Fortranによる高性能計算のハードウェア化と高位合成ツールFortRock

山下 貴大[1]　五十嵐 雄太[1]　中條 拓伯[1]

概要：
近年の GPU や FPGA の発展に伴い，高性能計算において，ハードウェアアクセラレーションによる高速計算が行われるようになった．FPGA 上での回路設計には，HDL による記述とともに C や Java といった高級言語を用いて，高位合成ツールによる回路生成を行い，複雑なアルゴリズムを記述できるようになった．しかしながら，高性能計算にはこれまで Fortran で記述されたライブラリ，パッケージが受け継がれ，今もなおその改良が行われており，現状の高位合成ツールを利用するには，Fortran で記述されたソースプログラムを C や Java に書き換えて合成を行うこととなる．本研究では，Fortran で記述された流体解析アプリケーションを手動で Java のソースに書き換え，高位合成ツール JavaRock-Thrash を用いて Verilog HDL に変換し，性能を評価する．次に，Fortran のソースプログラムを直接 Verilog HDL に変換する高位合成ツール FortRock について紹介し，これまでの高性能計算のソフトウェア資産をハードウェア化するための可能性を示す．

キーワード：高位合成，FPGA，Fortran，アクセラレーション，高性能計算

Hardware Acceleration for High Performance Computing with Fortran and a High-Level Synthesis Tool FortRock

Takahiro Yamashita[1]　Yuta Ikarashi[1]　Hironori Nakajo[1]

Abstract:
In recent years, with growth of using GPU and FPGA, hardware acceleration in simulation of Computational Fluid Dynamics (CFD) is focused. In designing circuits with an FPGA, high level synthesis with a C or a Java language as well as an HDL is utilized to describe complex algorithms. However, there still exist a large amount of heritage of Fortran programs in high performance computing also still have been improved. For Fortran programs, we have re-written them into Java source codes and have converted into a Verilog HDL codes with our HLS called JavaRock-Thrash. Due to experiences and knowledge, we have been developing FortRock which converts a Fortran source code into a Verilog HDL code directly to utilize software heritage in high performance computing.

Keywords: High Level Synthesis, FPGA, Fortran, Acceleration, High Performance Computing

1. はじめに

近年，マルチコア，メニィコアプロセッサや GPU の登場に伴い，高性能計算がデスクトップ計算機上でも行われるようになった．流体力学の分野においても，計算機を用いた数値流体力学（Computational Fluid Dynamics：CFD）が身近なものとなったが，シミュレーションに膨大な時間を要するといった問題がある．これに対し，アルゴリズムに工夫を凝らす数学的なアプローチのほかに，アクセラレーション技術が注目を集めている．その中で，Field Programmable Gate Array (FPGA) を用いたハードウェア (HW) アクセラレーションに期待が寄せられている．

FPGA 上での回路設計では HW 記述言語 (HDL) を用いて設計する方法が主流となってはいるが，C や Java のような高級言語を用いた高位合成により回路を生成する方法も広まりつつある．これより，HDL を使用するよりも，設計者が記述するソースコードの量が少なく，設計負担も軽減され，複雑なアルゴリズムを記述しやすいという利

[1] 東京農工大学
Tokyo University of Agriculture and Technology

点もある．現在，C をベースとする高位合成ツールとしては，ImpulseC[2]，CyberWorkBench[3] や LegUp[4] が挙げられ，Java ベースには，JHDL[5] や Lime[6]，JavaRock[7] がある．

しかしながら，高位合成ツールによって作成された回路は，従来の HDL によるものや IP コアと比べ，回路規模や動作周波数の面で不利となる場合がある点には注意が必要である．それでもなお，その欠点を凌駕するような並列性を発揮できれば，アクセラレータとしての可能性には大いに期待できる．

そこで我々は，CFD を中心に Fortran により記述された高性能計算プログラムを Java に移植し，高位合成ツール JavaRock-Thrash[1] を用いて Verilog HDL に変換し，HW アクセラレーションの性能評価を行った．その，Fortran から Java へのソースコード変換作業は，手作業で行わなければならず，高級言語による HW アクセラレーションの実現において，高位合成ツールのもつ，“開発期間の短縮”という目的に反することとなる．Fortran のソースから C のソースに変換するツールはあるにはあるが，こういったツールを用いた場合にも，高位合成ツールによっては文法などに制約がある場合が多く，その修正・変換に手作業を要することとなる．

また，Fortran のプログラムには数多くの高度な最適化技術が施されており，これもまた資産を再利用したいという要望が起きる要因の一つである．しかしながら，FPGA においてこの最適化を考慮した回路を設計するためには，プログラマ (かつ HW 設計者) に対し，HW に関する知識と，高位合成ツールのコードの出力方法などへの経験を要求することとなる．このハードルは極めて高いものと考えられ，これもまた，高位合成ツールのもつ “低い習得コスト” という利点に反する．

そこで，Fortran のソースプログラムを入力とし，Java ベースとした高位合成ツール JavaRock-Thrash(JRT) で論理構成可能なコードを生成するツール F2JRT (Fortran to JavaRock-Thrash) を実装した．F2JRT が JRT の文法制限などを考慮した Verilog HDL コードの出力を行うことで，Fortran から FPGA への移植作業の際に，JRT に関する習得コストを下げることができると考えられる．これにより，Fortran により記述された科学技術計算の過去の資産の再利用が容易になると考えた．しかしながら，Fortran には様々な記述方法があり，また，標準入出力や組み込み関数など，HDL に一意に対応するコードが必ずしも存在するとは限らないため，そこは依然として手作業での修正が必要となる．

以上の流れから，Fortran のソースから直接 Verilog HDL コードに変換する高位合成ツール FortRock の開発に着手することとなった．本論文では，Fortran のソースから JRT Java に自動変換を行うツール F2JRT について述べ，さらに高位合成ツール FortRock の設計方針と現状について報告する．

2. 関連研究

CFD プログラムを FPGA を用いて HW 化する際の課題が文献 [8] で挙げられている．これは独立行政法人宇宙航空研究開発機構 (JAXA) が開発した流体解析ソフトウェア UPACS[9] を対象に検証を行って得られたものであり，プログラムを処理ごとに分割し，Fortran のソースコードでおよそ 2,000 行となる 8 つの処理に対し HW 化を行った結果から以下に示す 8 つが課題として提示された．

(1) 大量のデータ入力
(2) 回路規模と実装
(3) メモリスケジューラの実装
(4) ホスト計算機とのインタフェース
(5) 浮動小数点演算器のチューニング
(6) CFD 研究者による FORTRAN プログラムの FPGA 回路化
(7) 実行計算機を意識したソースプログラムのチューニング
(8) IP コアの無い演算処理の実装

(1)〜(5) については HW に関する課題であり，(6) と (7) については設計手法に関する課題である．(6)，(7) では回路を設計する際には専用の知識を必要とし，CFD 研究者が回路設計のための知識を学ぶ必要があると述べている．

CFD における HW アクセラレーションの研究として他に文献 [10] がある．この研究では UPACS と同様に JAXA が開発した流体解析ソフトウェア FaSTAR[11] を対象に HW アクセラレーションを行っている．前述の 8 つの課題のうち「(3) メモリスケジューラの実装」を行っている．FaSTAR を解析した結果，中間結果を一時的にメモリに保存する必要がある箇所でメモリ上の同一のアドレスに連続して複数回アクセスすることがあり，ストールが多数発生することがわかった．したがって，その個所をハードウェア化する際に，ストール回数を削減するための Out-Of-Order 実行をする機構を作成することでハザードを解消した．メモリスケジューラを実装した上で HW 化を行ったところ，FPGA での実行速度は CPU での実行速度の 2.35 倍速いという結果が得られた．

これらはいずれも，UPACS，FaSTAR に収納された Fortran のソースコードからアルゴリズムを解析し，HDL により記述したものであり，その実装には多大な労力が投入されたものと思われる．CFD といった高性能計算に高位合成ツールを利用した研究については，過去の資産の継承の上でも重要であり，今後注目されていくものと思われる．

2.1 性能向上への方向性

JRT で作成した回路とソフトウェアとで処理時間を比較

した場合，目標とするハードウェアアクセラレーションを実現できなかった．その原因としては十分な並列度を発揮できなかったことが考えられる．並列度を高めるにループの展開開数を増やせばパイプラインの段数は増え，ループ回数が減るため，全体としての処理サイクル数は減少する．しかし，それに伴った回路の複雑化によるリソース量の不足という問題が発生し，さらに，回路が複雑化することで論理合成に要する時間が長大するという問題が発生する．

これらの問題は，JRT を用いて回路を作成した場合，並列度を高める方法がループ展開しか現状では存在しないことが主な要因となっている．展開数を増やしたことにより回路規模が増大した回路の論理合成は，数日かかっても完了しなかった．解決策として考えられるのは，パイプラインスケジューリングを行い，連続的なデータ入力を実現することが考えられる．これにより，ループ展開を用いてパイプラインの段数をある程度増やした上でループのパイプラインへ連続的なデータの入力ができれば，アクセラレーションとしての効果を得ることができる．従って，今後は特定のパイプラインへ連続的にデータを入力するパイプラインスケジューリングを考案する必要がある．

3. Fortran から Java への変換ツール F2JRT

3.1 設計方針

F2JRT とは，Fortran で記述されたプログラムを，JavaRock-Thrash でコンパイル可能なものに変換するプログラムである．これを用いることにより，Fortran で記述された数値計算プログラムを FPGA 上で再利用が容易になると考えられる．JRT は，Java をベースとした高位合成ツールであるため，文法には一部制限があるため，F2JRT もその文法によって制限がかかる．例えば，Java では GOTO 文がサポートされていないため，Fortran において GOTO 文が使用されていると，その箇所は変換不可能な箇所として出力される．F2JRT における文法の制限を以下に示す．

- GOTO 文をサポートしない
- 組み込み関数をサポートしない
- 配列のサイズに変数を指定することはできない
- 記述方法を制限する

まず，GOTO 文をサポートしないのは，前述したとおりであり，これを回避する方法としては，GOTO 文の移動先のプログラム行すべてを，GOTO 文が記載された位置に記述するという方法で対応できる．

次に，組み込み関数をサポートしないのは，組み込み関数がどのように実装されているか不明であり，これを回避する方法としては，組み込み関数を自分で定義 (Fortran で実装) し，組み込み関数の代わりにそれを呼び出すように変更するという方法が考えられる．

配列のサイズに変数を指定することはできないのは，JRT の制限によるものである．JRT では，配列を宣言する位置がフィールドに制限されており，また，フィールドで宣言する際に，同時に配列のインスタンス化 (サイズ指定) を行うという制限もあるため，配列のサイズに変数を使用することができなくない．

最後に，記述方法の制限については，F2JRT の実装を簡単化するためである．Fortran では，同じ意味でも多種多様の記述方法 (文法) が許されるため，F2JRT の変換部分をその文法に対応することで，この問題を解決することができると考えられる．

4. 高位合成ツール FortRock

本章では，Fortran から Verilog HDL を出力する高位合成ツール FortRock について述べる．高位合成ツール FortRock は，Fortran ソースプログラムから Verilog HDL コードを出力する高位合成ツールである．Fortran は，長期にわたり科学技術計算に用いられてきたプログラミング言語で，そのプログラム資産は大量にあり，それを再利用して HW 化したいというニーズが存在する．

4.1 FortRock の設計方針

FortRock は，単に Fortran による HW 設計環境を提供するということではなく，FPGA による HW アクセラレータの実現を目標とする高位合成ツールである．FPGA 設計では，回路規模がコストに直接影響を及ぼすため FortRock では，回路の性能を制約として，回路規模が最小となることを目標とした．

4.2 FortRock のコンパイルフロー

図 1 に FortRock によるコンパイルフローを示す．FortRock は，入力ファイルとして Fortran のソースファイルと，高位合成に用いる制約条件などを記述した config ファイルの 2 つを入力として受け取る．受け取った Fortran のソースファイルから，Verilog HDL のソースファイルを出力する．config ファイルは，emacs の Org-mode の形式 (.org) で記述される．フロントエンドには Dragonegg(gfortran) を用いた．Dragonegg は LLVM のフロントエンドの一つで，GCC(GNU Compiler Collection) により，LLVM で使用する中間表現である LLVM IR(LLVM Intermediate Representation) を出力する．LLVM とはコンパイラ基盤であり [12]，それによって提供される様々な機能 (API) を用いることで，再利用可能なモジュールを用いたコンパイラの設計が可能となる．出力される Veriog HDL ファイルは，ハードウェア化可能な形式で記述されているため，論理シミュレーションや論理合成で利用できるものとなっている．

4.3 FortRock のコード変換規則

FortRock では，Fortran の SUBROUTINE を 1 つのモ

ジュールとして出力する．そのため，FortRock は，1 ソースファイルごとに 1 つのモジュールを定義しているものと仮定して処理を実行する．また，モジュールには，予め 2 入力と 1 出力が予約されている．2 つの入力 “res” と “clk” は，それぞれモジュールの初期化 (処理の実行開始) とクロックを入力する．出力 “fin” は，モジュールの処理が完了した (モジュールのステートマシンが終了状態になった) ときに 1 を出力する．

図 2 に CALC という SUBROUTINE が Fortran で定義されている記述例を示す．その変換後のブロック図を図 3 に示す．この例では，パラメータには A，B，C，RET の 4 つが定義されているが，SUBROUTINE の入力は複数のパラメータ，出力は末尾に記述されたパラメータを用いる．また，config ファイルで指定することによって，複数の出力を定義することもできる．これより入出力ポートを明示的に指定することが可能となる．CALC モジュールには，A，B，C，RET の入出力信号に加えて，前述の通り “res”，“fin” というそれぞれ入力，出力信号が追加で定義される．

SUBROUTINE がモジュールとして扱われるため，親ルーチンでの SUBROUTINE 呼び出しは，Verilog ではサブモジュールの接続として定義される．したがって，モジュールをテストするためのテストベンチを，Fortran のテストコードとして記述することができる．これにより，1 ソースファイルで HW とソフトウェアの両方の記述ができる．

図 4 にサブルーチン内部で別のサブルーチンを呼び出す例を示す．また，その変換後のブロック図を図 5 に示す．この例では，TOP モジュールが SUB モジュールを利用するという状態を，Fortran を用いて記述した．Fortran ソースコードでは，通常通り関数として SUB を呼び出している．実際の Verilog ソースコード上では，TOP モジュールに SUB モジュールが含まれる形になり，“fin”，“res” 信号を用いて SUB モジュールを呼び出している．呼び出し処理は FortRock が自動的に記述する．

4.4 スケジューリング/バインディングアルゴリズム

高位合成ツールの出力する回路の性能は，スケジューリング/バインディングアルゴリズムに大きく左右される．スケジューリングとは，各 Data Flow Graph (DFG) の実行サイクルを決定する処理である．FortRock では，LLVM IR から CDFG (Control DFG) を作成し，CDFG レベルでのスケジューリング，最適化を行う．CDFG とは制御依存とデータ依存の両方を一つのグラフに表したものである．

スケジューリングアルゴリズムは，資源制約スケジューリングと時間制約スケジューリングに大別される．また SoC，FPGA では，回路面積がコストに直結するため，高

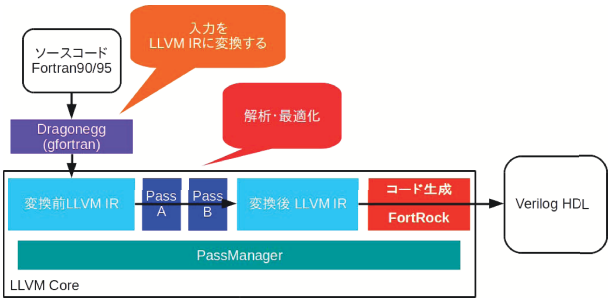

図 1 FortRock のコンパイルフロー
Fig. 1 Compilation Flow of FortRock

```
0 SUBROUTINE CALC(A, B, C, RET)
1 INTEGER A, B, C, RET
2
3 RET = A * B + RET
4
5 RETURN
6 END
```

図 2 SUBROUTINE 記述例 (Fortran ソースコード)
Fig. 2 SUBROUTINE description example (Fortran)

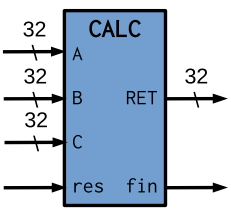

図 3 SUBROUTINE 変換例 (ブロック図)
Fig. 3 SUBROUTINE translation example (Block diagram)

```
1 SUBROUTINE TOP(A, B)
2 INTEGER A, B, S_OUT
3
4 B = A + SUB(A, S_OUT)
5
6 RETURN
7 END
```

```
1 SUBROUTINE SUB(C, D)
2 INTEGER C, D
3
4 D = C * 2
5
6 RETURN
7 END
```

図 4 SUBROUTINE の演算器化 (記述例)
Fig. 4 SUBROUTINE modulation example(description)

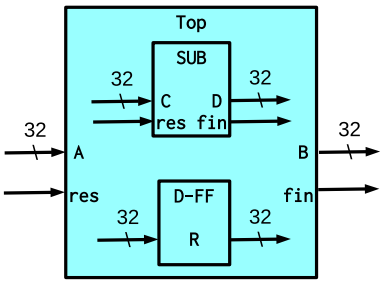

図 5 SUBROUTINE の演算器化 (ブロック図)
Fig. 5 SUBROUTINE modulation example(Block diagram)

位合成には性能を維持したまま回路面積を最小にすることが求められる．そこで FortRock では，リソースの共有によるマルチプレクサの増加に伴う回路規模の増加を抑えるために，スケジューリングアルゴリズムに資源制約スケジューリングを採用した．

スケジューリングアルゴリズムには，ASAP(as soon as possible)[13] や ALAP(as late as possible) といったものが存在する．ASAP はできるだけ早く，ALAP はできるだけ遅く計算資源を割り当てるスケジューリングであるが，どちらも資源制約, 時間制約ともに考慮しない．そこで，よりよい演算器割り当てを求めるために，FortRock ではヒューリスティック解法であるリストスケジューリング [14] を採用した．リストスケジューリングは，各演算に対して演算の優先順位を計算し，各ステップで割り当て可能な演算資源を，その優先順位を基準に割り当てる．優先順位の計算には，演算の結果が後の演算で利用される回数が多いものを優先する等の評価関数を用いる．その他に，ASAP, ALAP から計算される自由度等を用いることもできる．

FortRock では，config で与えられた計算機資源の数を基にリストスケジューリングを行う．実際には，config ファイルで指定されたリソース数で回路を出力するのではなく，指定されたリソース数内で可能なスケジューリングを行い，処理サイクル数，回路規模などを考慮していくつか出力する．これにより FortRock のいくつかの出力から，実条件に最適となる回路を選択することができる．

同様に，config ファイルでレジスタの最大共有ステート数を指定できるようにした．これにより，レジスタの共有ステート増加に伴うマルチプレクサの増大を抑えることができ，結果的に回路規模の増加や動作周波数の低下を防ぐことができる．

4.5 FortRock の現状と今後

FortRock では，現状では単純に Fortran ソースファイルから Verilog HDL を出力する機能のみが実装されている．実際に FortRock を用いて Fortran のソースファイルを Verilog HDL に変換する例を示す．Fortran での記述を図 6 に，それを FortRock で Verilog HDL に変換したものを図 7 に示す．

用いたプログラムは，2 入力から最小公倍数を用いるプログラムである．LLVM IR レベルでの最適化を施しているため，元の Fortran のプログラムと一対一にコードが対応しているわけではない．リセット信号がイネーブルになると，リセット処理が行われ，モジュールの動作が開始する．その後，モジュールの終了状態のステートに至るまで処理を続け，終了状態になると fin 信号をイネーブルにすることで，呼び出し元に処理が完了したことを伝える．

実際の高位合成ツールとして評価する場合には，これに加えてバインディング/スケジューリングアルゴリズムの

```
      SUBROUTINE LCM(I, J, ret_lcm)
      INTEGER I, J, IR1, IR2, IR, ret_lcm

      IF(I < J) THEN
         IR1 = J
         IR2 = I
      ELSE
         IR1 = I
         IR2 = J
      ENDIF

      IR = IR1 - (IR1/IR2) * IR2

      DO WHILE(IR>0)

         IR1 = IR2
         IR2 = IR
         IR = IR1 - (IR1/IR2) * IR2
      ENDDO

      ret_lcm = I*J/IR2

      RETURN
      END
```

図 6 最小公倍数例 (Fortran ソースコード)
Fig. 6 LCM program example (Fortran source code)

```
module lcm_opt(clk, res, fin, reg_i, reg_j, reg_ret_lcm);
  input clk, res;
  output fin;

  reg fin;

  input  [31:0]  reg_i, reg_j;
  output [31:0]  reg_ret_lcm;

  reg          reg_tmp2, reg_tmp4, reg_tmp8;
  reg [1:0]    prev_state, current_state;
  reg [31:0]   reg_ret_lcm, reg_tmp, reg_tmp1, reg__,
               reg__1, reg_tmp3, reg_tmp6, reg_tmp7,
               reg_tmp5, reg_tmp9, reg__lcssa, reg_tmp10;

  always @(posedge clk)
  begin
  if(res)
    begin
      fin = 1'b0;
       reg_ret_lcm = 32'b0;
       prev_state = 2'b0;
       current_state = 2'b0;
    end // if res
  else if(fin == 1'b0)
    begin
    case (current_state)
      2'd0:
        begin
          reg_tmp = reg_i;
          reg_tmp1 = reg_j;
          reg_tmp2 = (reg_tmp < reg_tmp1);
          reg__ = (reg_tmp2 == 1'b1) ? reg_tmp1 : reg_tmp;
          reg__1 = (reg_tmp2 == 1'b1) ? reg_tmp : reg_tmp1;
          reg_tmp3 = reg__ % reg__1;
          reg_tmp4 = (reg_tmp3 < 1);
          prev_state = current_state;
          current_state = (1'b1 == reg_tmp4) ? 2'd2 : 2'd1;
        end
      2'd1:
        begin
          case(prev_state)
            2'd1 : reg_tmp5 = reg_tmp6;
            2'd0 : reg_tmp5 = reg__1;
          endcase
          case(prev_state)
            2'd1 : reg_tmp6 = reg_tmp7;
            2'd0 : reg_tmp6 = reg_tmp3;
          endcase
          reg_tmp7 = reg_tmp5 % reg_tmp6;
          reg_tmp8 = (reg_tmp7 < 1);
          prev_state = current_state;
          current_state = (1'b1 == reg_tmp8) ? 2'd2 : 2'd1;
         end
      2'd2:
        begin
          case(prev_state)
            2'd0 : reg__lcssa = reg__1;
            2'd1 : reg__lcssa = reg_tmp6;
          endcase
          reg_tmp9 = reg_tmp1 * reg_tmp;
          reg_tmp10 = reg_tmp9 / reg__lcssa;
            prev_state = current_state;
            current_state = 2'b11;
        end
      2'b11:
        begin
           fin = 1'b1;
       end
      endcase
    end // if fin
 end // always
endmodule
```

図 7 変換例 (Verilog HDL コード)
Fig. 7 Converted example (Verilog HDL code)

拡充が必要となる．また，開発者をサポートする機能の実装が必要になると考えている．具体的には，以下のような機能である．

- 最適化アルゴリズムの拡充
- プログラムの初期状態と最適化後の CDFG を出力する機能
- スケジューリング後にバインディングに関するパラメータを指定する機能
  - 演算器のビット幅を任意に指定できる機能 (計算する可能性のある値の範囲の指定)

最適化アルゴリズムの拡充は，ループ展開や TreeHeightReduction などの回路の性能に影響を及ぼす最適化機能の実装である．また，これらを適宜自動的に選択することにより，回路開発者にいくつかの回路を出力できるようにする．

CDFG を出力機能は，使用した CDFG と最適化後の CDFG を出力する．出力形式は DOT 言語を用いる．DOT 言語は Graphviz でグラフを描画するために用いられる言語で，グラフを表現することができる．Graphviz は AT&T 研究所が開発した，オープンソースのツールパッケージである．なお，出力される CDFG は LLVM IR レベルではなく，演算子，レジスタを頂点に，データフローを辺にもつ RTL のグラフを想定している．

次に，スケジューリング後にバインディングに関するパラメータを指定する機能については，スケジューリング後に変更の余地のあるパラメータ (計算精度や考えられる値の範囲 (ビット幅)) を開発者が FortRock に与えるものである．演算器やレジスタのビット幅の最適化は回路規模の最小化につながり，動作周波数の向上にも寄与する．予め値を指定することも可能であるが，その場合開発者の自由度が高いため，その時に指定した値に最適化の結果が影響を受ける可能性が考えられる．そこで FortRock では，一度スケジューリングを行った後にパラメータを指定することを可能にすることで，開発者にビット幅の指定が最適化にどのように影響を及ぼすかをある程度提示することが可能になる．また，ビット幅 (もしくは計算結果の値の範囲) を指定した場合，FortRock が自動的にすべての演算資源のビット幅を変更する．

5. まとめ

本論文では，Fortran で記述された科学技術計算プログラムを高位合成により，HDL による設計より簡便に設計できる方向性として，Java ベースの高位合成ツールで利用可能な形にソースコードを変換するツール F2JRT について示した．そして，Fortran のソースプログラムから直接 Verilog HDL に変換する高位合成ツール FortRock について説明した．

FortRock は，LLVM の技術を利用して実装されており，FortRock のフロントエンドには Dragonegg(gfortran) を用いて，FortRock は LLVM のバックエンドとして実装した．今後の課題としては，高位合成ツールとして必要となる種々の機能の実装があり，その実装の後，さまざまな Fortran による科学技術計算プログラムの HW 化に取り組んでいきたい．

謝辞 本研究の一部は，文部科学省特別経費「持続可能社会にむけた知的情報空間技術の創出」及び JSPS 科研費基盤研究 (C) 25330067 による支援を得た．ここに記して感謝する．

参考文献

[1] 小池恵介，三好健文，船田悟史，中條拓伯: JavaRock-Thrash の実装，組込みシステムシンポジウム (ESS2013) 論文集, pp.41-48, (2013.10).
[2] Impulse accelerated technology: http://www.impulseaccelerated.com/.
[3] CyberWorkBench: http://jpn.nec.com/cyberworkbench/.
[4] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S. and Czajkowski, T.: LegUp: high-level synthesis for FPGA-based processor/accelerator systems, *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pp. 33–36 (2011).
[5] JHDL: http://www.jhdl.org/.
[6] Auerbach, J., Bacon, D. F., Cheng, P. and Rabbah, R.: Lime: a Java-compatible and synthesizable language for heterogeneous architectures, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 89–108 (2010).
[7] JavaRock: http://javarock.sourceforge.net/.
[8] 藤田直行: 実用 CFD コードの FPGA 回路化における技術課題の実験的検討, 第 23 回 数値流体力学シンポジウム講演論文集, E2-3 (2009).
[9] 山本一臣：並列計算 CFD プラットフォーム UPACS について，航空宇宙数値シミュレーション技術シンポジウム'99 講演集 (1999).
[10] 公之赤嶺, 健太田舎片, 保範長名, 直行藤田, 英晴天野：Fastar における流束の面積分計算高速化のための Out-Of-Order 機構 (科学技術計算), 電子情報通信学会技術研究報告. RECONF, リコンフィギャラブルシステム, Vol.111, No.31, pp.73–78 (2011.5).
[11] 橋本敦, 村上桂一, 青山剛史, 菱田学, 大野真司, 坂下雅秀, ラフールパウルス，佐藤幸男：高速流体ソルバ FaSTAR の開発, 第 42 回流体力学講演会/航空宇宙数値シミュレーション技術シンポジウム 2010 (2010).
[12] The LLVM Compiler Infrastructure, http://www.llvm.org/
[13] Giovanni De Micheli: Synthesis and Optimization of Digital Circuit, McGraw-Hill Higher Education (1994).
[14] M. J. M. Heijligers and J. A. G. Jess: High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques. *Proceedings of the ASP-DAC95/CHDL95/VLSI95. Asia and South Pacific Design Automation Conference. IFIP International conference on Computer Hardware Description Languages and their Applications. IFIP International Conference on Very Large Scale Integration*, pp. 56–61 (1995).