社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

リコンフィギュラブルハードウェアを用いた高速ストリーム処理の一検討

福田エリック駿† 川島 英之†† 井上 浩明††† 藤井 太郎†††† 古田浩一朗††††
浅井 哲也† 本村 真人†

† 北海道大学 情報科学研究科 〒 060–0814 北海道札幌市北区北 14 条西 9 丁目
†† 筑波大学 システム情報工学研究科 〒 305–8573 茨城県つくば市天王台 1–1–1
††† 日本電気株式会社 〒 211–8666 神奈川県川崎市中原区下沼部 1753 番地
†††† ルネサスエレクトロニクス株式会社 〒 211–8668 神奈川県川崎市中原区下沼部 1753 番地

あらまし リコンフィギュラブルハードウェアの有力な応用先としてストリーム処理が注目されている。本研究では、ストリーム処理のオペレータの一つである Window Join のリコンフィギュラブルハードウェアへの実装を通して、1) ソフトウェアエンジニアがリコンフィギュラブルハードウェアを活用するためには何が必要か、2) そのような環境で適応型ストリーム処理はどれほど効果があるのか、の二つを明らかにする。C 言語による開発が可能な動的リコンフィギュラブルハードウェアを用いた実装を通し、ソフトウェアエンジニアがストリーム処理用ハードウェアを開発するために必要なハードウェア開発の知識を、「五つの視点」としてまとめた。また、動的リコンフィギュラブルハードウェアを使うことで適応型ストリーム処理の電力効率を向上させることができる。
キーワード ストリーム処理、ウィンドウ結合、適応型クエリ処理、動的リコンフィギュラブルハードウェア

An Implementation of High Performance Stream Processing on a Reconfigurable Hardware

Eric-Shun FUKUDA†, Hideyuki KAWASHIMA††, Hiroaki INOUE†††, Taro FUJII††††, Koichiro FURUTA††††, Tetsuya ASAI†, and Masato MOTOMURA†

† Graduate School of Information Science and Technology, Hokkaido University Kita 14, Nishi 9, Kita-ku, Sapporo, Hokkaido, 060–0814
†† Graduate School of Systems and Information Engineering, University of Tsukuba 1–1–1 Tennodai, Tsukuba, Ibaraki, 305–8573
††† NEC Corporation 1753 Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa, 211–8666
†††† Renesas Electronics Corporation 1753 Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa, 211–8668

Abstract Stream processing is one of the applications that reconfigurable hardware can be highly effective. In this paper, we give answers to the following questions by implementing *window join*, one of the operators of stream processing, on a reconfigurable hardware: 1) what do software engineers need in order to utilize reconfigurable hardware; 2) how can *adaptive stream processing* be implemented with such solution. Through our implementation on a dynamically reconfigurable hardware with C-based HLS, we found that software engineers would need five awarenesses in order to utilize hardware as a stream processing platform. Also, dynamically reconfigurable hardware improves the power efficiency of adaptive stream processing.
Key words stream processing, window join, adaptive query processing, dynamically reconfigurable hardware

1. はじめに

ビッグデータをリアルタイムに処理するための新たな方式であるストリーム処理が注目されている [1]。ストリーム処理ではデータベースへのアクセスを必ずしも行わないため、メモリやディスクへの入出力にかかるオーバーヘッドが少ない。従って並列分散化によって処理を高速化していた従来のサーバ処理よりも少ないサーバで高速なリアルタイム処理が可能であり、

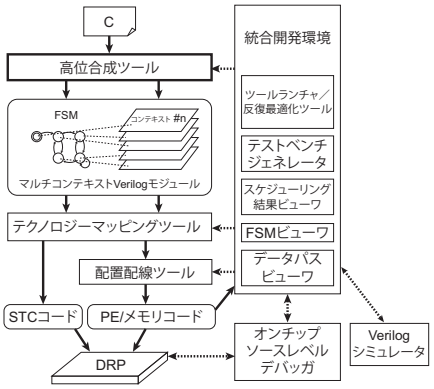

図 1　C コードからハードウェアへの変換の流れ

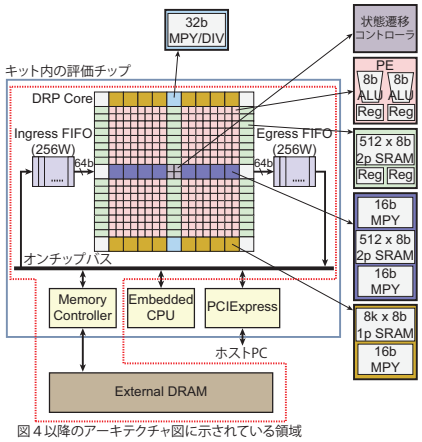

図 2　DRP の構成

増え続ける消費電力の削減にもつながるとの期待も高い。さらに、これまで主にソフトウェアで行われていたストリーム処理に、FPGA を応用して高速化・低消費電力化する研究が活発に行われている [2]〜[4]。

しかし、ストリーム処理を含めた特定のアプリケーションに対して開発されたハードウェアは、汎用プロセッサに比べて一般的にはるかに高い電力効率が得られる一方、(1) ソフトウェアエンジニアによる開発が難しい、(2) 処理の柔軟性が低い、という問題点があった。ストリーム処理を効率よく行う複雑なアルゴリズムは主にソフトウェアエンジニアが開発するため (1) は重要な課題であり、適応型処理が重要視されているストリーム処理では (2) も大きな問題である。

筆者らは、このような問題への対処方法として、Dynamically Reconfigurable Processor (DRP) [5] が有効であると考えた。DRP はこれまで主に画像処理の分野でハードウェアエンジニアに利用されてきたが、高位合成ツール（HLS）を使った C 言語での開発が可能であるため、ソフトウェアエンジニアにとって比較的使いやすい環境である。また、動的再構成が可能であるため、適応型ストリーム処理に適していると考えられる。

本研究では、ストリーム処理のオペレータの一つである Window Join を DRP 上に実装し、次のことを明らかにする。

- 最先端の HLS によりどれくらいの性能が得られるか
- どのようなハードウェア知識が HLS による開発に必要か
- 動的再構成機能がどのように適応型処理に役立つか

2. 関 連 研 究

[2] の研究が提案するシステムでは、単純な演算を行う C 言語の関数をいくつか定義し、それらを HLS により回路化する。それらの回路が、開発者の指定した正規表現に対応する信号を出力すると、イベント検知信号が出力される。このシステムは FPGA に実装され高い性能を示した。その後、このシステムは動的再構成機能を取り入れ、システムの動作中にクエリを動的に切り替えられるようなシステムに改良された [6]。

一方、[3] の研究では SQL を開発言語に用いた。この研究では、SQL 文を小さな演算要素に分解し、それをテンプレート化された回路に置き換える。このシステムによりストリーム処理回路を FPGA 上に実装し、CPU よりも高速・省電力であることを示した。この研究をベースにした、クロックサイクルごとに再構成が可能なシステムも別グループにより提案されている [7]。

これらの研究では、ストリーム処理に特化したフレームワークを導入することでソフトウェアエンジニアでも容易にハードウェア処理を利用することを可能にしている。我々が用いる環境はストリーム処理に特化しているわけではないが、完全に C 言語による開発が可能であるためソフトウェアエンジニアによる利用のハードルは低いと考えられる。

3. DRP: 評価プラットフォーム

DRP は 2003 年に発表された動的リコンフィギュラブルプロセッサである [5]。DRP 上には小さな要素回路とメモリが並べられ、ユーザが記述したプログラムはハードウェア構成として回路上にマッピングされる。回路はクロック毎に再構成が可能で、回路の切り替えはユーザのプログラムから生成された有限状態機械 (FSM) によって制御される。回路の再構成は FSM で状態遷移が発生した際に行われる [8]。

DRP を使ったシステムの設計は HLS を用いて行う [9]。図 1 に示すように、コンパイルはまず C 言語で記述されたプログラムを、FSM と、FSM の各状態に対応するハードウェアの「コンテキスト」に変換する。そして FSM は状態遷移コントローラ (State Transition Controller: STC) にコンパイルされ、ハードウェアコンテキストは PE とメモリのアレイにマッピングされる（図 2）。演算の並列性は基本的に、プログラムに含まれる演算要素（あるいはデータ構造）を PE（あるいはメモリ）にマッピングする際に得られる。開発ツールは、予測条件分岐や演算ツリーのバランス調整などの最適化により、演算の段数や使用 PE・メモリ数の最小化を行う。また、ループ展開やループフォールディング（パイプライン化）、ループ結合などの最適化をプログラマが指定することができ、これにより更なる演算の並列化が可能となる [8]。このような機能を使うことによって、データと制御の依存性を侵さない限りにおいて高い並列性を実現することができる。

4. DRP における Window Join

Join は重要なデータベース処理演算の一つである。Join の基本的な処理は、2 つのテーブル間でキーとなるフィールドを比較し、そのフィールドの値が一致したタプルを結合し、新たなテーブルを作ることである（図 3a)。通常のデータベース処理ではテーブル内のタプルの数が有限であるのに対し、リアルタイムのストリームではタプルの数は無限である。そこでストリーム処理ではスライディング・ウィンドウを導入し、Join を適用すべきストリームの範囲を指定する（図 3b)。これは Window Join という名前の由来になっている。ストリームは絶えずウィンドウを通過していくので、新たなタプルがウィンドウに入った際に、そのタプルともう一方のストリームのウィンドウ内に含まれるタプルすべてを比較すれば良い（図 3c)。

4.1 評価方法

Window Join のアルゴリズムは単純だが、実際のアプリケーションではウィンドウの大きさがタプル数万個分になることもあるため、これら全てに対して並列処理を行うことで高速化する専用ハードウェアを設計するのは現実的ではない。この問題に対処するため、Handshake Join と呼ばれるアルゴリズムが提案・実装されている [4], [10]。このアルゴリズムではウィンドウが小さなサブウィンドウに分解され、それぞれのサブウィンドウの中で並列処理が行われる。こうすることで巨大な Window Join の処理を分割し、複数の FPGA に分散させることができる。本研究ではこのコンセプトに基づき、分割された小さなサイズの Window Join を効率的に処理することを目的とする（以下、このサブウィンドウを単にウィンドウと呼ぶ)。

本研究では DRP の限られたリソース量を考慮し、次のような仮定の下で評価を行った。(1) 取り扱うウィンドウのサイズは 16 タプルとする。(2) ストリーム R と S のタプルはそれぞれ、16 ビットのキーとなるフィールドと 16 ビットの数値フィールドからなる合計 32 ビットの幅を持つものとする。キーの値はある範囲のランダムな値をとり、この範囲を調整することでタプル間の一致率を変化させることができる。

4.2 最も単純な実装

後述するアーキテクチャとの比較のため、CPU 向けの C 言語プログラムで最も単純と考えられるプログラムを作成した。ストリーム $R(S)$ 中のタプル $r(s)$ は $register_{R(S)}$ に代入され、その度に比較が行われる（図 4)。ソースコード内で宣言された $register_{R(S)}$ は DRP コア内でレジスタとして配置され、図 5 に示すようなアーキテクチャとなる。このアーキテクチャにおけるスループット（DRP へのタプルの入力レートとする）は 6.7 Mbps であった。

4.3 マッチテーブルを使ったアーキテクチャ

DRP において効率よく Window Join を行うため、図 7 に示すようなアーキテクチャの回路を実装した（[11] では前節で述べた最も単純な実装と本節で述べる実装の間の段階的な最適化について詳しく紹介している)。この回路を実現するために、図 6 に示すようなアルゴリズムを C 言語で記述した。この回路では以下（4.3.1〜4.3.5）に示すような工夫が施されている。

4.3.1 ウィンドウ内タプルのバッファリング

4.2 では同じタプルを複数回 DRAM から読み込む必要があったた。いま、Join を適用する範囲はウィンドウの中に限られているため、ウィンドウの分だけタプルを DRP コア内部にバッファリングすれば DRAM アクセスを減らせるはずである。そこで配列 w_R と w_S を宣言し、ウィンドウ内のタプルをバッファリングする（ただし $w_{R,S}$ はキーとなるフィールドのみを保持し、タプル全体は $storage_{R,S}$ に保持される)。w_{R_0} と w_{S_0}（新たに入力されたタプル）はそれぞれ w_S と w_R に保持されたタプルと比較される。

4.3.2 マッチテーブル

レジスタを使ったバッファは同時に複数の要素にアクセスが

```
1: loop
2:   register_R ⇐ ingressFIFO ⇐ r_t
3:   for i ⇐ 0 to N − 1 do
4:     register_S ⇐ ingressFIFO ⇐ s_{t−i}
5:     if regisger_S = register_R then
6:       O ⇐ egressFIFO ⇐ {register_S, register_R}
7:     end if
8:   end for
9:   (Repeat line 2 to 8 here reversing R and S, and r and s.)
10:  t ⇐ t + 1
11: end loop
```

図 4 最も単純なアルゴリズム

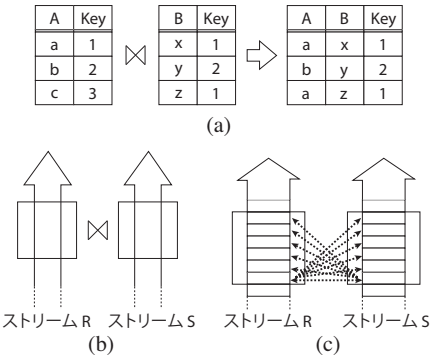

図 3 Join と Window Join

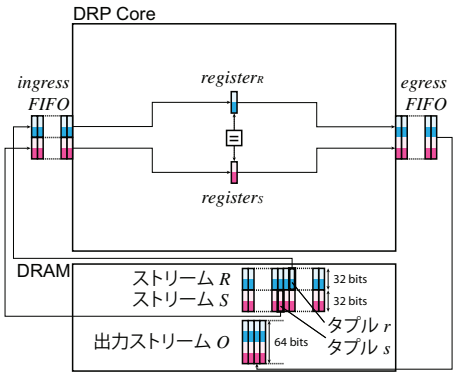

図 5 最も単純なアルゴリズムから合成されたアーキテクチャ

```
1: loop
2:   optr ⇐ 1
3:   for iptr ⇐ 1 to L do
4:     for i ⇐ N − 1 to 1 do
5:       w_Ri ⇐ w_R(i−1)
6:       w_Si ⇐ w_S(i−1)
7:     end for
8:     w_R0,S0, storage_R,S[iptr] ⇐ ingressFIFO ⇐ {r_t, s_t}
9:     table[iptr] ⇐ compare_2(w_R, w_S)
10:    O ⇐ egressFIFO ⇐ output_1(table[optr], storage_R,S)
11:    if table[optr] has no positive bits then
12:      optr ⇐ optr + 1
13:    end if
14:    t ⇐ t + 1
15:  end for
16: end loop
```

図 6　マッチテーブルを用いたアルゴリズム

```
1: loop
2:   for iptr ⇐ 1 to L do
3:     for i ⇐ N − 1 to 1 do
4:       w_Ri ⇐ w_R(i−1)
5:       w_Si ⇐ w_S(i−1)
6:     end for
7:     w_R0,S0, storage_R,S[iptr] ⇐ ingressFIFO ⇐ {r_t, s_t}
8:     buf_1∼2N ⇐push compare_3(w_R, w_S)
9:     t ⇐ t + 1
10:  end for
11:  for i ⇐ 1 to 2N do
12:    while buf_i is not empty do
13:      O ⇐ egressFIFO ⇐ output_2(pop(buf_i), storage_R,S)
14:    end while
15:  end for
16: end loop
```

図 8　低一致率ストリームに最適化したアルゴリズム

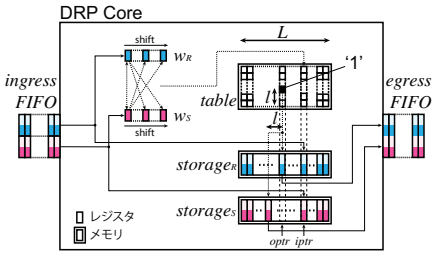

図 7　マッチテーブルを用いたアーキテクチャ（図 5 に記載した DRAM はこの図では省略した）

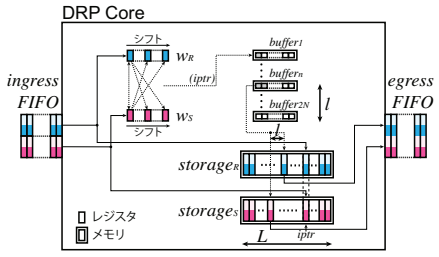

図 9　低一致率ストリームに最適化したアーキテクチャ（図 5 に記載した DRAM はこの図では省略した）

できるという点で使い勝手が良いが、数が限られているためレジスタの代わりにうまくメモリを活用する必要がある。そこで図 7 に示すような *table* と $storage_{R,S}$ を導入した。DRP コアに入力されたタプルは、w_{R_0} と w_{S_0} に入力されるのと同時に、メモリで実装された $storage_{R,S}$ の *iptr* が指す位置に保存される。w_R と w_S 間の比較結果は、$storage_{R,S}$ 上のタプルの位置を表すビットベクタとして *table* に保持される。出力処理では、*table* から得たビットベクタをデコードし、出力すべきタプルを $storage_{R,S}$ から読み出して *egressFIFO* に書き込む。

4.3.3　バーストメモリアクセス

4.2 ではストリーム R と S からタプルを一つずつ読んでいるため、DRAM からタプルが到着するまでに DRP コアでは待ち時間が生じてしまっていた。この問題に対する解決策として、まとまった量のタプルを DRAM から *ingressFIFO* に読み込む。こうすることで新たなタプルを待ち時間なしで *ingressFIFO* から得ることができる。これはソースコード上で専用の API を呼ぶことで実現できる。

4.3.4　並列テーブル参照

図 7 中の *table* の各列は 32 ビットのビットベクタとなっている。しかしそのエンコーディングとデコーディングは DRP のような粗粒度のアーキテクチャには不向きな作業である。デコーディングの部分の回路は遅延が大きく、最大動作周波数低下の原因となってしまう。この問題を解決するために *table* を 8 ビット幅に分割した（図が複雑になるのを避けるため、図 6 と 7 にこの工夫は反映されていない）。こうすることにより、ビットベクタのエンコード及びデコードがそれぞれ 4 並列で行える。

4.3.5　ループフォールディング (パイプライン化)

3. で述べたように、DRP の開発ツールにはループフォールディング（パイプライン化）を指定するオプションがあり、データや回路制御の依存性に違反しない範囲で直前のイテレーションの終了前に次のイテレーションを開始することができる。これは処理の並列性を向上させる強力な HLS の機能であるが、フォールディングを適用するループの中にループがあってはいけないなど、ソースコード上で満たすべき条件がいくつかある。これを満たすため、同様にツールが用意しているオプションを用いて内部ループを展開している。

4.4　低一致率のストリームに最適化したアーキテクチャ

タプル間の一致率が非常に低い場合には、出力処理はほとんど実行されないため、処理のメインループから出力処理を外すことで処理の高速化が図れる。そこで図 9 に示すようなアーキテクチャを実装した。図 8 に示すように、ソースコードでは入力・比較処理と出力処理を別々の `for` ループとして記述している。さらに *table* を廃止し、代わりに出力すべきタプルの $storage_{R,S}$ 上での位置を指すインデクスを保持するバッファを導入した。これは、*table* の列のビットベクタをエンコード・デコードするための遅延を小さくするためである。

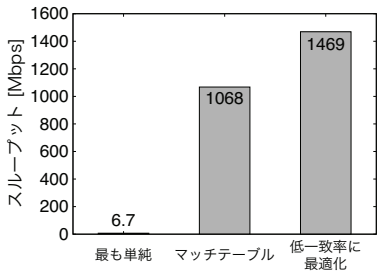

図 10　一致率が 0.1%の場合の各アーキテクチャのスループット

5. 考　　察

5.1 性能の評価

図 10 はストリーム間の一致率が 0.1%の場合の、各アーキテクチャのスループットを比較したものである。マッチテーブルを使ったアーキテクチャは最も単純な実装に比べ 216 倍に性能が改善している。比較のために C 言語で作成した単純な Window Join プログラムの性能を Intel® Core™ i5-2520M(注1) プロセッサ (2.5GHz) で測定したところ、スループットは 290Mbps であった。また、DRP の電力あたりのスループットは CPU の約 89 倍であった。ただし電力を直接測定する設備がないため、この値は最大 TDP を基準に計算した。

5.2 実装において必要な視点

1. で議論したように、リコンフィギュラブルハードウェアによる高速処理がより広く利用されるには、ソフトウェアプログラマにとってリコンフィギュラブルハードウェアがいかに使いやすいかが重要である。本研究では HLS を使用することによりハードウェアの詳細な設計の大部分が隠蔽されることがわかった。例えば従来の方法でハードウェアをパイプライン化する際には、回路の遅延を検証しながらデータの依存性を崩さないようにデータパスにパイプラインレジスタを挿入する必要があった。しかし本研究で用いた HLS では、パイプライン化したい `for` ループの直前にディレクティブとパラメータを書き込むだけで済む。

一方で、HLS を用いてもなおハードウェア開発の視点が必要となることも明らかになった。そこで本論文でこれまでに述べてきた Window Join の最適化の過程において得られた知見を、ストリーム処理プログラマがハードウェア処理を利用する上で必要となる「五つの視点」としてまとめた。

（1） **入出力の視点**：入出力は高スループットのストリーム処理システムを設計する際に最もボトルネックになりやすく、プログラマはより効率よく入出力を行い待ち時間を減らす方法を見つける必要がある。例えば本提案手法では、Window 内のタプルをバッファリングすることによって入力データの量を減らしたり（4.3.1）、メモリにバーストアクセスすることで待ち時間を減らしている（4.3.3）。アプリケーションに適したバッファリング方法は入出力を最大化し、ボトルネックを解消する助けとなる。

（2） **バッファリングの視点**：バッファを使うことによって処理を入力、中間、出力の各処理に分けることができるようになり、開発者は各処理におけるボトルネックを解消することに注力することが可能になる。例えば $w_{R,S}$（4.3.1）や $storage_{R,S}$（4.3.2）はタプル間の比較を行う中間処理から入出力処理を隔離する。これにより、開発者はメインループの並列性向上とステップ数低減に注力することができる。

（3） **リソース量の視点**：一方で、ハードウェアリソースが足りなくなった場合には開発者はバッファリングの仕組みを考え直す必要がある。これはバッファで隔離されているとは言え、入力、中間、出力の各処理はハードウェアリソースを分け合っているためである。$w_{R,S}$ で一致したタプルを $table$ や $buffer_i$、$storage_{R,S}$ を使わずに並列化したバッファで直接バッファリングしようとすると大量の ALU とレジスタが必要となるため、DRP でそのようなアーキテクチャを採用することは現実的ではない。本研究で提案したアーキテクチャではうまく内部メモリを使ってレジスタや ALU の不足を避けている（4.3.2）。バランス良くリソースを使用するアーキテクチャを導入することで、開発者はそれぞれの処理の最適化にフォーカスできる。

（4） **ループの視点**：処理全体を入力、中間、出力の各処理に分け、使用リソースのバランスが改善し、入出力がボトルネックではなくなると、次は中間処理内のループを最適化する段階となる。ループの並列度を向上させる上で最も一般的で強力な手法はフォールディングである（4.3.5）。しかし、フォールディングをするためにはソースコード上でいくつか満たしておくべき条件がある。例えばフォールディングしようとしているループの中にあるループは展開しておく必要がある。また、処理の順番やメモリのポート数などを考慮したコーディングをすることでデータの依存性を解決しておく必要がある。

（5） **リソースタイプの視点**：ALU やメモリの粒度といったハードウェアリソースの知識は、ターゲットハードウェア上でソースコードを効率よく実行する助けとなる。例えば、$table$ は 8 ビット幅のメモリを四つ用いることで 4 並列のアクセスを可能にしている（4.3.4）。メインループの処理を最適化する段階で、このような知識が特に重要となる。リソースの性質を知ることにより回路遅延が低減したり並列度が向上し、メインループのサイクル数を削減することができるためである。ソフトウェアを開発する際にもリソースの性質を知ることは重要であるが、ハードウェアを作成する際にはきわめて重要であり、プログラマはこれを意識する必要がある。

5.3 適応型ストリーム処理

図 11 はタプル間の一致率を変化させた際のマッチテーブルを用いたアーキテクチャと低一致率に最適化したアーキテクチャのスループットを比較している。一致率が 1.5%以上の時には低一致率に最適化したアーキテクチャよりマッチテーブルを用いたアーキテクチャの方が高速である。両アーキテクチャのスループットの比は一致率が 0.1%の時に 0.69、10%の時に 1.62 だった。DRP コアはクロック毎に回路を変更することができるため、ストリームの一致率が変化した場合に二つのアーキテクチャを切り替えて処理することにより、より効率的に処理が

(注1)：Intel and Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries.

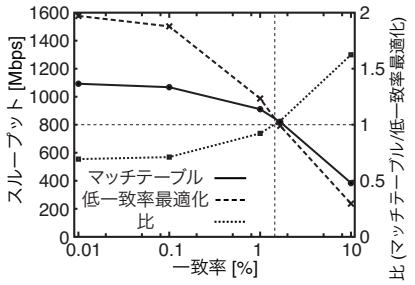

図 11　ストリーム間の一致率による各アーキテクチャの効率の違い

できると考えられる。この例が示すように、動的再構成は適応型ストリーム処理に利用することができ、今後より現実的なアプリケーションにおいて効果を検証する必要がある。

5.4　今後の方向性

本研究でハードウェア合成用に記述したソースコードはすべて C 言語で記述することができ、FSM 合成や ALU・レジスタ・メモリのマッピング、配置配線、入出力インターフェースの制御など、ハードウェアの詳細については開発ツールにより隠蔽されている。しかしながら、それでも上記の「五つの視点」は回路の最適化を行う上で重要である。問題なのは開発ツールが将来的に「五つの視点」を開発者が意識しなくても十分にシステムを最適化できるようになるかどうかである。Window Join の例を通して、我々はそれは難しいのではないかと考える。なぜなら最適化の途中では技術的な改善を行うだけでなくハードウェアのアーキテクチャ自体を構築することが求められるためである（技術的な改善だけであれば今後さらに開発ツールによる自動化が進むものと予想される）。

開発ツールがハードウェアの処理アーキテクチャを構築することを期待するよりも、「五つの視点」を生来的に備えた、ストリーム処理に特化したプログラミングモデルをプログラマに対する抽象レイヤとして構築すべきであると我々は考えている。したがって、我々が今後目指すのは 1)DRP やその他のリコンフィギュラブルハードウェアにおいて他のストリーム処理も実装して、ソフトウェアプログラムによるハードウェア設計に必要と思われるものを洗い出すことと、2) ストリーム処理アプリケーションを構築する上で必須条件を満たすプログラミングモデルを構築することである。

6. ま と め

本研究の目的は、ソフトウェアエンジニアがどれほどハードウェアによる高速化を実現することができるかと、どれほど適応型処理が効率的に行えるかを明らかにすることであった。そこで本論文では、DRP コアと呼ばれる動的リコンフィギュラブルハードウェアとその開発環境を用いて Window Join を実装するというアプローチをとった。その結果、以下の結論が導かれた。1) 最先端の HLS を用いることでハードウェアの詳細を開発者が意識することなく、並列演算を行う回路を完全に C 言語で開発することが可能である。 2) 最適化によりスループットを二桁向上させることが可能である。 3) 最適化する上で「五つ（入出力、バッファリング、リソース量、ループ、リソースタイプ）の視点」をソフトウェア開発者が持つことが重要である。4) 時間によって性質が変化するストリーム処理を行う上で動的リコンフィギュラブルハードウェアは非常に有望である。

Window Join を対象とした分析の結果、リコンフィギュラブルハードウェアがストリーム処理の高速化に広く使われるようになるためには専用のプログラミングモデルを構築しソフトウェア開発者に抽象レイヤを提供する必要があると考えられる。今後は、より多くのストリーム処理アプリケーションを FPGA などの他のリコンフィギュラブルハードウェアで実装し、本研究の提案をより発展させていく。

謝辞　本校執筆にあたり、有益なアドバイスを与えていただいたルネサスエレクトロニクス株式会社の犬尾武氏と戸井崇雄氏に心より感謝する。また、本研究は一部 JSPS 挑戦的萌芽研究 24650033 の助成を受けたものである。

文　献

[1] A. Arasu, S. Babu, and J. Widom, “The cql continuous query language: semantic foundations and query execution,” The VLDB Journal, vol.15, no.2, pp.121–142, 2006.

[2] H. Inoue, T. Takenaka, and M. Motomura, “20Gbps C-based complex event processing,” Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL), pp.97–102, 2011.

[3] R. Mueller, J. Teubner, and G. Alonso, “Streams on wires - a query compiler for FPGAs,” Proceedings of the VLDB Endowment, vol.2, no.1, pp.229–240, 2009.

[4] J. Teubner and R. Mueller, “How soccer players would do stream joins,” Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp.625–636, 2011.

[5] M. Motomura, “A dynamically reconfigurable processor architecture”. Microprocessor Forum, 2002.

[6] M. Takagi, T. Takenaka, and H. Inoue, “Dynamic query switching for complex event processing on FPGAs,” Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL), pp.599–602, 2012.

[7] T. Miyoshi, H. Kawashima, Y. Terada, and T. Yoshinaga, “A coarse grain reconfigurable processor architecture for stream processing engine,” Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL), pp.490–495, 2011.

[8] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing, “High-level synthesis challenges and solutions for a dynamically reconfigurable processor,” Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design (ICCAD), pp.702–708, 2006.

[9] T. Toi, T. Awashima, M. Motomura, and H. Amano, “Time and space-multiplexed compilation challenge for dynamically reconfigurable processors,” IEEE International Midwest Symposium on Circuits and Systems, pp.1–4, 2011.

[10] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, “Design and implementation of a handshake join architecture on FPGA,” IEICE Transactions on Information and Systems, vol.95, no.12, pp.2919–2927, 2012.

[11] 福田エリック駿，川島英之，井上浩明，浅井哲也，本村真人，“C 言語による動的リコンフィギュラブルハードウェアへの window join の実装，” 信学技報 (IN2013-26)，vol.113，no.106，pp.7–12，June 2013.