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.

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

福田エリック駿† 川島 英之†† 井上 浩明††† 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
E-mail: †{fukuda@lalsie., asai@, motomura@}ist.hokudai.ac.jp, ††kawasima@cs.tsukuba.ac.jp,
†††h-inoue@ce.jp.nec.com

あらまし 近年、ハードウェアによるストリーム処理が注目されている。これまでソフトウェアエンジニアによるハードウェアの開発は難しかったが、これが可能になれば処理速度や電力効率の面で大きなメリットがある。本研究ではソフトウェアエンジニアがハードウェアを開発・利用するにはどうすればよいかを検証する。C言語によるハードウェア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てストリーム処理回路を実装・最適化した結果、スループットは最適化前と比べて200倍以上に向上し、CPUと比較した電力当たりのスループットは約50倍となった。しかし依然として最適化にハードウェア開発の知識が求められるため、ストリーム処理において広くハードウェアによる高速化が行われるようになるためには、今後ストリーム処理に適したプログラミングモデルの開発が必要である。
キーワード ストリーム処理，ウィンドウ結合，動的リコンフィギュラブルハードウェア

Writing Window Join Processor in C

Eric Shun FUKUDA†, Hideyuki KAWASHIMA††, Hiroaki INOUE†††, Tetsuya ASAI†, and Masato MOTOMURA†

† Graduate School of Information Science and Technology, Hokkaido University Kita 14, Nishi 9, Kita-ku, Sapporo, Hokkaido, 060–0814 Japan
†† Graduate School of Systems and Information Engineering, University of Tsukuba 1–1–1 Tennodai, Tsukuba, Ibaraki, 305–8573 Japan
††† NEC Corporation 1753 Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa, 211–8666 Japan
E-mail: †{fukuda@lalsie., asai@, motomura@}ist.hokudai.ac.jp, ††kawasima@cs.tsukuba.ac.jp,
†††h-inoue@ce.jp.nec.com

Abstract In the past, there has always been a wide gap between the skills for designing software and hardware. Now that reconfigurable hardware is attracting wide attention as a platform of stream processing, a method that bridges this gap will have a large impact on processing speed and power efficiency. In this paper we look into how a software engineer will develop and utilize a reconfigurable hardware for stream processing. Our examination of a dynamically reconfigurable hardware which features a C-based development environment showed that the throughput improves by 200 times after code optimization, and the power efficiency is 50 times higher than that of a CPU. However, a programming paradigm for stream processing is still needed for wide use of hardware acceleration.
Key words stream processing, window join, dynamically reconfigurable hardware

1. はじめに

ストリーム処理はビッグデータやクラウドコンピューティング時代の新たな情報処理方式として注目を集めている[1]。膨大な情報をリアルタイムに処理するため、従来のデータベース処理ではサーバの並列分散化を行うのに対し、ストリーム処理で

はメモリやディスクの入出力を省くことによって処理を高速化する。これらの方法はこれまで効果を発揮してきたが、更なる高速化への要求があると同時に、増え続ける消費電力を削減する必要がある。特にデータセンタでは電力削減が喫緊の課題となっており、電力の消費が大きい技術に頼って処理の更なる高速化を図るのは困難である。

この問題に対する有効な解決手段として FPGA によるストリーム処理の高速化が盛んに研究されている [2], [6], [9]。一般に、ある目的に対して設計されたハードウェアは、CPU のような汎用ハードウェアよりも電力に対してはるかに高い性能が得られる。しかしそのようなハードウェアには、ソフトウェアエンジニアによる開発が難しいというデメリットがある。ストリーム処理ではソフトウェアエンジニアにより高度に作り込まれたアルゴリズムを用いるため、これは大きな問題である。この問題を解決することが最近の大きな流れとなっている [2], [8]。

本論文の著者のうち数名が開発に携わった Dynamically Reconfigurable Processor (DRP, [4]) は、専用の開発環境上で C 言語による開発を行うことができる。これまで DRP は主に画像処理の分野で [5] ハードウェアエンジニアによる開発が行われてきたため、DRP をストリーム処理に応用することによっていかに上記の問題を解決できるかは興味深い挑戦である。

開発を行うにあたって、我々は Window Join というストリーム処理の要素演算であり単純かつ重要な演算を題材に取り、DRP 上に段階的に処理回路を実装した。これを通して本論文では次のことを明らかにする。

- 最先端の高位合成ツールを使った開発したハードウェアでどの程度の性能が得られるか
- ソフトウェアエンジニアがどのようなハードウェア設計知識を知っておく必要があるか

2. 関 連 研 究

ここ数年、ストリーム処理をハードウェアで高速化する研究が行われている。ある研究では、C 言語と高位合成を用いた実装を行っている [2]。この研究が提案するシステムでは、算術演算や集約などの単純な演算行う C 言語の関数をいくつか定義し、それらを正規表現を用いて並べる。高位合成ツールによって合成された回路はストリーム情報を処理し、それぞれの関数に対応する回路の出力する信号の並びが、与えられた正規表現に対応していれば、システムは信号を出力する。このシステムは FPGA 上に実装され高い性能を示した。

この研究が開発用言語として C 言語を用いていたのに対し、Mueller らは SQL クエリをハードウェアにマッピングする Glacier というシステムを提案した [6]。回路は SQL の各演算に対応する要素回路から構成される。彼らはこのシステムで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、SQL クエリ処理専用ハードウェアも後に別グループにより提案されている [3]。

これらの研究は、用途をストリーム処理に限定した設計フレームワークで回路を FPGA 上に構成することによって、最適化を行うことなく十分高速な回路をソフトウェアから合成できることを示唆している。一方で本研究で用いる DRP は用途を限定したフレームワークを持っていないが、C 言語による開発環境を持っているため、1. で述べたハードウェアのデメリットを克服しており、ソフトウェアエンジニアによるストリーム処理回路の開発は十分可能であると考えられる。

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 [4]。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 [11]。

DRP を使ったシステムの設計は高位合成ツールを用いて行う [10]。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。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 [11]。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。

本研究では市販されている DRP の評価キットを用いた [5]。図 2 は DRP がオンチップ IP コアとして搭載されている評価チップの構成を示す。チップは PCI Express カードにマウントされており（商品名は DRP Express)、開発ツールがインストールされた通常の PC で動作させることができる。

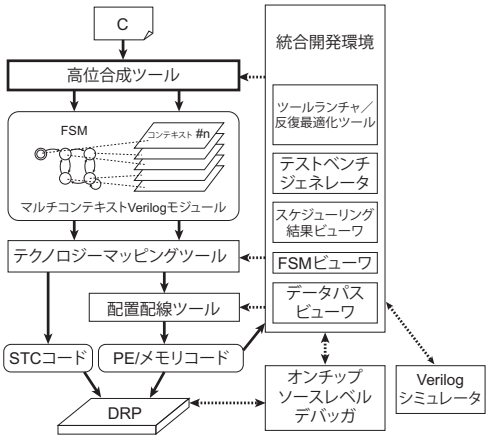

図 1　C コードからハードウェアへの変換の流れ

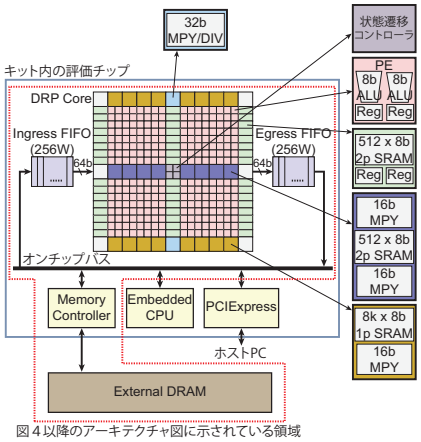

図 2　DRP の構成

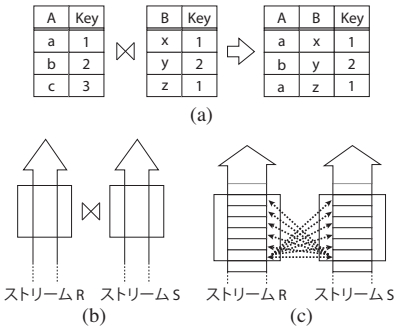

図 3　Join と Window Join

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ゆくので、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も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 [7], [9]。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。

Algorithm 1 ステップ 1 のアルゴリズム

```
loop
  register_R ⇐ r_t ∈ R
  for i ⇐ 0 to N − 1 do
    register_S ⇐ s_{t−i} ∈ S
    if regisger_S = register_R then
      O ⇐ {register_S, register_R}
    end if
  end for
  (Repeat with R(r) and S(s) reversed)
  t ⇐ t + 1
end loop
```

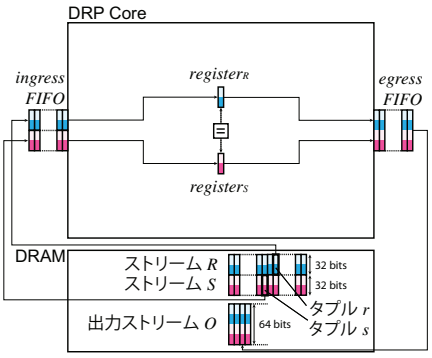

図 4　合成されたハードウェアのアーキテクチャ（ステップ 1）

DRP 上に Window Join 処理回路を実装するに当たって、最も基本的なソフトウェア処理と考えられるプログラムから始める。その後、合成されたハードウェアの非効率な点を分析し、それを改善する修正をプログラムに加え、再度ハードウェアを合成する、という手順を繰り返しながら回路を最適化していく。このようなプロセスを経て、各段階のソースコードを比較することにより、1. に挙げた問いに対する回答を導く。

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) 取り扱うサブ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　ステップ 1: 通常のソフトウェアコード

最初のステップは、CPU で実行することを前提に C 言語で記述する上で最も単純と考えられるプログラムを用いる。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（Algorithm 1）。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 4 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では同じタプルを何度も外部 DRAM から DRP コアに読み込む必要があるため、明かに非効率的である。このアーキテ

Algorithm 2 ステップ 2 のアルゴリズム

```
loop
  for i ⇐ N to 1 do
    w_Ri ⇐ w_R(i−1)
    w_Si ⇐ w_S(i−1)
  end for
  w_R0,S0 ⇐ ingressFIFO
  for i ⇐ 0 to N − 1 do
    if w_R0 = w_Si then
      egressFIFO ⇐ {w_R0, w_Si}
    end if
    if w_S0 = w_Ri then
      egressFIFO ⇐ {w_S0, w_Ri}
    end if
  end for
  t ⇐ t + 1
end loop
```

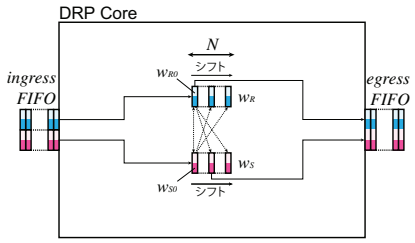

図 5　合成されたハードウェアのアーキテクチャ（ステップ 2）．

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　ステップ 2: サブウィンドウ内タプルのバッファリング

ステップ 1 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め、これを削減することにより性能向上が期待できる。いま、Join を適用する範囲はサブウィンドウの中に限られているため、サブ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。そこでコードのなかに配列 w_R と w_S を宣言し、w_{R_0} と w_{S_0}（新たに入力されたタプル）をそれぞれ w_S と w_R に含まれるタプルと比較する（Algorithm 2）。合成されたアーキテクチャは図 5（本図以降 DRAM の図示を省略）のようになり（シフトレジスタ w_R と w_S が配置されている）、スループットは 17 倍に向上した。

4.4　ステップ 3: 出力バッファへの並列書き込み

サブウィンドウ内で行われる $2N$ 回の比較にはデータや制御の依存性がないことから、$2N$ 個の比較器を用いることによって DRP コア内で並列に比較を行うことができるはずである。ところが図 5 に示したアーキテクチャでは並列に比較が行われていなかった。これは、$2N$ 回の比較により複数の出力タプルが生成される可能性がある一方で、出力 FIFO（$egressFIFO$）には同時に一つしかデータを入力できないことが原因である。

これを解決するため、$2N$ 個の出力バッファを導入し、同時に生成し得る出力タプルをすべて出力バッファに保持できるようにした。一方で、1 回のメインループにつき一つの出力バッファから出力 FIFO にタプルを書き込む（本ステップ以降はアルゴリズムを示さないが、全て C 言語により記述が可能である）。この改善により、並列に行う比較と逐次的に行う $egressFIFO$ への書き込みを切り離すことができる。$egressFIFO$ の帯域が $ingressFIFO$ よりも狭い（タプルの一致率が低い）限りにおいてこのアーキテクチャは有効である。タプルの一致率の問題については 4.9 で議論する。

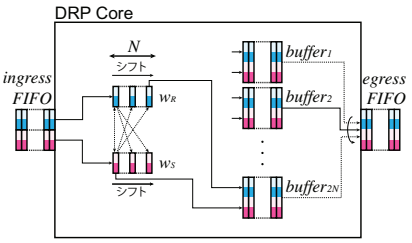

図 6　期待されるハードウェアのアーキテクチャ（ステップ 3）．

この改善によって図 6 に示すようなアーキテクチャが合成されることで大幅に性能が改善されることが予想されたが、実際にはスループットは低下し 8 Mbps となった。これは図 6 のようなアーキテクチャを構成するには DRP コア上にレジスタやメモリの数が足りなかったためである。

4.5　ステップ 4: マッチテーブル

Window Join を並列に行う上でレジスタとメモリが不足しやすいことがわかったので、少ないリソースでバッファリングを行う仕組みを作る必要がある。一般的にメモリはレジスタに比べて、並列にアクセスすることができないという欠点がある反面、大容量のデータを保持できるため、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 に示すようにアーキテクチャを実装した。$w_{R(S)}$ はここで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 と w_S 間の比較結果は、$storageR(S)$ 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。

図 6 に示すアーキテクチャでは、複数の出力バッファで同じタプルのコピーを保持する場合があるなどリソースの利用が非効率的だったが、本ステップではそのようなことがないため、タプルのバッファリングに必要なリソース量を大幅に削減することができた。そして $2N$ の比較を並列に行うことができるようになり、スループットは 103Mbps まで上昇した。

4.6　ステップ 5: ブロックデータ転送

レジスタとメモリの不足問題が解決されたので、次は明らかなボトルネックとなっている DRP コアへの入力を改善する。これまでのところ、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る。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。タプルのブロックの大きさは $ingressFIFO$ と同じ大きさとした。スループットは前

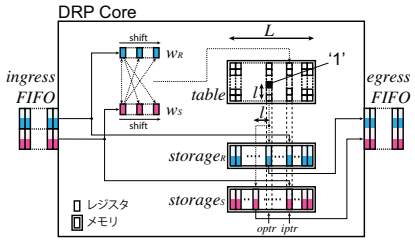

図 7　合成されたハードウェアのアーキテクチャ（ステップ 4）

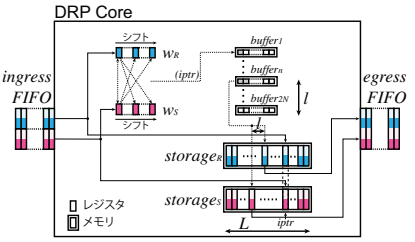

図 8　合成されたハードウェアのアーキテクチャ（ステップ 8）

ステップと比べ 3 倍に向上した。

4.7　ステップ 6: 並列テーブル参照

ステップ 4 とステップ 5 では、*table* の列は 32 ビットのビットベクタであった。しかしそのデコーディングは 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っている。この問題を解決するために *table* を 8 ビット幅に分割した。こうすることにより、ビットベクタのエンコードとデコードがを 4 並列で行えるようになり、遅延が減少した。結果として、スループットは 83%向上した。

4.8　ステップ 7: 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高位合成ツールの機能であるが、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た。これによりスループットは 87%向上した。

4.9　ステップ 8: 低一致率最適化

明示してこなかったが、ステップ 3 はタプル間のキーの一致率が高い場合に正常な処理ができなくなるのに対し、ステップ 4 からステップ 7 までは一致率によらず処理が可能であった。これは、$storage_{R(S)}$ が一杯になる（$iptr = L$）と、$optr$ が L となるまで出力処理処理のみを実行し $storage_{R(S)}$ が空になるようにしていたためである。しかし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と仮定できるため、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 8 に示すようなアーキテクチャを実装した。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。入力・比較処理はフォールディングし、各イテレーションがクロック毎に開始されるようになった。この結果スループットは 38%向上し、1.47Gbps となった。

5.　考　　察

図 9 はスループットの改善をグラフにしたものである。ステップ 8 はステップ 1 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 のプログラムを Intel Core i5 (2.5GHz) の CPU で性能を測定したところ、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 50 倍であった。これはリコンフィギュラブルハードウェアの電力効率の高さを示している。(CPU のコードも最適化することで処理性能が改善するが、多くの場合 10 倍かそれ以下の改善にとどまる。)

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。本研究では高位合成ツール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高位合成ツールでは、パイプライン化したい `for` ループの直前にコメントの形でパラメータを書き込むだけで済む。

一方で、高位合成ツール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から、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」を提案する。

（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もっともボトルネックとなりやすい点であり、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ための方法を見つける必要がある。(ステップ 2)

（2） バッファリングの視点： 入力・中間・出力データをどのようにバッファリングするかは処理の並列度に大きな影響を与える。(ステップ 3 と 4)

（3） リソース量の視点： 並列に実行できる処理を記述したとしても、処理やバッファリングを行う上でリソース量がその実現を制限してしまう。(ステップ 4)

（4） 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、ソースコードを効率よくターゲットハードウェア上で実現する助けとなる。(ステップ 6)

（5） ループの視点： ソフトウェアでも同様だが、ハードウェア合成においてループは並列化することで効率を大きく向上させることができるポイントである。

誤解を避けるために繰り返すが、本研究でハードウェア合成

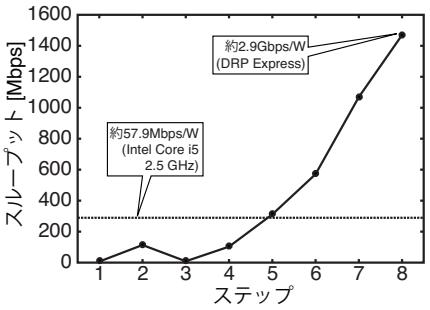

図 9　各ステップにおけるスループットの向上

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否かである。Window Join の例を通して、我々はそれを不可能であると考える。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をその開発環境を用いて Window Join を実装してみるというアプローチをとった。度重なる最適化の結果、以下の結論が導かれた。1) 最先端の高位合成ツール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) 最適化により二桁スループット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、リソースタイプ、ループ）の視点」をソフトウェア開発者が持つことが重要である。

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古田浩一郎氏と藤井太郎氏、犬尾武氏、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, 2006.

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

[3] T. Miyoshi, H. Kawashima, Y. Terada, and T. Yoshinaga. A coarse grain reconfigurable processor architecture for stream processing engine. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[4] M. Motomura. A dynamically reconfigurable processor architecture. *Microprocessor Forum*, 2002.

[5] M. Motomura. STP engine, a C-based programmable HW core featuring massively parallel and reconfigurable PE array: Its architecture, tool, and system implications. In *Proceedings of the COOL Chips XII*, 2009.

[6] R. Mueller, J. Teubner, and G. Alonso. Streams on wires - a query compiler for FPGAs. *Proceedings of the VLDB Endowment*, Vol. 2, No. 1, 2009.

[7] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga. Design and implementation of a merging network architecture for handshake join operator on FPGA. In *2012 IEEE 6th International Symposium on Embedded Multicore SoCs (MCSoC)*, 2012.

[8] M. Takagi, T. Takenaka, and H. Inoue. Dynamic query switching for complex event processing on FPGAs. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[9] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[10] T. Toi, T. Awashima, M. Motomura, and H. Amano. Time and space-multiplexed compilation challenge for dynamically reconfigurable processors. In *IEEE International Midwest Symposium on Circuits and Systems*, 2011.

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