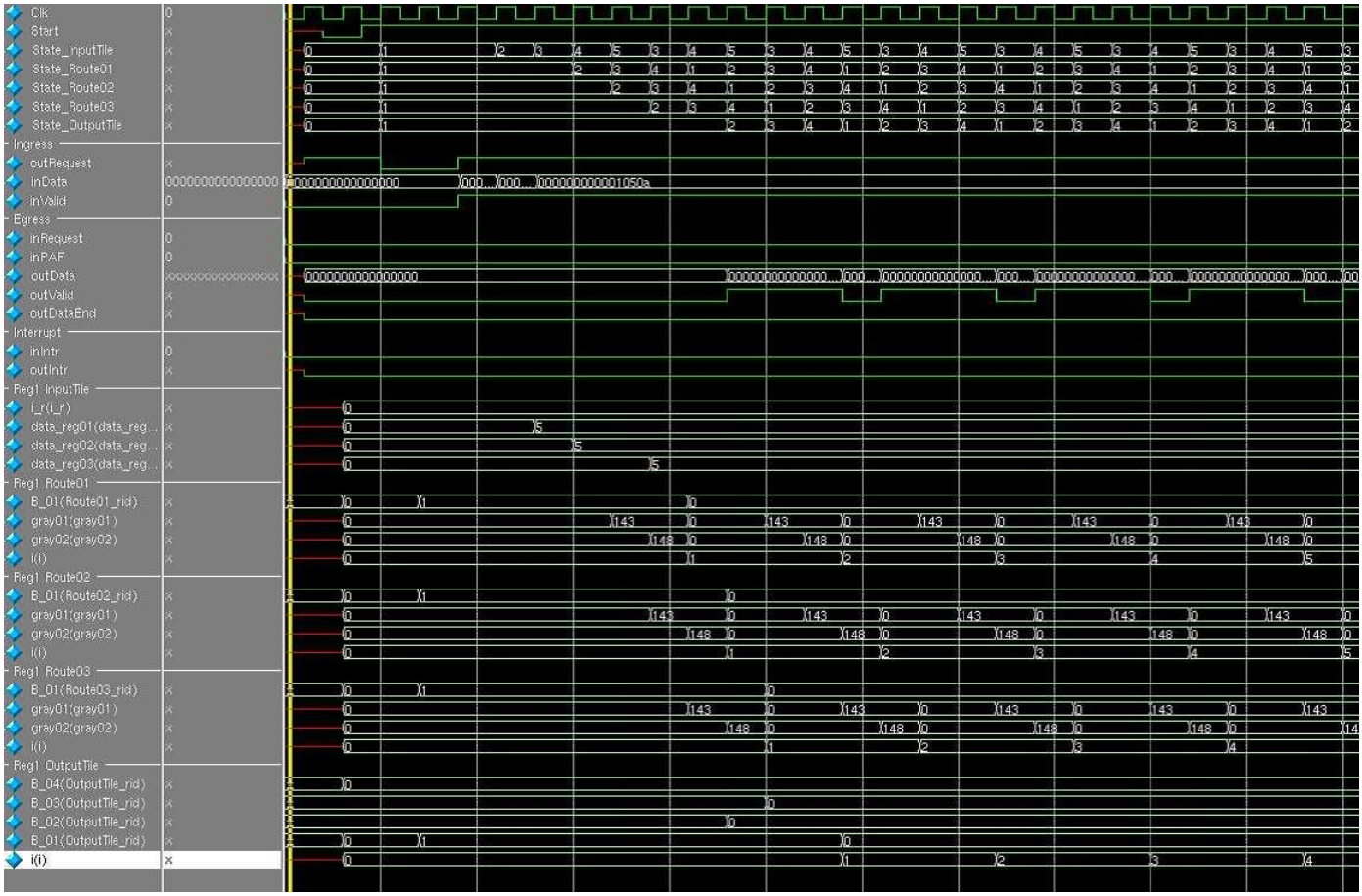
Clk
Start
State_InputTile
State_Route01
State_Route02
State_Route03
State_OutputTile
Ingress
outRequest
inData
inValid
Egress
inRequest
inPAF
outData
outValid
outDataEnd
Interrupt
inIntr
outIntr
Reg1 InputTile
i_r(i_r)
data_reg01(data_reg...
data_reg02(data_reg...
data_reg03(data_reg...
Reg1 Route01
B_01(Route01_rid)
gray01(gray01)
gray02(gray02)
i(i)
Reg1 Route02
B_01(Route02_rid)
gray01(gray01)
gray02(gray02)
i(i)
Reg1 Route03
B_01(Route03_rid)
gray01(gray01)
gray02(gray02)
i(i)
Reg1 OutputTile
B_04(OutputTile_rid)
B_03(OutputTile_rid)
B_02(OutputTile_rid)
B_01(OutputTile_rid)
i(i)

図 4.9: 3PT の動作

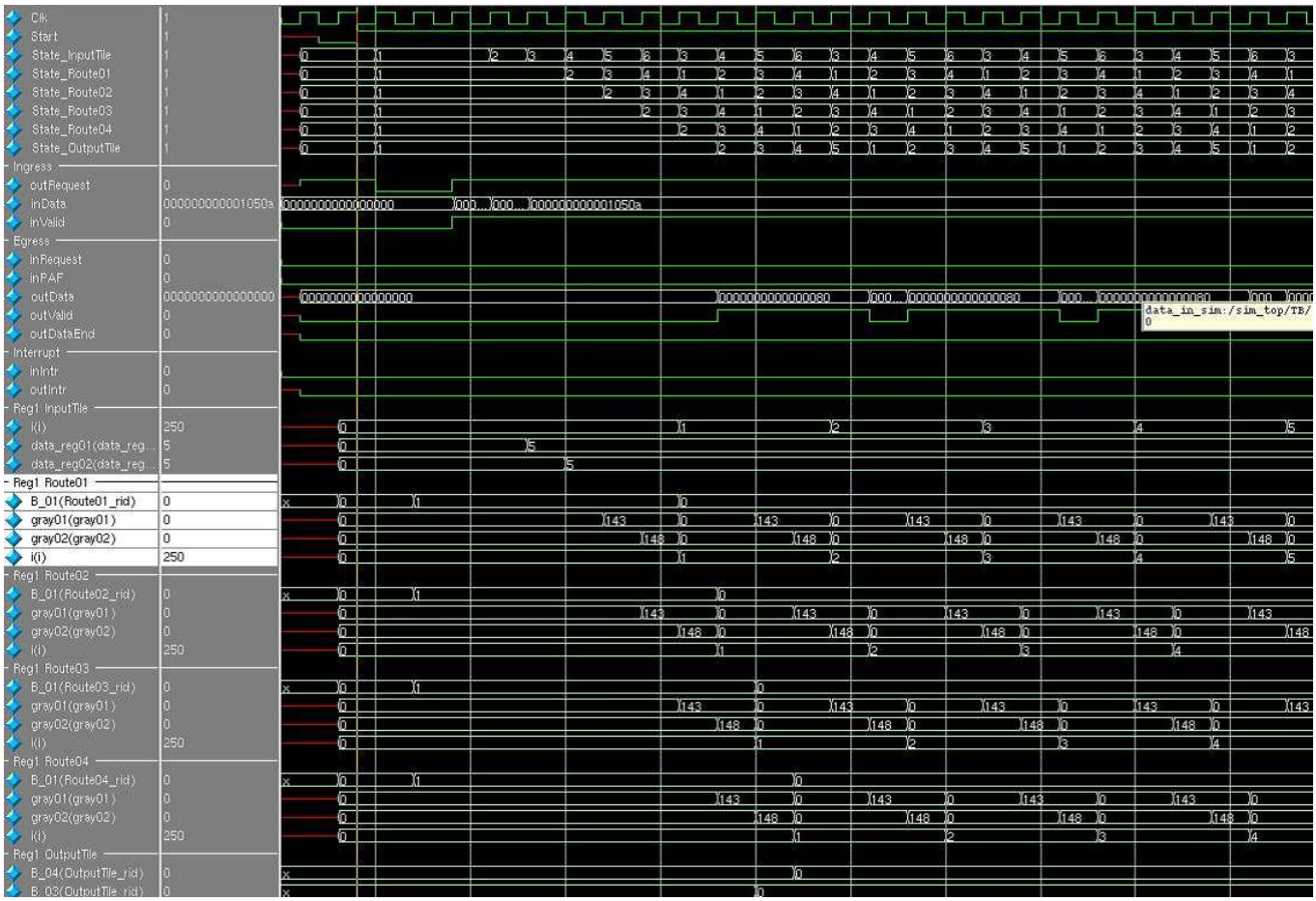
Clk
Start
State_InputTile
State_Route01
State_Route02
State_Route03
State_Route04
State_OutputTile
Ingress
outRequest
inData
inValid
Egress
inRequest
inPAF
outData
outValid
outDataEnd
Interrupt
inIntr
outIntr
Reg1 InputTile
i(i)
data_reg01(data_reg...
data_reg02(data_reg...
Reg1 Route01
B_01(Route01_rid)
gray01(gray01)
gray02(gray02)
i(i)
Reg1 Route02
B_01(Route02_rid)
gray01(gray01)
gray02(gray02)
i(i)
Reg1 Route03
B_01(Route03_rid)
gray01(gray01)
gray02(gray02)
i(i)
Reg1 Route04
B_01(Route04_rid)
gray01(gray01)
gray02(gray02)
i(i)
Reg1 OutputTile
B_04(OutputTile_rid)
B_03(OutputTile_rid)
data_in_sim:/sim_top/TB/

図 4.10: 4PT の動作

表 4.5: 入出力と演算のオーバーラップによるクロックの削減率

プログラム	手法	全体クロック数	従来手法のクロック数の差	削減率 (%)
Laplacian Filter	従来手法	2258002	0	100
	2PT	2007002	251000	88.8
	3PT	1342002	916000	59.4
	4PT	1258002	1000000	55.7

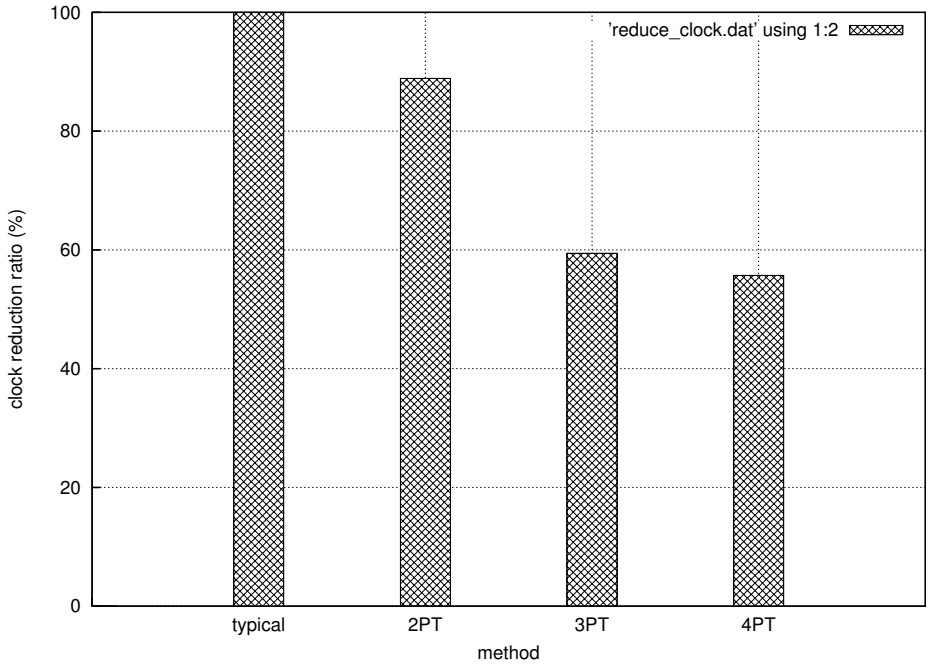

図 4.11: クロック削減率

表 4.6: 実行時間の比較

プログラム	手法	実行時間 (ms)	CPU の実行時間の差 (ms)	向上率 (%)
Laplacian Filter	CPU	49.56	0	100
	従来手法	32.72	16.83	151
	2PT	34.01	15.54	145
	PT3	22.74	26.81	217
	PT4	20.96	28.59	236

4.4.2 実行速度の比較

各プログラムの CPU、従来手法、提案手法の PT 数の計 5 つの実行時間を表 4.6 と図 4.12 に示す。CPU の実行時間の測定は、関数化したループ内の命令文を gprof で 20 回測定し、その平均値とした。一方、DRP の実行時間は、ModelSim でカウントされた実行クロック数と DRP コンパイラの Place & Route 後の動作周波をもとに式 (4.2) で求めた。

$$execution\_time\_on\_DRP = \frac{clock\_cycle}{frequency} \tag{4.2}$$

- execution_time_on_DRP : DRP での実行時間
- clock_cycle : 実行クロックサイクル
- frequency : DRP の動作周波数

提案手法は、CPU と従来手法より高速に実行することが可能であることが分かった。提案手法は、入出力と演算がオーバーラップしているので、2PT では 2 つのデータを、3PT では 3 つのデータを、4PT では 4 つのデータを連続的に出力している。この結果、従来手法より実行時間を短縮することが可能となった。

図 4.13 は、従来手法と提案手法の動作周波数を示している。一般的なの並列処理であれば、並列度が倍になれば実行時間も倍になると考えられるが、PT へのデータ入出力のタイミングがずれていること、動作周波数が低下していることから実行時間は、PT 数の増加に伴った倍率で向上していない。しかし、提案手法は、従来手法の動作周波数より約 10MHz 低いが、実行クロック数を削減したことで提案手法より処理の高速化を実現していることが分かる。

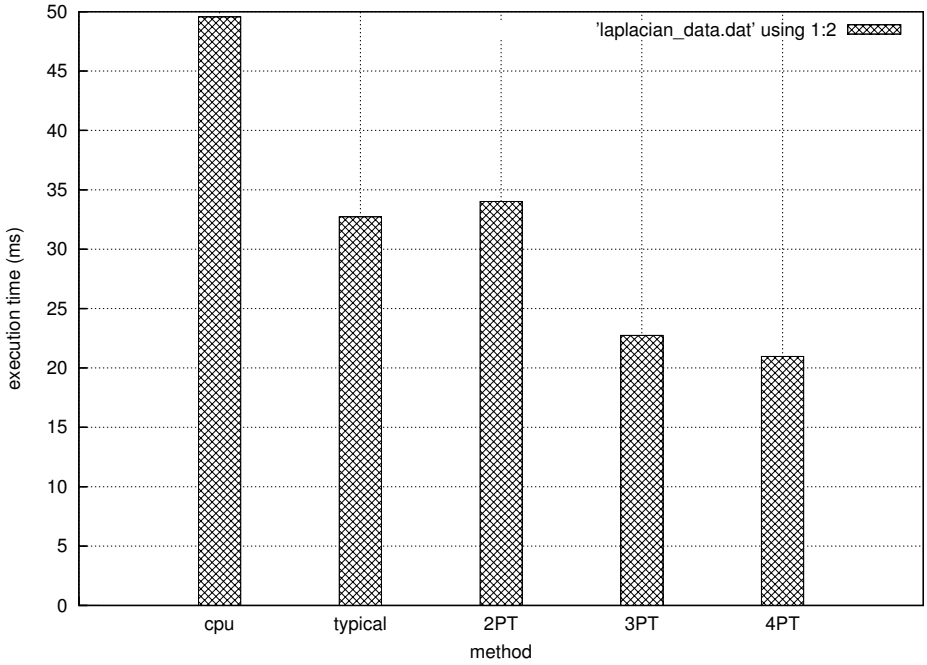

図 4.12: ラプラシアン・フィルタの実行時間の比較

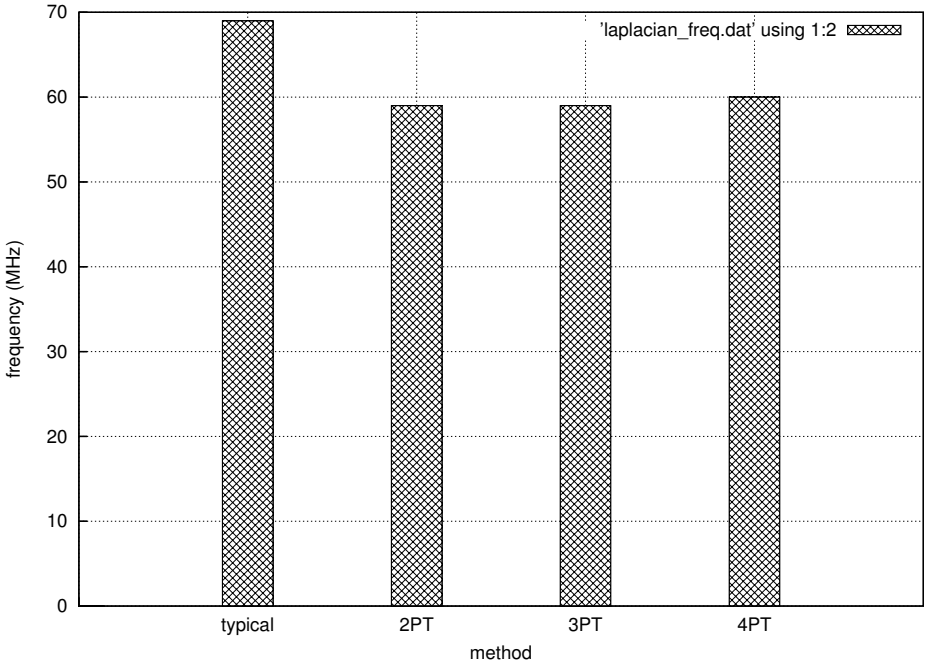

図 4.13: ラプラシアン・フィルタにおける従来手法と提案手法の動作周波数の比較

表 4.7: コンテキスト数の比較

プログラム	手法	コンテキスト数	内訳		
			Input Tile	Processing Tile	Output Tile
Laplacian Filter	従来手法	13			
	2PT	5	5	5	4
	3PT	6	6	5	6
	4PT	6	6	5	6

4.4.3 コンテキスト数の比較

DRP コンパイラのフロントエンド合成でコンテキストの生成を行うので、実験プログラムのコンテキスト数はフロントエンド合成後のコンテキスト数とした。表 4.7 は、従来手法と提案手法の各 PT 数のコンテキスト数である。提案手法のコンテキスト数は、Input Tile、PT、Output Tile のコンテキスト数で最も大きいコンテキスト数を示している。

提案手法は、従来手法のコンテキスト数の半分以下になっていることが分かる。Input Tile と Output Tile のコンテキスト数が PT のコンテキスト数より多いため、全体のコンテキスト数でボトルネックとなっていることが分かる。本研究では、PT を対象としてコンテキスト数の削減を行っていたので、Input Tile と Output Tile に対しては何も工夫を凝らしていない。

4.5 考察

本研究で提案した 2 つの手法について実験を行い、評価を行った。ラプラシアン・フィルタで提案手法の実行時間は、CPU より最大で 236%向上した。これは、動作周波数が低下しても、データ入出力と演算をオーバーラップさせることで実行クロックを最大 55%に削減したことで達成することができた。2 つ目の提案であるデータ駆動方式によるアルゴリズムの削減を適応しても PT のコンテキスト数は、従来手法の半分のコンテキスト数で実装できることが分かった。また、PT 数を向上させると Input Tile と Output Tile のコンテキスト数が増加し最終的には、全体のコンテキスト数が増加してしまうことが判明した。しかし、Input Tile と Output Tile は、DRP の制約により大幅にコンテキストを削減することが不可能である。従って、この PT 数と Input Tile と Output Tile のトレードオフを解析する必要がある。

4.6　まとめ

本章では、前章で提案した 2 つの手法を評価するために実験を行った。評価は、実行クロック数の削減率、実行時間の比較、コンテキスト数の削減率を行った。実験の結果、本研究で提案した両手法とも従来の DRP の処理法より有効な手法であることが分かった。

第5章　まとめ

5.1　本研究の目的

本研究では、マルチコンテキスト型動的リコンフィギャラブルプロセッサである NEC エレクトロニクス社の Dynamically Reconfigurable Processor(DRP) を対象デバイスとした。DRP は、リコンフィギャラブルデバイスの一種で任意の回路を何度でも構成可能なデバイスである。基本セルの Processing Elements(PE) が 8 ビットの ALU ベースで構成されているので、ストリーム処理の高速処理と面積効率の向上を実現することが可能である。また、DRP は、複数のコンテキストをデバイス内に保存することができるので、瞬時にコンテキストスイッチを行うことができる。動的にデバイスの一部を再構成する動的部分再構成も容易に実現できることも大きなメリットである。このような DRP の特徴を活かして、CPU でボトルネックとなる処理を DRP でハードウェア化することによって高速に処理できる可能性がある。

しかし、現在の DRP の I/O ビット幅が入出力共に 64 ビットと非常に少ないので、処理によっては演算に必要なデータを 1 クロックで入力することが不可能となる。この場合、複数クロックでデータを入力してから演算を行わなければならない。従って、典型的な DRP の処理は、入力・演算・出力と 3 つのフェーズで構成されることとなる。天野ら [19] の報告では、DRP 本来の I/O 数である 128 ビットに強化したとしてもデータ入出力時間が全体の実行時間の 20 %近くになると述べている。また、DRP は、コンテキストを 16 個までしかデバイス内に保存することができない。16 個以上のコンテキストを要求する処理を実行する場合、実行中にデバイス内に保存できなかったコンテキストをホストメモリからダウンロードする必要があり、この結果、ダウンロード時間がボトルネックとなるってしまう。このことから、処理が要求するコンテキスト数を削減することは、コンテキスト型方式のデバイスにとって課題である。そこで、本研究では入出力時間の削減とコンテキスト数の削減を目的とした。

5.2　提案手法

本研究では、入出力時間の削減とコンテキスト数の削減を達成するために 2 つ手法を提案した。1 つ目の手法は、入出力時間の削減するためにデータの入出力と演算をオーバーラップさせて処理する手法である。DRP は、動的部分再構成をコアとなる Tile 毎に行うことができるので、他の Tile に依存することなく独自に処理を行うことが可能である。こ

の機能を利用して、あるTileでデータ入力を行っている間に、他のTileは演算処理を行っている、また、他のTileが演算処理を行っている間に、あるTileはデータ出力を行っているというように各Tileへのデータ入力のタイミングをずらすことによってデータ入出力と演算処理をオーバーラップさせる。この手法により、データ入出力の時間を隠蔽することが可能である。

2つ目の手法は、1クロックで入力可能なデータ数からデータ駆動型のコンテキストを生成することでコンテキストを削減する手法である。DRPで最も高速に処理するためには、動作周波数とコンテキスト数のトレードオフで決定される。DRPコンパイラは、Control Data-Flow Graph(CDFG)を分割することで複数のコンテキストを生成する。このコンテキスト内の一つをDRP上に実現するので、CDFGを細かく分割すればハードウェア上のクリティカルパスを減少させて動作周波数を向上させることができる。しかし、CDFGを細かく分割することは、コンテキスト数の増加と実行クロック数の増加となる。本提案では、1つ目の提案でオーバーラップ処理を行っているので、従来のDRPの処理方法より高速に処理をすることが可能と考えられる。従って、コンテキスト数を削減するためにCDFGを粗く分割することが可能であると考えられる。分割は、1クロックで入力されたデータからCDFGで実行可能な演算子を探索する方法で行う。この分割で生成されるコンテキストは、1クロックで入力されたデータのデータ駆動型コンテキストである。

5.3 実験結果

提案手法を評価するために、CPUと従来手法との実行時間の比較・実行クロック数の削減率・コンテキストの削減数の評価を行った。その結果、ラプラシアン・フィルタを4PTで処理を行った場合、実行時間の向上率が236%向上した。これは、本研究で提案したDRPによるデータ入出力と演算をオーバーラップさせたことで達成できた。この手法を用いると動作周波数は低下するが、実行クロック数を大幅に削減することができるので高速に処理することができる。2つ目の提案であるデータ駆動方式によるコンテキスト数の削減を適応した結果、従来研究のコンテキスト数の半分で実装することが可能となった。

5.4 本研究の貢献

データ入出力を演算処理とオーバーラップさせて実行クロック数を大幅に削減でいたので、演算を1回行うために多くの入力データが必要な処理に対しては、本提案を適応することで実行時間を短縮することが可能である。また、コンテキスト数を多く要求する処理に対してデータ駆動方式のコンテキストを適用することでコンテキストを削減することができる。これにより動作周波数は、低下するが1つ目の提案による実行クロックの削減により高速に処理することが可能である。

5.5　今後の課題

本研究では、4PT まで実装し評価を行ったが、それ以上の PT 数については行っていない。従って、DRP で 5PT 以上実現できた場合の本研究の有効性を見出す必要がある。また、PT 数を上げると Input Tile と Output Tile のコンテキスト数が増加してしまうことが本研究で明らかになったので、Input Tile と Output Tile をどのような回路構成にするか、どのようなアーキテクチャにするべきかを提案する必要がある。

参考文献

[1] Masayasu Suzuki, et al, "Stream Applications on the Dynamically Reconfigurable Processor" Proc. IEEE International Conference on Field-Programmable Technology(ICFPT'04),pp137-152

[2] Noriaki Suzuki, et al, "Implementing and Evaluating Stream Applications on the Dynamically Reconfigurable Processor" Proc. IEEE Field-Programmable Custom Computing Machines(FCCM'04),pp.328-329

[3] H.Isonaga, Y.Inoguchi "Implimentation of high-speed audiofingerprint system using FPGAs" IEICE Technical Report,RECONF2005-77

[4] 丹羽雄平, 前田敦司, 山口善教 "暗号通信パケットストリームの n-gram 予測による FPGA 動的再構成手法とその評価" 情報処理学会論文誌:コンピューティングシステム Vol.48,No.SIG3(ACS17),pp27-43(2007)

[5] S.Abe, et al, "Implementation of AES-CBC on the Dynamically Reconfigurable Processor", Technical Report of IEICE

[6] R.Nakahashi, et al, "A New Design Method for Implementing Real-Time Embedded Systems on Dynamically Reconfigurable Proccessors", IEICE Technical Report,RECONF2005-58

[7] T.Kitaoka,H.Amano,K.Anjo "Compressing configuration data of DRP for quick rumtime configuraton", Technical Report of IEICE

[8] 鈴木, 長谷川, 阿部, ヴマン, 天野 "動的再構成可能プロセッサにおけるコンテクストメモリの削減方法", 電子情報通信学会論文誌 D Vol.J89-D No.6 pp1101-1109

[9] Xilinx Corporation, "Virtex-2 Platform FPGAs:Complete Data Sheet",

[10] Xilinx Corporation, "マイクロプロセッサを使用するスレーブシリアル/SelectMAP モードによるザイリンクス FPGA のコンフィギュレーション",

[11] Y.Hori,H.Yokoyama,H.Sakane,K.Toda, "Design and Implementation of Self Run-Time Partical Reconfigurable System", IEICE Technical Report,RECONF2006-75

[12] Zhiyuan Li,Scott Hauck, “Configuration Compression for Virtex FPGAs”, Proc. IEEE Field-Programmable Custom Computing Machines(FCCM’01),pp.147-159

[13] Krishna Raghuraman,Haibo Wang, and Spyros Tragoudas, “Minimizing FPGA Reconfiguration Data at Logic Level”, Proc. ISQED’06,pp.27-29

[14] 末吉敏則, 天野英晴, “リコンフィギャラブルシステム”, 東京, オーム社,2006 p279

[15] NEC エレクトロニクス社http://www.necel.co.jp, Nov. 2007

[16] IPFlex Inc. http://www.ipflex.com/, Nov. 2007

[17] Narasimhan Ramasubramanian,Ram Subramanian and Santosh Pande, “Automatic Compilation of Loops to Exploit Operator Parallelism on Configurable Arithmetic Logic Units”, IEEE Trans. On Parallel And Distributed System, Vol.13,No1,pp45-66,Jun 2002

[18] Y.Inoguchi “Outline of the Ultra Fine Grained Parallel Processing by FPGA”, Proc. High Performance Computing and Grid in Asia Pacific Region (HPCAsia’04),pp.434-411

[19] H.Amano, S.Abe, K.Deguchi, Y.Hasegawa An I/O Mechanism on a Dynamically Reconfigurable Processor-Which Should Be Moved:Data or Configuration Proc. IEEE Field Programmable Logic and Applications (FPL’05),pp.347-352

[20] X.P.Ling, H.Amano WASMII: a data driven computer on a virtual hardware Proc. IEEE Field-Programmable Custom Computing Machines(FCCM’93),pp.33-42

[21] Y.Awashima, et al, “C Compiler for Dynamically Reconfigurable Processor: DRP”, Technical Report of IEICE VLD2003-118,CPSY2003-27(2004)

[22] T.Toi, N.Nakamura, Y.Kato, T.Awashima, K.Wakabayashi “High-Level Synthsis Challenges and Solutions for a Dynamically Reconfigurable Processor”, IEEE/ACM International Conference ICCAD’06, November 5-9, pp702-708

[23] 酒井 幸市 “デジタル画像処理の基礎と応用 -基本概念から顔画像認識まで-”, CQ 出版社,2003

謝辞

本研究を行うにあたり、多くの御助言、御指導を賜りました北陸先端科学技術大学院大学の情報科学センター井口 寧准教授に深く感謝するとともに、ここに御礼申し上げます。

貴重な御助言、御意見を頂いた松澤 照男教授、田中 清史准教授に深く感謝いたします。

井口研究室のゼミで貴重な御助言、御意見を頂いた佐藤 幸紀助教と井口研究室のメンバーに深く感謝いたします。

そして、ハードウェア合同ゼミで貴重な御助言、御意見を頂いた皆様、本研究を応援していただいた藤田唯氏、同フロアの松本 正教授と松本研究室のメンバー、田中研究室の請園智玲氏、日比野研究室の矢澤慶樹氏、同期の親友に深く感謝いたします。また、

本研究を進めるにあたり DRP-1 と開発環境を提供していただき、トラブルのサポートを行っていただいた NEC エレクトロニクス社に深く感謝いたします。

最後に、ここまで暖かく見守って下さった父、母、妹に深く感謝いたします。