# マルチFPGA向けシミュレーションコードの再利用性を改善するコード生成系の実装

福田 航生[1,a)]　池原 陽大[1,b)]　本吉 圭吾[1,c)]　眞邉 泰斗[1,d)]　柴田 裕一郎[1,e)]　上野 知洋[2,f)]
佐野 健太郎[2,g)]

**概要**：複数の FPGA から構成されるシステムの動作シミュレーションは，複数のモジュールを通信可能な状態で並列動作させる必要がある．しかし，既存ツールチェーンによるサポートは不十分で，そのようなシステムのシミュレーションには妥協が必要なのが現状である．
本研究の手法では，複数のモジュールの処理を単一のモジュール内で並列のタスクとして実行するように書き換える．これにより既存のツールチェーンを使用した動作シミュレーションが可能となる．
ソースコードをパースし，本手法を用いて単一のモジュールに統合したソースコードを生成する処理系の実装を行った．変換処理の時間はコンパイルを含む全体の処理時間の 0.3% 未満となった．

**キーワード**：リコンフィギュアラブルコンピューティング，並列アーキテクチャ，FPGA

## 1. はじめに

近年のプロセッサの高集積化の鈍化や電力等の問題 [1], [2] により，HPC で現在主流であるメニーコアプロセッサを多数接続するアーキテクチャでの性能向上が困難になっている．また，メモリのバンド幅が性能のボトルネックになる場合が多いという課題もある．一方，FPGA を使用した計算では専用のハードウェアを構成することにより，チップ上の資源を効率よく利用し，低消費電力で低遅延な処理を実現できる．さらに，ストリーム処理との相性がよいことや，内蔵された SRAM は小容量であるものの高いバンド幅を提供することから，外部メモリのバンド幅が並列処理のボトルネックになりにくい．こうした理由から，HPC の分野で FPGA が注目されている．

このような背景の下，理化学研究所計算科学研究センターでは，HPC の新たな計算アーキテクチャとして FPGA クラスタシステム ESSPER を開発している [3]．ESSPER は並列に動作する多数の FPGA をネットワークを介して接続した構成をもち，それらが持つ再構成可能な領域を利用して所望のシステムを実現できる．ESSPER で動作するシステムはさまざまな手法での実装ができ，C++での動作の記述からハードウェア合成を行う高位合成の使用も可能である．本研究ではこの方法について議論する．

高位合成の強みの一つは，開発用のホストでソフトウェア開発で用いられるような手法でシミュレーションすることが可能な点である．ハードウェア用に記述されたソースコードは，ソフトウェア向けの実行ファイルとしてもコンパイル可能であり，テストベンチからハードウェア向けに記述したルーチンを呼び出し，シミュレーションを実行できる．ESSPER において，C++記述からのハードウェア合成には Intel HLS Compiler[4] が用いられ，この高位合成系もそのような仕組みを提供している．一般的な，単体の FPGA を用いるシステムの動作シミュレーションは十分にサポートされており，この手法で問題なく動作のテストが行える．

しかし，複数の FPGA で構成されたシステムのシミュレーションは，Intel HLS Compiler では十分にサポートされておらず，高位合成の恩恵を完全には受けられないのが現状である．複数の FPGA からなるシステム全体のシミュレーションを C++記述ベースで行うことはできず，関数などの細かい単位でテストを行うなどの妥協が必要である．

上記の問題を克服するため，本研究では複数の FPGA が

[1] 長崎大学
〒 852–8521 長崎県長崎市文教町 1–14
[2] 理化学研究所計算科学研究センター
〒 650–0047 兵庫県神戸市中央区港島南町 7–1–26
[a)] fukuda@pca.cis.nagasaki-u.ac.jp
[b)] ikehara@pca.cis.nagasaki-u.ac.jp
[c)] kmoto@pca.cis.nagasaki-u.ac.jp
[d)] tmanabe@nagasaki-u.ac.jp
[e)] yuichiro@nagasaki-u.ac.jp
[f)] tomohiro.ueno@riken.jp
[g)] kentaro.sano@riken.jp

協調して動作するようなシステムの動作シミュレーション手法について検討した．また，その手法を実現するフレームワークの実装を行った．

## 2. 関連研究

吉内らは，大規模システムの分割実装の際に利用できる分散 RTL シミュレーションを提案した [5]．大規模システムを複数の FPGA に分割実装するような構成のシミュレーションは単一のマシン上で行われるが，これにはパフォーマンスの問題を伴う．吉内らの研究では複数の FPGA を用いるシステムのシミュレーションを分散環境で行うためのシミュレーションモジュールとデバイスドライバを実装し，従来手法でのシミュレーションより 2.33 倍から 5.70 倍の高速化を達成している．

新井らは，C++で記述した複数のコンポーネントの統合シミュレーションのための通信フレームワークを提案した [6]．各コンポーネントは C++で記述された単体テストで行えるものの，複数のコンポーネントを組み合わせた場合の統合テストは RTL で行う必要があった．新井らの手法では C++で実装された publish/subscribe 通信フレームワークを利用し，C++記述のまま機能シミュレーションを行える．

弘中らは，M-KUBOS というマルチ FPGA クラスタの HLS 向けのメッセージパッシングインタフェース (MPI) を提案した [7]．これは，FPGA 間通信を行うアプリケーションを標準的な MPI アプリケーションとして実装できるものである．シミュレーションも可能であり，ローカルマシン上ではソケットを使用した通信を行うマルチプロセスアプリケーションとして FPGA 間通信が再現される．

## 3. シミュレーションフレームワーク

### 3.1 提案手法

Intel HLS Compiler を使用する場合，単一のテストベンチから複数のコンポーネントを並列実行することはできないものの，単一のコンポーネントから複数のタスクを並列実行することは可能である．そのため，コンポーネントに含まれる全てのタスクの実行を 1 つのコンポーネントにまとめることにより複数コンポーネントで分散して行われていた処理を単一コンポーネント内で完結させることが可能である．また，コンポーネント間で通信を行う場合にも，タスク間の通信に変換することで同等の処理が可能である．

この際にソースコードに必要となる変換は次の 4 つである．

(1) `component` 属性が適用された関数の内容を結合する
(2) `component` 属性が適用された関数の引数を結合する
(3) グローバルに宣言された関数や変数を統合する
(4) モジュール間の通信をタスク間の通信に変換する

コード 3 はこれらの変換を適用することによってコード 1 とコード 2 を統合した例である．

各変換で考慮すべき点と，出力されるソースコードとの対応について述べる．特に明記しない限り行数はコード 3 内の行番号を指す．

1 について，コンポーネントの関数では `ihc::launch` および `ihc::collect` のみを呼び出すという規約にする．この状況では変数名の衝突は起こり得ないため，そのまま内容を結合すればよい．ただし，`ihc::collect` はタスクの終了までブロックするため，`ihc::collect` の呼び出しは全ての `ihc::launch` の呼び出しの後に配置しなければならない．この変換の結果，17–25 行目の部分が出力される．

2 については，もともと別の関数であり同名の引数を持つ可能性があるため名前が衝突しうるという点がある．そこで，全ての引数にモジュール名でプレフィックスをつけることにする．それに伴って，関数内で当該引数を使用している箇所は参照先の修正が必要である．この変換の結果，19–20 行目が書き換えられる．

3 についても，分割してコンパイルすることを前提としていたソースコードであるため，2 と同様に単純に結合すると名前の衝突が問題となる．そのため，衝突する変数名を変更するなどの対策が必要である．C++には namespace 機能により，単一の実行環境に複数の名前空間を定義することが可能である．そのため，引数と異なりモジュールごとに異なる namespace に定義を移動することにより名前の衝突を回避する．この変換の結果，5–15 行目が出力される．

4 については，`ihc::stream` 型の接続用の FIFO バッファを新たに宣言して接続する．ストリーム出力インタフェースである `ihc::stream_out` への write とストリーム入力インタフェースである `ihc::stream_in` からの read を，接続用の FIFO バッファへの read/write に変換することで実現可能である．ただし，接続箇所を機械的に判定することは不可能なため，コード 4 のような設定ファイルから与えることを想定している．この変換の結果，3 行目に新たに変数が宣言され，7 行目と 13 行目の参照先が書き換えられている．

別の考慮すべき点として，ヘッダファイルを名前空間内でインクルードすべきでないという点がある．include ディレクティブは関数や変数の宣言が記述されたソースコードを展開しているにすぎないため，本来の名前空間と異なる名前空間でライブラリ関数を宣言したことになってしまい，リンク時にエラーとなるためである．そこで，include ディレクティブはモジュールごとの namespace の外で include することにする．この場合モジュール内で定義した変数を外から参照できるようヘッダファイルに宣言を記述していた場合には正常に動作しないことになる．この点については現時点ではヘッダファイルの使用は Intel HLS のライブラリとモジュール共通の宣言が主であると判断し，特に対

策は行わないこととした．

コード 1: module_a.cpp

```
#include "HLS/hls.h"

ihc::stream_out<float> st_out;

void read_sender(ihc::mm_master<float> *mm_in)
     {
    st_out.write((*mm_in)[0]);
}

component hls_avalon_slave_component
void module_a(
    hls_avalon_slave_register_argument ihc::
        mm_master<float> &mm_in) {
    ihc::launch<read_sender>(&mm_in);
    ihc::collect<read_sender>();
}
```

コード 2: module_b.cpp

```
#include "HLS/hls.h"

ihc::stream_in<float> st_in;

void receive_writer(ihc::mm_master<float> *
    mm_out) {
    (*mm_out)[0] = st_in.read();
}

component hls_avalon_slave_component
void module_b(
    hls_avalon_slave_register_argument ihc::
        mm_master<float> &mm_out) {
    ihc::launch<receive_writer>(&mm_out);
    ihc::collect<receive_writer>();
}
```

コード 3: 結合済みのモジュール

```
#include "HLS/hls.h"

ihc::stream<float> conn_port_0;

namespace module_a {
    void read_sender(ihc::mm_master<float> *
        mm_in) {
        ::conn_port_0.write((*mm_in)[0]);
    }
}

namespace module_b {
    void receive_writer(ihc::mm_master<float> *
        mm_out) {
        (*mm_out)[0] = ::conn_port_0.read();
    }
}

component hls_avalon_slave_component
void module_a_module_b(
    hls_avalon_slave_register_argument ihc::
        mm_master<float> &module_a_mm_in,
    hls_avalon_slave_register_argument ihc::
        mm_master<float> &module_b_mm_out) {
    ihc::launch<::module_a::read_sender>(&
        module_a_mm_in);
    ihc::launch<::module_b::receive_writer>(&
        module_b_mm_out);
    ihc::collect<::module_a::read_sender>();
    ihc::collect<::module_b::receive_writer>()
        ;
}
```

コード 4: 設定ファイル

```
modules:
  - "hw_mod/module_a.cpp"
  - "hw_mod/module_b.cpp"
connections:
  - from: "module_a.st_out"
    to: "module_b.st_in"
```

### 3.2 変換の自動化

ソースコードの変換を自動で行うコード変換系を C++ で実装し，orochi というコマンドラインユーティリティとして使用できるようにした．このプログラムにはプロジェクト立ち上げ時用に雛形コードを出力する機能なども実装しているが，ここではコード変換処理についてのみ述べる．

コード変換の主な処理の流れは以下のようになる．

まず，設定ファイルから与えられたモジュールのソースファイルを全て読み込み，抽象構文木の構築を行う．C++ のパーサは GCC[8] や Clang[9] といった C++コンパイラが実装しているものの，コードの変換後に元のテキストに近い形態で出力を行いたいため，今回の要件には合わないと判断したため，独自のパーサを実装した．C++の仕様は膨大であることから，高位合成での使用頻度が低い構文は現時点では未実装である．

次に，構築した抽象構文木を走査し，参照しているヘッダファイルのうち，システムにインストールされているもの以外を検索し，出力ディレクトリにコピーする．この操作が必要な理由は，出力ファイルにインクルードディレクティブがそのままコピーされるため，元のハードウェアモジュール用のディレクトリに存在していたヘッダファイルにアクセスできなくなるためである．

その後，構築した抽象構文木を走査し，与えられたモジュールを結合した新たなモジュールのソースコードを生

成する．

最後に，出力ディレクトリに生成されたソースコードを書き出す．

なお，トークンや抽象構文木のオブジェクトは明示的に指示されない限りコピーが行われないように実装している．これによりオブジェクトのコピーによる性能劣化を防止したりメモリを節約したりする効果が見込まれる．

#### 3.2.1 字句解析

Flex を用いて生成した字句解析器を使用している．

入力ファイルをトークンに分割する．後述する構文解析器はここで生成されたトークンを Token クラスのインスタンスとして受け取るようにしている．

コード生成後に入力と近い形態で出力したいため，トークンの文字列と空白文字やコメントを保持する必要がある．ただ，空白文字を単一のトークンとして処理してしまうと，後の構文解析で使用する文法規則に空白文字トークンを含めなければならなくなり，冗長になる．そこで，トークンとトークンの間にある空白文字もトークンにデータとして保持するようにした．

具体的な字句解析の流れは以下のようになる．字句解析器が空白文字列を読み取ったとき，その文字列を一時的な変数に保存しておく．コメントを読み取った際にも空白文字と同様の扱いをする．字句解析器が通常のトークンを読み取ったとき，トークンの文字列と共にその時点で保持されている空白の文字列をトークンオブジェクトに保存する．ファイルの終端のスペースの保持はトークンではなくファイルが保持するものとして，構文解析のステップで処理している．

#### 3.2.2 構文解析

Bison を用いて，C++記述された構文解析器を生成し使用している．このステップでは構文解析を行い，抽象構文木を構築する．

ソースファイル内の式や文といった要素は Partial として抽象化し，具体的な構文上の要素はここから派生したサブクラスで保持することにした．Partial クラス自体は要素のタイプ，ファイルへの書き込み，再帰的な走査などの共通する操作を行えるインタフェースを定義している．

#### 3.2.3 コード生成

構文解析によって構築された抽象構文木を走査し，モジュールを結合した新たなソースコードを生成する．

変換の際，モジュール間の接続の情報が必要であるが，その情報はコード 4 のような YAML 形式 [10] の設定ファイルから取得する．YAML ファイルの解析には yaml-cpp[11] を利用した．

ここでも Token や Partial のオブジェクトのコピーを避けて実装している．オブジェクトを元のオブジェクト集合から取得し，元のオブジェクト集合から削除する操作をここでは抽出と呼ぶことにする．

変換の処理の流れは以下のような流れで行う．

(1) インクルードディレクティブの抽出
(2) 接続に使用する変数の抽出
(3) 変数の参照の書き換え
(4) component の生成
(5) モジュール間通信用の変数の生成
(6) モジュール別 namespace の生成
(7) 生成したコード片のマージ

「インクルードディレクティブの抽出」ステップでは，全ソースファイルからのインクルードディレクティブの抽出を行う．これは通常インクルードディレクティブは namespace 内で行うべきでないため，後のステップで namespace 内にインクルードディレクティブが配置されてしまわないようにするためである．

「接続に使用する変数の抽出」ステップでは，設定ファイルで定義された，モジュール間の接続に関係する変数の抽出を行う．これにより，不必要となる変数宣言を除去するとともに通信に使用するデータの型情報を取得している．

「変数の参照の書き換え」では，接続に使用する変数の抽出のステップで抽出された変数への参照を，後に生成するモジュール間通信用の変数への参照に書き換える．

「component のマージ」ステップでは，全モジュールから `component` 属性が適用された関数を抽出し，1 つのコンポーネントにマージする．全ての `ihc::launch` の呼び出しは `ihc::collect` の前の配置しなければならないため，呼び出しの並べ替えも行う．

「モジュール間通信用の変数の生成」ステップでは，元の通信用変数の型情報を利用してモジュール間通信用の変数を新たに生成する．この際，元の接続用の変数は `ihc::stream_in<type>`か `ihc::stream_out<type>` の型であるが，これらは read と write のいずれか一方のみが可能である．これを read/write 操作が可能な `ihc::stream<type>`に書き換えることにより，接続元タスクで書き込んだデータを接続先タスクで読み出せるようになる．

「モジュール別 namespace の生成」ステップでは，変数名や関数名の衝突を避けるため，モジュールごとに別の namespace に宣言を移動する．orochi ではさらに，モジュールごとの namespace を，新たに生成した namespace の内部に配置している．これはグローバルな空間には他のライブラリの宣言が存在する可能性があり，それらとの名前の衝突を避けるためである．

「生成したコード片のマージ」ステップでは，これまでのステップで生成したコードをコンパイル可能な形に統合し，ファイルに書き出している．

### 3.3 既存のツールチェーンとの親和性

実際のプロジェクトでは make のような差分ビルド可能

なツールを使用している．差分ビルドではファイルのタイムスタンプを確認し，ソースファイルが生成物よりも古い場合はコンパイルをスキップするという実装がなされている．シミュレーション用実行ファイルのコンパイルには時間がかかるため，本システムが差分ビルドを妨げると，ビルドに必要な時間が長くなり，ソースコードの変換の自動化による恩恵が少なくなってしまう．そこで，ソースコードの変換にも差分ビルドの機能を実装した．orochi の差分ビルドは 2 つの処理からなる．それが，依存するヘッダファイルのうち必要なもののみをコピーし直す処理と，入力ファイルが更新されていた場合のみ変換を行う処理である．2 番目の処理である依存するヘッダファイルのコピーには，使用しているヘッダファイルの検出が必要なため，ソースファイルのパース自体を避けることはできない．

変換処理全体の流れを説明する．まず，入力であるモジュールのソースファイルを全て読み，抽象構文木を構築する．次に，抽象構文木の中からインクルードディレクティブを探し，以下の全てを満たすもののみを出力ディレクトリにコピーする．

(1) ソースファイルの存在するディレクトリから探索して，ヘッダファイルが見つかる
(2) 見つかったファイルは出力先のファイルより新しい

最後に，入力ファイルに出力ファイルよりも新しいものが含まれていた場合のみ，ソースコードの変換処理を行う．

## 4. 評価

コード 5 とコード 6 を例に，コードの自動変換により設計者が手動で記述すべきコードの量がどの程度削減できるのかを評価する．さらに，大規模な実装で本研究で実装したコード変換系を使用することを想定し，コードの変換のパフォーマンスについても評価する．なお，これらのコードは，1 つの float 型のデータをコード 5 の `func0`, `func1`，コード 6 の `func0`, `func1` をこの順で経由し，`mod0` の `mm_in` から `mod1` の `mm_out` に転送するものである．

### 4.1 実装するコード量

コード 5 とコード 6 からコード 7 のようなコードが生成される．統合後のコードを用いたシミュレーションにおいて入力した値と同じ値を読み出せることが確認できた．この 46 行のコードを生成するためには，設計者は 6 行程度の設定ファイルを記述すればよいため，ハードウェアのモジュールのソースコードを設計者自らコピーして同等の変換を手動で行う場合と比較して，設計者の負担を大きく削減できる．

実際の科学計算向けのシステムを実装する場合にはさらにコードベースが拡大し，複雑な変更が必要になると考えられるため，設計者がハードウェア用のモジュールのコードを変更してシミュレーション用のモジュールを作成する場合と比較した作業効率の差はさらに大きくなると考えられる．

### 4.2 性能評価

#### 4.2.1 方法

2 つの単純なモジュール使用し，モジュールの結合を行った後にシミュレーションの実行ファイルをビルドし，全体の所要時間に占める変換処理の時間の割合を計算した．その際に，各モジュール内部のタスク数を変化させて繰り返し計測を行った．

内部のタスク数が 3 の場合のモジュールの例をコード 5, コード 6 に示す．これはコード 5 の `mm_in` から読み取った値が全ての関数を経由してコード 6 の `mm_out` に書き込まれるモジュールである．変換処理によってコード 5 の `st_out` は，コード 6 の `st_in` に接続される．

コード 5: 評価に使用するコードの例（1）

```
1  #include "HLS/hls.h"
2
3  ihc::stream_out<float> st_out;
4  ihc::stream<float> pipe0;
5
6  void func0(ihc::mm_master<float> *mm_in) {
7      pipe0.write((*mm_in)[0]);
8  }
9
10 void func1() {
11     st_out.write(pipe0.read());
12 }
13
14 component hls_avalon_slave_component
15 void mod0(
16     hls_avalon_slave_register_argument ihc::
           mm_master<float> &mm_in) {
17     ihc::launch<func0>(&mm_in);
18     ihc::launch<func1>();
19     ihc::collect<func0>();
20     ihc::collect<func1>();
21 }
```

コード 6: 評価に使用するコードの例（2）

```
1  #include "HLS/hls.h"
2
3  ihc::stream_in<float> st_in;
4  ihc::stream<float> pipe0;
5
6  void func0() {
7      pipe0.write(st_in.read());
8  }
9
10 void func1(ihc::mm_master<float> *mm_out) {
11     (*mm_out)[0] = pipe0.read();
```

```
}

component hls_avalon_slave_component
void mod1(
    hls_avalon_slave_register_argument ihc::
        mm_master<float> &mm_out) {
    ihc::launch<func0>();
    ihc::launch<func1>(&mm_out);
    ihc::collect<func0>();
    ihc::collect<func1>();
}
```

コード 7: モジュールの統合の結果生成されるコード例

```
#include "HLS/hls.h"
#include "HLS/hls.h"
#include "orochi.h"

namespace orochify {
ihc::stream<float> conn_port_0;

namespace mod0 {
ihc::stream<float> pipe0;

void func0(ihc::mm_master<float> *mm_in) {
    pipe0.write((*mm_in)[0]);
}

void func1() {
    ::orochify::conn_port_0.write(pipe0.read()
        );
}
}

namespace mod1 {
ihc::stream<float> pipe0;

void func0() {
    pipe0.write(::orochify::conn_port_0.read()
        );
}

void func1(ihc::mm_master<float> *mm_out) {
    (*mm_out)[0] = pipe0.read();
}
}

}

component hls_avalon_slave_component
void orochi(
    hls_avalon_slave_register_argument ihc::
        mm_master<float> &mod0_mm_in,
    hls_avalon_slave_register_argument ihc::
        mm_master<float> &mod1_mm_out) {
    ihc::launch<orochify::mod0::func0>(&
        mod0_mm_in);
    ihc::launch<orochify::mod0::func1>();
    ihc::launch<orochify::mod1::func0>();
    ihc::launch<orochify::mod1::func1>(&
        mod1_mm_out);
    ihc::collect<orochify::mod0::func0>();
    ihc::collect<orochify::mod0::func1>();
    ihc::collect<orochify::mod1::func0>();
    ihc::collect<orochify::mod1::func1>();
}
```

モジュール内部のタスク数を $N$ 個にしたとき，FIFO バッファおよび他モジュールとの通信用のポート，タスクを処理する関数，`ihc::launch` の呼び出し，`ihc::collect` の呼び出しが $N$ 個ずつ生成される．よって，タスク数を 1 増加させたとき，空行を除くソースコードの行数（Source Lines Of Code: SLOC）は 6 増加する．

変換およびシミュレーションのコンパイルは Intel Xeon E3-1225 プロセッサ (3.30 GHz)，メモリ 16 GB のマシンで 8 回実行し，その平均実行時間を評価した．

#### 4.2.2 評価結果と考察

タスク数を増加させてシミュレーションを行った場合にも，入力した値と同じ出力値を得られた．よって，本手法は規模の大きいモジュールのシミュレーションにも適用可能であることが確認できた．モジュールあたりのタスク数ごとの変換時間と，統合処理前後のコンパイル時間のグラフを図 1 に示す．統合処理前のコンパイル時間は，2 つのモジュールをそれぞれコンパイルし，その合計時間を参考としてプロットしたものである．統合を行わずに実際にシミュレーションが行えるというわけではない．また，統合後の全体の処理時間（変換とコンパイルにかかった時間の合計）に占める変換時間のグラフを図 2 に示す．

変換処理前後のコンパイル時間を比較すると，タスク数が 4000 を超えたあたりから統合後のモジュールのコンパイルのパフォーマンスが低下している．これは，コンパイラが巨大なモジュールのコンパイル時にパフォーマンス低下する実装になっているか，メモリの使用量が増加したことによってスワップアウトが発生している可能性があると考えている．

図 2 から，モジュールの規模が大きくなった場合にも全体の処理時間に占める変換時間の割合は 0.3% 程度に収まっている．処理時間への相対的な影響は小さいことから，設計時に使用する際の体感的なコンパイル時間への影響は軽微であると考えられる．また，全体の処理時間に占める変換時間の割合は，4000 を超えたあたりから減少している．このことから，変換処理はコンパイルで発生するようなパフォーマンスの低下の影響を受けておらず，モジュールの規模が大きくなるほど全体の処理時間に対する相対的な変換処理時間の影響は小さくなると考えられる．ただ，

モジュールの規模がさらに拡大すると，変換処理でも同様のパフォーマンスの低下が発生する可能性はある．

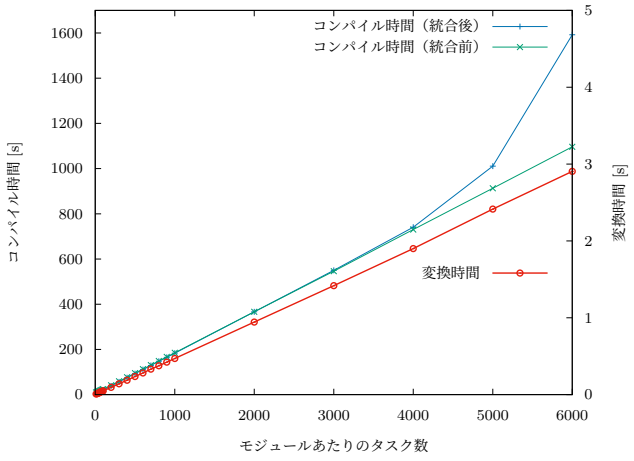


図 1: 変換とコンパイルの所要時間

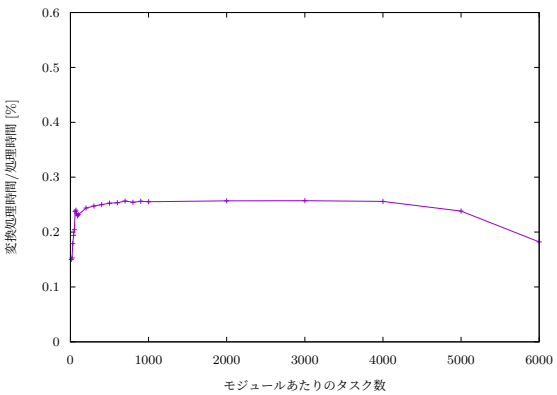


図 2: 処理時間全体に占める変換処理時間の割合

## 5. おわりに

本稿では，C++で記述された複数 FPGA 向けのハードウェアの動作シミュレーション手法の提案と，それを実現するためのコード変換系の実装と評価を行った．

これを利用することにより設計者は数行の設定ファイルの記述から，複数モジュールを統合したシミュレーション用のモジュールを生成することができ，シミュレーションにおける設計者の手間を大幅に削減できる．今回の実装では C++の文法を全て網羅したものではないため，実際のアプリケーションの実装に使用するには対応する文法の拡充が必要になると考えられる．ESSPER では FPGA 間をより複雑なトポロジで接続することができるため，この点についても別途検討が必要である．

変換の処理にかかる時間について，実行ファイルのコンパイルを含む全体の処理時間の 0.3% 未満となり，処理時間への相対的な影響は十分に小さいといえる．ただ，統合の結果生成された大きなモジュールはコンパイルにかかる時間が長くなったため，この原因の調査が必要である．また，実際のモジュールと比較して単純なモジュールを使用しているため，実際のコードベースの特性を反映していない可能性がある．この点については，今後実際のアプリケーションのソースコードを用いた評価を行う必要がある．コードの変換がシミュレーションの性能に与える影響についての検証も必要である．

## 参考文献

[1] Mack, C. A.: Fifty Years of Moore's Law, *IEEE Transactions on semiconductor manufacturing*, Vol. 24, No. 2 (2011).
[2] Bohr, M.: A 30 Year Retrospective on Dennard's MOSFET Scaling Paper, Technical report, IEEE (2007).
[3] 佐野健太郎，上野知洋，宮島敬明，Jens Huthmann，小柴 篤史：ESSPER: 高性能計算のためのスケーラブルかつ柔軟な FPGA クラスタシステムの開発，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. 120, No. 339, pp. 7–12 (2021).
[4] Intel Corporation: High-Level Synthesis Compiler - Intel® HLS Compiler, `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`. (参照日: 2023/01/11).
[5] 吉内大成，中條拓伯：スケーラブル・ハードウェア機構による分割回路のための分散 RTL シミュレーション，組込みシステムシンポジウム 2015 論文集， Vol. 2015, pp. 137–138 (2015).
[6] 新井健太， 大川猛, 大津金光, 横田隆史：高位合成可能な C++記述のための分散ミドルウェアを用いた機能シミュレーション，研究報告組込みシステム (EMB)， Vol. 2019, No. 6, pp. 1–2 (2019).
[7] 弘中和衛, 飯塚健介, 天野英晴：M-KUBOS マルチ FPGA における HLS 向けメッセージパッシングインタフェースの実装，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. 122, No. 286, pp. 61–66 (2022).
[8] Free Software Foundation: GCC, the GNU Compiler Collection - GNU Project, `https://gcc.gnu.org/`. (参照日: 2023/01/18).
[9] LLVM Project: Clang C Language Family Frontend for LLVM, `https://clang.llvm.org/`. (参照日: 2023/01/18).
[10] The YAML Project: The Official YAML Web Site, `https://yaml.org/`. (参照日: 2023/01/18).
[11] Jesse Beder: GitHub - jbeder/yaml-cpp: A YAML parser and emitter in C++, `https://github.com/jbeder/yaml-cpp`. (参照日: 2023/01/18).