LLVM ベースの高位合成向けモデルのソース・コンパイラ：StoS を用いた高位設計フロー

立岡真人[†1]　呉政訓[†1]　金子峰雄[†1]

現在の EDA 業界で利用される高位合成ツールは入力モデルコードと合成制約に従った動作合成が可能であるが、入力モデルコード記述から最適化された RTL を得るには入力コードの最適化が必要である。また、入力モデルコードを高位合成ツールに制限されたコードに書き換えることが必要であり、未だに人手によるコード書き換えが必要となって、設計期間の短縮にはなっていない。このような現状に鑑み，コード記述の制約に整合するコード生成やコードの最適化を自動化できるソース・コンパイラ（Source-to-Source:StoS）と EDA の高位合成ツールのツールチェーンによる高位設計フローを提案する。本稿では再帰関数を使ったソースを StoS により再帰呼出除去された C ソースを生成し、従来の EDA 高位合成ツールを用いて回路合成する高位設計フローを紹介する。

High Level Synthesis Design Flow with Source Compiler based on LLVM

MASATO TATSUOKA[†1]　JUMGHOON OH[†1]
MINEO KANEKO[†1]

The high level synthesis tool used in the current EDA industry can perform behavioral synthesis according to input model code and synthesis constraints, but input code optimization is required to obtain optimized RTL from input model code description. In addition, it is necessary to rewrite an input model code to a restricted code which is synthesizable by a high level synthesis tool. The manual code rewriting is still required, and it prevents us from shortening the design period. Considering such a problem, we have developed a high level synthesis design flow incorporating a source compiler (source-to-source: StoS) with a tool chain of EDA tool(high level synthesis). The role of StoS is to automate code re-generation and code optimization considering code description constraints for synthesizability. As a synthesis example, the removal of recursive function calls and C code re-generation in StoS followed by circuit synthesis using a conventional EDA high level synthesis tool is demonstrated.

1. はじめに

近年、LSI の設計規模が大きくなり、手設計による RTL では市場投入の時間的制約（Time-to-Market）を満たせない状況になっている。そのため半導体メーカーあるいはセットメーカでは RTL の設計時間を要するマニュアルによる設計手法から抽象度の高い C/C++/SystemC などの言語による高位合成を用いた設計手法を採用する事例が多くなっている。

高位合成導入のメリットは抽象度の高い言語を用いることで正確な C モデルを作成し、C モデルと同じ機能の RTL を短時間で合成できることである。また使用するプロセス・テクノロジーにマッピングし、要求するレイテンシ、スループット、面積を満たす RTL を生成することができる。

しかし、現状の高位合成ツールは入力モデルの記述に従った動作を RTL へ合成するため、設計者の要求に合致した RTL を生成するにはそのような RTL が合成されるように意図した入力コードが必要になる。このコーディング技術は、人の RTL 設計の経験やスキルに依存しており、さらに高位合成ツール依存の記述もあり、最適な RTL を生成するのは未だに難しい。

一方、高位合成ツールは入力コードが合成にかかるコード記述であれば RTL を生成する。しかし、その RTL は面積やスループット、電力などが良くない例が多い。通常、要求を満たす RTL を得るには入力コードの書き換えや要求する RTL が得られように高位合成ツールをコントロールしなければならない。高位合成ツールは入力コード記述の範囲で合成制約を指定した RTL に変換することができる。コード記述から最適化された RTL を得るには入力コードの最適化が必要である。また、入力コードの書き換えでは、通常のソフトウエアで使っている再帰呼出などの記述やメモリ・アロケートなどの関数などを書き換える必要がある。

本稿では、上記の課題を解決するために入力ソースコードを高位合成ツール向けの入力コードに変換するソース・コンパイラ（StoS）を用いたデザインフローを提案[1]する。実験では高位合成できない再帰呼出関数を入力コードとして用いた実行例を説明し、本アプローチの可能性について述べる。

2. 従来のアプローチ

2.1 従来の設計フロー

図 1 に高位合成を用いた従来フローを示す。SoC システムを設計するにはまず特定のアプリケーションをソフトウ

†1 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology

ⓒ2017 Information Processing Society of Japan

エア処理部(SW 部)とハードウエア処理部(HW 部)に分割し、切り出した HW 部を動作合成にかける。切り出した HW 部をハードウエア化するには設計者が要求する RTL 結果が得られるまで高位合成入力コードの書き換えや合成制約を変えながら高位合成を繰り返し行う必要がある。

切り出してきた HW 部のコードはハードウエアを実現するにはリソース制限を意識したものではない記述が大量にあり、ハードウエアにするためにリソース制限されたコードや高位合成ツールが対応できる制限された記述に書き換える必要がある。例えば、書き換えが必要なコードとして再起呼び出しやループの条件のパラメータ、フィードバック・パス・ループ、ポインタの制限、ダイナミック・メモリ・アクセスなどがあり、これを現存の高位合成ツールは扱うことができない。

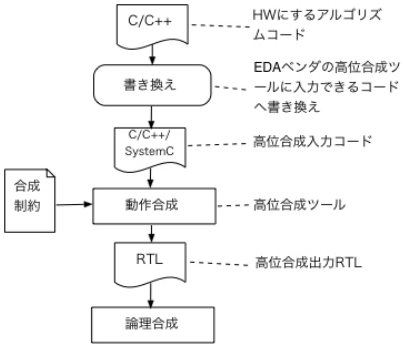

図 1 従来の高位設計フロー

2.2 課題

設計者の作業を RTL 作成からより抽象度の高いコードでの開発に移行させたことにより，従来フローは設計者を大規模な RTL 設計や FF などステートマシーンの設計作業から解放した。事実，動作合成ツールへの入力コード設計により、RTL 設計と比べて大きな工数削減が可能となったが、その一方で最適な RTL を得るための高位合成入力モデルのコード作成が重要になっている。また動作合成ツールがそもそも受け付けないコードとして、ダイナミックメモリやループ最適化[12]、再帰呼出などがあり、これらは書き換えを必要とする。

本稿では高位合成への入力記述の書き換えの自動化を目指す試みとして Source-to-Source (StoS) Compiler を用いた設計アプローチを提案する。

3. アプローチ

3.1 要素技術

従来フローの書き換えフェーズを StoS に置き換える。StoS に LLVM[13]のコンパイラ基盤を採用する。LLVM の処理は図 2 のようにフロントエンド、ミドルエンド、バックエンドの３つに分かれ順に処理される。

図 2 LLVM の処理フロー

フロントエンドでは字句解析、構文解析し中間コード LLVM Intermediate Representation (IR)を出力する。ミドルエンドではフロントエンドから受け取った IR を最適化 PASS を通して最適化し、最適化された IR を出力する。

バックエンドではミドルエンドから渡された IR から特定の CPU のネイティブコードや JavaScript ソースコードなどのターゲット・コードを生成する。これらの３つのフェーズは IR を介することにより独立しており、独立的に設計できるのが特徴である。中間コード IR は仮想機械をターゲットとした中間言語であり、特定のマシーン・コードに依存しない独立した仕様となっており、人間が読めるコードやバイナリ表現などで表現されている。IR を把握すれば入力されたコードを解釈できる。

3.1.1 フロントエンド

フロントエンドでは入力コードから IR を得ることができれば十分であり、C/C++/Objective-C/C++のパーサである clang[14]を使用した。

3.1.2 ミドルエンド

フロントエンドで生成された IR をミドルレンジの最適化 PASS を用いて IR を操作する。LLVM には各種 PASS が用意されており、高位合成入力コード作成にも利用できる。本稿では再帰呼出の除去を例に説明する。

関数の再帰呼出はソフトウエア実装では手続き呼び出しやメモリ使用量が変化するスタック操作が行われ、通常インライン展開ができない。現在の高位合成ツールではスタック・メモリを使用するコードを合成することはできない。コンパイラの研究では再帰プログラムを反復型プログラムに変換する研究[4][5][6][7][9]がなされ、末尾再帰関数に対する再帰除去が近年、多くのコンパイラで実装されている[8]。LLVM でも実装されている再帰呼出除去の PASS は末尾呼出最適化 tail recursion elimination (pass argument: -tailcallelim)であり、IR で処理するため入力言語に非依存である。本稿ではこの PASS を使用して再帰除去を行った。

3.1.3 Instruction Based Dependence Graph

LLVM には BasicBlock 単位で CFG(Control Flow Graph)を生成する PASS があるが、プログラムを最適化するとプログラムが変更される。プログラムの変更の度に依存関係の解析を行うことになる。ここでは IR ベースで最適化を行うため命令やデータの依存関係そのものをみるだけでよい。プログラムの文の間に存在するデータや制御の依存関係を見るための PDG(Program Dependence Graph)[2][3]を利用するが、LLVM では生成する PASS はないため、IR の解析をするために IR のレベル PDG(IBDG : IR-Based Dependence Graph)の生成 PASS を開発した。

本稿では依存関係を IBDG で確認するために使用したが、RTL から C へ戻して設計資産の再利用[1][10]においても利用でき、将来、高位合成入力コードの最適化解析 PASS として利用する予定である。

ⓒ2017 Information Processing Society of Japan

3.2 バックエンド

バックエンドでは高位合成入力コードを生成する。LLVM には CBackend や CppBackend があるが LLVM の API を利用したものであり、可読性がない。これらは高位合成ツールの入力コードとして入力できないコードであった。一方、IR を C に変換する Julia CBackend[11]があるが、我々は高位合成向けコードを柔軟に生成したいため、独自に IR から可読性があり且つ既存の高位合成ツールにて動作合成可能なコードを生成する C バックエンド PASS を開発した。

4. 設計フロー

従来フローの書き換えフェーズを StoS で置き換えたフローを図 3 に示す。本アプローチは、StoS の最適化 PASS を充実することにより、完全にハードウエア設計のスキルなどを補うことが可能になると考えている。この設計フローにより、課題であるコード開発の工数を削減でき、設計者はアルゴリズム開発に注力できるようになる。

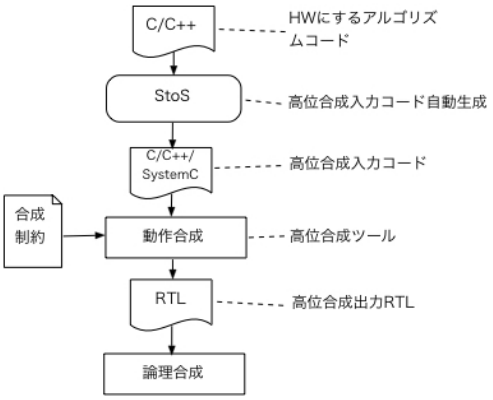

図 3　StoS を用いた高位設計フロー

4.1 実装

StoS は図 5 に示すスクリプトである。実装について以下に述べる。

4.1.1 実装環境

実装環境を以下の表 1 に示す。MacOS Sierra 上で動作する仮想 OS の上で開発をしている。

表 1　実装環境

計算機	OS　: MacOS, Sierra, 10.12.5 CPU : Intel Core m5 1.2GHz Mem : 8GB 1867MHz LPDDR3
仮想環境	VMWare Fusion8.5.8 CPU : 2 Core 指定 Mem : 2GB 指定
OS	Ubuntu 14.04LTS 64bit
コンパイラ基盤	llvm 3.4
フロントエンド	clang 3.4
ランタイムライブラリ	compiler-rt

4.2 フロントエンド

表 1 の(1)のように clang により単純に入力コードから IR を生成する。

4.3 ミドルエンド PASS の実装

始めに IBDG にてフロントエンドから渡される IR の解析を行い、各種最適化 PASS の適用により得られた IR を再び IBDG PASS にて、解析を行う。

4.3.1 IBDG の実装

本稿では、図 5 の(2)、(4)で変更前後の IBDG PASS を実行して IBDG を生成し、(3)の再帰呼除去 PASS の入出力 IR にて依存関係の変更（ここでは再帰呼出の変更）について確認する。なお、IBDG 処理として表 2 の-mem2reg, -instnamer globaldce の PASS を適用し、各処理を実行している。IBDG の PASS 引数は-instbasedg としており、出力として Graphviz[15]の dot ファイルを生成する。

表 2 IBDG 処理時の PASS

PASS の引数	実行内容
mem2reg	メモリからレジスタへ変更
instnamer	名前のない命令にナンバリングした命令を与える。
globaldce	アンリーチャブル・グローバル・コード（変数）除去
instbasedg	IR から IBDG の dot ファイルを生成

IDBG PASS では IR から IBDG 生成に必要なデータを選び、抽出する。各ノード表記は[10]で定義したノードを IR ベースのノードにした。図 4 にノード及びエッジの種類を定義する。

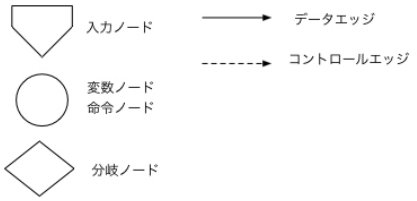

図 4　IBDG のノードとエッジ

各ノードの説明は表 3 及び表 4 に示す。ノードには i32, i64, [5 x [5 x i32]などビット幅や符号などの情報を持たせることができる。

表 3 IBDG のノード説明

ノードの種類	IR　の例	説明
入力ノード	関数の引数%x, %y,	関数の引数
データノード	%1, %2, %i, %cmp1, %array	パラメータや定数など
命令ノード	add, mul, ret, など	各種演算、return など
分岐ノード	icmp, br, switch	比較命令、Switch 命令

表 4　IBDG のエッジ説明

エッジの種類	説明
データエッジ	ビット幅、符号など
コントロールエッジ	True/False, sw.bb/sw.bb1/sw.default

IBDG 生成処理では、BitCode（IR）の情報からノードやエッジに必要な情報をリンクリストや連想メモリなどに格納し、最後に dot ファイルを生成する。

4.3.2 再帰呼出除去 PASS の実装

本フレームワークで末尾再帰除去コードを生成するた

ⓒ2017 Information Processing Society of Japan

めの PASS は tailcallelim であるが、本フレームワークでは最低限 C に変換するために tailcallelim の他に下表に示す PASS も使用する。また、-O1 オプションを使用しても同じ結果が得られる。-O1 の実行 PASS については -debug-pass=Arguments を引数に追加することで確認できる。

表 5　帰呼除去の設定 PASS

PASS の引数	実行内容
functionattrs	関数がローカルメモリをアクセスしない、グローバル変数へ返したりしない。関数からポインタ参照のみで書き換えたりしない関数を見つける。 このパスはコール・グラフのボトム・アップ・トラバースとして実装
early-cse	ドミネータ・ツリーを深さ優先で行い、冗長な命令を削除し、命令を単純化する
instcombine	単純な冗長命令を結合する
tailcallelim	末尾再帰をループに変更。
loop-rotate	単純ループローテーションへ変更
jump-threading	先行処理によりその先の分岐のスレッドを見つける

このように PASS はスクリプトで選択実行が可能であり、高位合成入力コードとして必要な最適化 PASS を開発すれば自由に追加適用できる。

4.4 バックエンド PASS の実装

各種最適化 PASS の適用により得られた IR から C へ変換する。

4.4.1 C バックエンド PASS の実装

CBackend のコマンドは図 5 の(5)のように実行する。IR 入力後、IR を module から Instruction まで 2 回辿り、2 回目スキャンの時に C 言語への変換処理を行う。

Step1：IR をスキャンして C 変換に必要な情報を抽出
Step2：BasicBlock 単位で C へ変換

ここでは、再帰除去後の IR から C への変換処理について述べる。IR には以下の特徴がある。

A)　終端命令は br, switch, ret であり、その分岐先により BasicBlock の情報より前後関係を抽出できる。

B)　再帰除去後は PHI 命令があり、この処理が除去前のスタック・メモリの役割を担っている。C へ変換するために PHI 命令を削除する。PHI 命令の引数を決定する命令列へ引数をフォワーディングする必要がある。PHI 命令の決定は決定元である BasicBlock で値が決定されるため、そのパラメータをその BasicBlock 内へ戻す。

ここでは紙面の関係で B)について例を用いて説明する。PHI 命令の構文は

<result> = phi <ty> [<val0>, <label0>], ...

である。PHI 命令の引数はペア[<val0>, <label0>]のリストを複数持つ。以下のルールにより、Label により PHI 命令が実行される BasicBlock と比較して処理を決定する。

<u>基本ルール</u>

PHI 命令が実行される BasicBlock(label)と引数の label が、同じである場合、

　パラメータ(val#)を<result>へ代入

異なる場合、

　パラメータ(val#)を label の BasicBlock へ移動

とする。IR では変数が生成されているので、Step1 にて情報収集した連想メモリから変数名に置換する。

```
#!/bin/csh

#FRONTEND
clang    -c -emit-llvm SRC/fact.c -o fact.bc       ・・・(1)

#IBDG
opt -tailcallopt -load <PATH>/Release+Asserts/lib/InstBaseDG.so
-mem2reg -instnamer -globaldce -instbasedg -x -p -flist <    fact.bc
・・・(2)

#MIDDLEEND
opt -functionattrs -early-cse    -instcombine -tailcallelim -loop-rotate
-jump-threading -o fact2.bc    fact.bc    ・・・(3)

#IBDG
opt -load
/home/tatsuoka/tools2/llvm-3.4/build/Release+Asserts/lib/InstBaseDG.
so -mem2reg -instnamer
-globaldce -instbasedg -x -p -flist <    fact2.bc    ・・・(4)

#CBACKEND
llc    -march=c -cgen=functions fact2.bc        ・・・(5)
```

図 5　全体スクリプト

5. 実験

本稿では再帰関数のサンプルとして階乗の関数を入力コードとした。図 6 は階乗関数のコードであり末尾再帰関数となる。このコードから Clang（図 5(1)）で生成した IR を図 7 に示す。

```
int fact(int n) {
   if (n < 1) {
      return 1;
   } else {
      return n * fact(n-1);
   }
}
```

図 6 再帰を利用した階乗関数

“if.else” の BasicBlock 内にて fact 関数が呼ばれており、再帰呼出であることがわかる。　図 5 の(2)で生成した IBDG を図 8 に示す。本 IBDG では、再帰について対応していないが依存関係がループで示される箇所が発生する。各リソース及び命令の依存系が把握できる。

ⓒ2017 Information Processing Society of Japan

```
; Function Attrs: nounwind uwtable
define i32 @fact(i32 %n) #0 {
entry:
  %retval = alloca i32, align 4
  %n.addr = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32* %n.addr, align 4
  %cmp = icmp slt i32 %0, 1
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  store i32 1, i32* %retval
  br label %return

if.else:                                          ; preds = %entry
  %1 = load i32* %n.addr, align 4
  %2 = load i32* %n.addr, align 4
  %sub = sub nsw i32 %2, 1
  %call = call i32 @fact(i32 %sub)
  %mul = mul nsw i32 %1, %call
  store i32 %mul, i32* %retval
  br label %return

return:                                           ; preds = %if.else, %if.then
  %3 = load i32* %retval
  ret i32 %3
}
```

図 7　階乗関数の IR

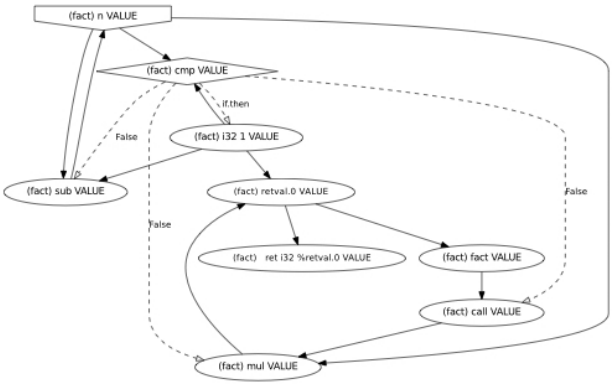

図 8 再帰除去前の IBDG

5.1 再帰呼び出し変換

図 5 の(3)の実行結果である IR を図 9 に示す。再帰呼出がなく、除去されていることがわかる。

```
; Function Attrs: nounwind uwtable
define i32 @fact(i32 %n) #0 {
entry:
  %cmp1 = icmp slt i32 %n, 1
  br i1 %cmp1, label %return, label %if.else
if.else:                                          ; preds = %if.else, %entry
  %n.tr3 = phi i32 [ %sub, %if.else ], [ %n, %entry ]
  %accumulator.tr2 = phi i32 [ %mul, %if.else ], [ 1, %entry ]
  %sub = add nsw i32 %n.tr3, -1
  %mul = mul nsw i32 %accumulator.tr2, %n.tr3
  %cmp = icmp slt i32 %sub, 1
  br i1 %cmp, label %return, label %if.else

return:                                           ; preds = %if.else, %entry
  %accumulator.tr.lcssa = phi i32 [ 1, %entry ], [ %mul, %if.else ]
  ret i32 %accumulator.tr.lcssa
}
```

図 9　再帰除去 PASS 後の IR

再帰呼出除去後の IBDG （図 5 の(4)）を図 10 示す。自分自身を呼ぶ依存関係はなくなっている。

図 10　再帰呼出除去後の IBDG

再帰除去された IR から C バックエンド（図 5 の(5)）により生成した C プログラムを図 11 に示す。ここでは使用する高位合成ツールの入力は SystemC であるため、StoS で得られたソースコードを用いて SystemC モデルのプロセス内の関数とした。その高位合成入力コードを図 13 に示す。

```
#include "fact.h"

const int const_int32_0 = 1;
const int const_int32_1 = -1;

int fact(int int32_n /* n */ ){
  int int32_n_tr3;
  int int32_accumulator_tr2;
  int int32_sub;
  int int32_mul;
  int int32_accumulator_tr_lcssa;

  int32_accumulator_tr2 = const_int32_0;
  int32_accumulator_tr_lcssa = const_int32_0;
  int32_n_tr3 = int32_n;

   if(int32_n < 1){

   }else{

    for(;;){
    int32_sub = int32_n_tr3 +    -1;
    int32_mul = int32_accumulator_tr2 *    int32_n_tr3;

    if(int32_sub < 1)    break;

      int32_accumulator_tr2 = int32_mul;
      int32_accumulator_tr_lcssa = int32_mul;
      int32_n_tr3 = int32_sub;

    }    // Recursion Loop end
   }

   return int32_accumulator_tr_lcssa;

  }
```

図 11　再帰除去後の C プログラム

ⓒ2017 Information Processing Society of Japan

マニュアルにて再帰除去したコード例を図 12 に示す。StoS 生成のコードとマニュアルコードは同じ処理を実行するコードであるが、ライン数は異なったものとなった。

```
int FACT::fact(int n){
    int ret;
    if (n == 0 || n == 1){
            ret = 1;
    }else if (n > 1) {
            ret = 1;
            for ( ; n > 0; n--)
                    ret = ret * n;
    }
    return ret;
}
```

図 12　人手によるコード例

StoS が生成したコードと人手で作成したコードの高位合成結果を比較するため、両方のコードから高位合成ツール CtoS を用いて RTL を生成し、論理合成した。

```
#include "systemc.h"

SC_MODULE(FACT)
{

public:
      sc_in <bool>              CLK;
      sc_in <bool>              XRST;

      sc_out<unsigned int>   ni;
      sc_out<unsigned int>   no;

      int fact(int n);
      void proc(){
          int n = 0;
          int ret = 0;
          wait();
          while(1){
                n = ni.read();
                ret = fact(n);
                no.write(ret);
                wait();
          }
       };

          SC_CTOR(FACT) :
          CLK("CLK"),
          XRST("XRST"),
          ni("ni"),
          no("no")
          {
                SC_CTHREAD(proc,CLK.pos());
                async_reset_signal_is(XRST, false);
          }
};
```

図 13　高位合成入力コード例

高位合成条件としてテクノロジライブラリは 65nm、動作周波数 200MHz、合成制約は(A)ブレイク・パイプライン指定とした。なお、RTL については誌面の関係でここでは省略する。

StoS および人手によるコードから生成した RTL のレイテンシはともに表のように同一の結果が得られた。

表 6 生成した RTL のレイテンシ

入力 n	1	2	3	4
レイテンシ[Cycle]	1	3	5	6

論理合成 RTL Compiler の結果、StoS の入力コードから生成した RTL の AREA はマニュアル C コードにくらべて 0.6%少なく、Sequential が 1 instance、inverter が 2 instance、logic が 10 instance 少ない結果が得られた。

6. まとめ

本稿では高位設計における高位合成コードの開発工数を削減するためにLLVMフレームワークを用いた高位合成ソースコード生成ツール StoS を提案し、再帰関数除去のコード生成の例を説明した。高位合成入力コード開発ツールを用いて実行することにより工数が削減できる可能性を示した。

今後はハードウエア向け各種最適化 PASS の研究開発と一部マニュアル作業の完全自動化を行う予定である。

文　献

[1] 立岡真人,金子峰雄,”既存 RTL 資産の高位からの再合成アプローチ,”信学技報 VLD2012-145, pp.55-60,2013

[2] M. Weiser, “Program slicing,” IEEE Transactions on Software Engineering, Vol.SE-10, No.4, July 1984, pp.352-357

[3] 下村隆夫, “プログラムスライシング技術と応用,” 共立出版,1995

[4] J. Ferrante, K.J. Ottenstein, and J.D. Warren. “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, 9(3), pp.319–349, 1987.

[5] Y. A Liu and S. D. Stoller. “From recursion to iteration: what are the optimization?.,” Proceedings of the ACM SIGPLAN 2000 Workshop on Partial Evaluation and Semantics-Based Program Manuipulation, Janualy 2000, pp.73-82.

[6] 二村良彦, 大谷啓記. “線形再帰プログラムからの再帰除去とその実際的効果,” コンピュータソフトウエア, Vol. 15, No.3, ３月, 1998, pp.38-39

[7] 市川裕輔, 小西善二郎, 二村良彦, “単一後継関数を持つ相互再帰プログラムに対する再帰除去の実装,” 日本ソフトウエア科学会第 21 回大会論文集, 7C-1, 9 月

[8] D. A. Bacon, S. L.Graham and O.J.Sharp,”Compiler Transformations for High-Performance Computing,” ACM Computing Surveys, Vol. 26, No.4, December 1994, pp.345-420

[9] Yusuke Ichikawa, Zenjiro Konishi, Yoshihiko Futamura. “Recursion Removal from Recursive Programs with Only One Descent Function,” IEICE TRANS.INF.&S_YST., Vol.E88-D,No.2 February 2005

[10] 立岡真人,青木利晃, 金子峰雄,”再利用のための RTL からの関数コード生成手法,”信学技報 VLD2013-165, pp.55-60,2013

[11] Transpiling Julia to C – The LLVM-CBackend, https://juliacomputing.com/blog/2016/03/10/j2c-announcement.html

[12] Yuta Kato, Kenshu Seto. “Loop Fusion with Outer Loop Shifting for High-level Synthesis,” IPSJ Transactions on System LSI Design Methodology, Feb. 2013, Vol.6, pp.71-75

[13] The LLVM Compiler Infrastructure Project, http://llvm.org

[14] Clang : C language Family Frontend for LLVM, https://clang.llvm.org

[15] Graphviz, http://www.graphviz.org

[16] LLVM’s Analysis and Transform Passes: https://llvm.org/docs/Passes.html

ⓒ2017 Information Processing Society of Japan