Vol.2021-ARC-244 No.46
Vol.2021-SLDM-194 No.46
Vol.2021-EMB-56 No.46
2021/3/26

C 言語ベースのシステムレベル設計における低コストで高速な協調検証環境

稲石 日奈子[1]　山本 椋太[1]　伊藤 慎治[2]　本田 晋也[3]　枝廣 正人[1]

概要：システムレベル設計においては，設計と性能評価を何度も繰り返し開発を行う．そのため 1 回の性能評価が低速であると開発に膨大な時間がかかる．そこで本研究では，高速かつ高精度な HW 性能評価手法の低コストな実現可能方法を検討した．システムレベル設計環境である SystemBuilder を用いて QEMU と SystemC を利用した協調検証環境を実現した．C 言語で記述された CNN 手書き数字推論ソースコードを評価対象として，本研究開始以前より SystemBuilder で提供されている HDL を用いた協調検証環境と比較した．その結果，同等の精度，測定粒度のまま 4.6 倍の高速化を実現した．また，無償ツールのみで協調検証環境を構築することで，ツールコストを 0 に削減した．

1. はじめに

近年，組込みシステム開発において System-on-a-chip（SoC）は主流になりつつある [1]．SoC の設計生産性向上のための方法として，システム全体を同一の言語で記述することで抽象度を高め，ハードウェア（HW）とソフトウェア（SW）の協調設計を可能とする，システムレベル設計が注目されている．

我々はシステムレベル設計環境である SystemBuilder[2] を研究開発している．SystemBuilder は，C 言語で記述されたシステム記述から，HW，SW およびその間の通信インタフェースを自動生成できる．HW 設計のために，C 言語からハードウェア記述言語（HDL）で記述されたクロックレベルの HW を生成する，高位合成（HLS：High-Level Synthesis），および回路記述の最適化・配置配線を行う論理合成を，既存ツールを用いてサポートしている．

システムレベル設計では，C 言語でシステムを記述するため，単純にアルゴリズムを実装するだけでは求められる HW 性能を達成できないことがあり，回路面積や実行速度を考慮して設計と性能評価を繰り返す，設計空間探索が必要になる．1 回の HW の設計空間探索にかかる時間が長いと開発期間の増大につながり，1 回あたりの時間を短縮することは重要である．高価なツールの使用により高速化が可能であるが，高価なツールの使用は SystemBuilder が広く使用される際の障害になる．

本研究では，高速かつサイクル精度で HW 性能評価可能な手法の低コストな実現方法を検討した．まず，高位合成ツールが生成する合成レポート等からシミュレーションコードを生成し，高速かつ正確に性能を見積もる既存研究 [3] の利用を検討したが，SystemBuilder での実現が困難であった．そこで，SystemBuilder が提供している HW 性能評価フローの 1 つである SL-Sim（システムレベルシミュレーション）をベースとした新たな協調検証環境を検討し，オープンソースのエミュレータである QEMU と，システムと HW 設計を記述するための C++ クラスライブラリを利用した言語である SystemC を用いて実現した．提案手法の評価として，高位合成向けに C 言語で記述された CNN（Convolutional Neural Network）手書き数字推論ソースコード [4] を対象に，SL-Sim と比較する．

2. 準備

2.1 SystemBuilder

本節では我々が研究開発している，システムレベル設計環境である SystemBuilder[2] について紹介する．SystemBuilder での開発フローを**図 1** に示す．なお，黄色で提案手法の導入により新たに追加された要素を，紫色で本研究開始以前より存在していたが，提案手法の導入により記述の変更があった要素を記述している．C 言語で設計する場合，固有の名前を持つプロセスをそれぞれ関数として記述する．プロセス間の通信は通信プリミティブを用いて定義し，プロセスからそのアクセス関数を呼び出して通信する．SystemBuilder が提供しているプロセス間通信には次のも

[1] 名古屋大学
Nagoya University
[2] システムアイ
[3] 南山大学
Nanzan University

Vol.2021-ARC-244 No.46
Vol.2021-SLDM-194 No.46
Vol.2021-EMB-56 No.46
2021/3/26

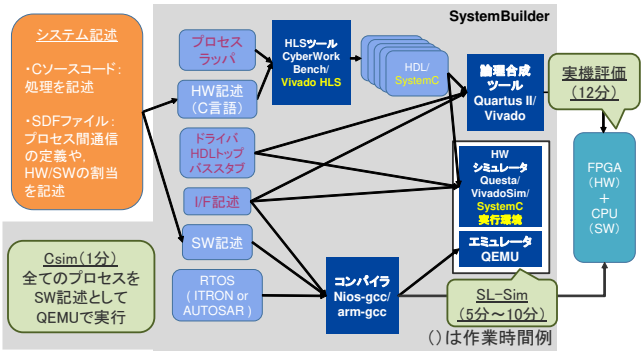

図 1 SystemBuilder の開発フロー

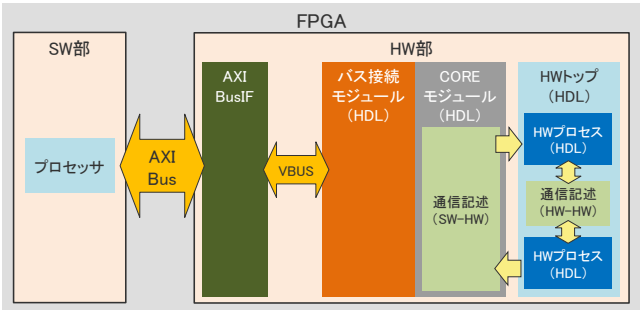

図 2 FPGA でのシステム構成

のがある．

- ノンブロック通信プリミティブ（NBC）
  SW としては共有変数，HW としてはレジスタに相当し，アクセスはノンブロックで行われる
- ブロック通信プリミティブ（BC）
  SW としては OS の同期通信機能，HW としては FIFO に相当し，アクセスはブロックで行われる
- メモリプリミティブ（MEM）
  内部または外部メモリに対してノンブロックでアクセスする

SystemBuilder を用いて開発する場合，設計者はプロセスや通信プリミティブの数などを yaml 形式で SDF（SystemDeFinition）ファイルに記述する．プロセスごとに SW，HW へ分割指定を行うことで，通信プリミティブも自動的に割り振られる．SystemBuilder の SystemBuilder/SysGen（以下，SysGen と呼ぶ）を実行することでプロジェクトの合成を行う．SysGen は C ソースコードと SDF ファイルを入力として実行する．これにより HW プロセスを接続するための HDL，インタフェース構成ファイル，FPGA での動作に必要なファイルを生成する．実機に実装した際のシステム構成を**図 2** に示す．

ここで，図 1 で示すように，本研究開始以前より SystemBuilder が提供している 3 つの HW 性能評価フローについて紹介する．() の時間は 500 行程度の C コードのビルドから性能評価実行までにかかる時間を示している．なお，SL-Sim にかかる時間は，無償の HDL シミュレータを使用した場合と，高速実行のために有償の HDL シミュレータを使用した場合の 2 通りがある．

- 実機評価（12 分）
  CPU（SW）と FPGA（HW）で実行
  SW も含めた実際の実行時間を得ることができるが，実機を用意しなければならず，得られる情報に限りがある．
- SL-Sim（5 分～10 分）
  ISS（Instruction Set Simulator）と HDL シミュレータによる SW と HW の協調シミュレーション
  各プロセスの実行時間をサイクル精度で取得することができるが，シミュレーションは低速であり，高速に実行するためには高価なツールが必要になる．
- Csim（1 分）
  ISS を用いた，全てのプロセスを SW としたシミュレーション
  高速かつ無償でシミュレーションが可能であるが，あくまでアルゴリズムレベルの正しさの確認であるため，性能評価に使用することはできない．

2.2 QEMU

Xilinx QEMU とは，Xilinx が QEMU を独自拡張したもので，Xilinx によって公開されている GitHub リポジトリ [5] に公開されている．これ以降，本論文では Xilinx QEMU のことを QEMU と呼ぶ．Xilinx は，QEMU と HW シミュレーション環境の間のインタフェースとして，SystemC で記述された libSystemctlm-soc [6] を提供している．この libsystemctlm-soc は QEMU と HW 間の通信に RP（Remote-Port）と TLM（Transaction-Level Modeling）を使用している．RP とは UNIX ドメインソケットと共有メモリを使用したプロトコルフレームワークである．また，TLM とは，抽象化したデータやアドレスなどの情報を一括して転送する，システムコンポーネント間の通信のモデリング手法である．このインタフェースによって QEMU と HW は独立して実行される．

3. 関連研究

システムレベル設計における HW 性能評価手法は，高位合成により生成される合成レポート等を用いて性能評価するものと，高位合成後協調シミュレーションによって性能評価するものに分類できる．

高位合成ツールが生成する合成レポート等を用いて性能評価するものとして，HLScope[7] やそれを改良した HLScope+[8] がある．これらは，高位合成ツールである Vivado HLS が出力する合成レポートと，関数やループのブロック依存関係解析の結果により実行サイクル数を見積もる．しかしこれらの手法には，設計者が対象としている FPGA とそのメモリに関して専門知識を持っていないと解析できないという欠点がある．

Vol.2021-ARC-244 No.46
Vol.2021-SLDM-194 No.46
Vol.2021-EMB-56 No.46
2021/3/26

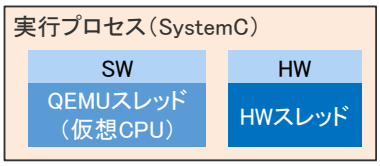

図 3 F-Covip のシミュレーション構造

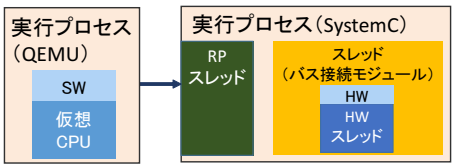

図 4 提案手法のシミュレーション構造

同様に，高位合成ツールが生成する，合成レポートとスケジューリング情報を使用してシミュレーションコードを生成，C シミュレーションによって，高速かつ正確に性能を見積もるフローを提案した FLASH[3] がある．本研究ではこの手法の再現を試みた．しかし，シミュレーションコードの一部は記載されていたが，全体像がわからず，最終的なシミュレーションコードが未記載であった．また，高位合成ツールが生成するスケジューリング情報と元の C コードとの対応付けが困難，終了状態がないデザインについての言及がないことにより，再現が困難であると判断した．そこで本研究では FLASH の再現，利用を断念し，新たに協調シミュレーションによる性能評価手法を検討した．

本研究と同様に SystemC シミュレーション環境で QEMU を利用して性能評価した研究として VPSim[9] や F-Covip[10] がある．これらは SW を QEMU で，HW を SystemC で再現し，両者を TLM を用いてシミュレーションする手法である．TLM を用いることで抽象度を高め，高速実行が期待できる．また SystemC はクロックレベルで記述されるため，サイクル精度で性能評価可能である．これらの手法は，サイクル精度で高速に性能評価するために**図 3** のように QEMU をスレッドとして SystemC 上で実行させる．そのため，実現には QEMU の理解が必要であり，QEMU の仕様変更やターゲット変更の度に SystemC の修正が必要になる．本研究では，**図 4** のように QEMU と SystemC を独立させてシミュレーションするため，QEMU の仕様変更の際の修正は一部で済む．また HW 部分を SystemC で記述するため，これらの先行研究と同様にサイクル精度で性能評価可能である．

4. SystemC を用いた協調検証環境の実現

本章では，本研究で提案する協調検証環境の概要と実現までに検討した項目について説明する．

4.1 設計方針

SystemBuilder が提供している SL-Sim では，HDL シミュレータにより協調シミュレーションを実行している．SL-Sim で使用可能な HDL シミュレータは以下である．

- Vivado Sim（Vivado シミュレータ）
  Xilinx が提供している，混合言語シミュレータである．無償で利用可能だが低速である．
- Questa（Questa Sim）
  Mentor Graphics が提供している，混合言語シミュレータである．グローバル・コンパイル/シミュレーション最適化アルゴリズムを適用することで高速実行が可能だが，高価である．

1 章で述べたように，高価なツールの使用は SystemBuilder の普及の障害となっている．しかし，無償の HDL シミュレータでの実行は低速であり，設計生産性の低下を招く．そこで本研究では，HDL シミュレータを使用しない性能評価手法として，SystemC を用いた協調シミュレーションを検討した．

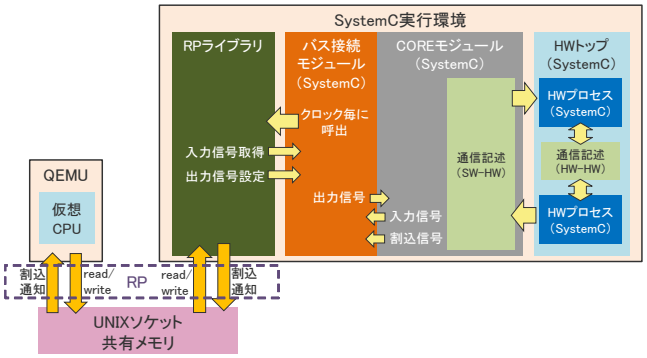

図 5 提案手法のシミュレーション構成

4.2 提案手法の概要

提案手法の導入により，図 1 の黄色で記述されている要素を新たに追加した．また，紫色で記述されている要素は，提案手法の導入により記述の変更があったことを示している．本研究では，無償で利用可能な高位合成ツールである Vivado HLS での高位合成を可能にし，SystemC を生成する．その SystemC と，QEMU の通信に RP を利用した協調シミュレーション実行環境を実現した．

図 5 に提案手法のシミュレーション構成を示す．SW 側は SL-Sim と同様にエミュレータとして QEMU を使用する．HW 側は以下の SystemC で記述されたファイルを，SystemC 実行環境上で実行する．SW と HW は RP を用いて通信する．

- HW プロセス：高位合成ツールによって生成される
- HW トップ：HW プロセス間の接続について記述されている
- CORE モジュール：HW プロセスと SW プロセス間の接続について記述されている
- バス接続モジュール：HW トップと RP ライブラリの

接続について記述されている
- RP ライブラリ：QEMU と SystemC の接続について記述されている
- 通信記述：BC，NBC，MEM の振る舞いについて記述されている

協調検証環境実現までに検討した項目は次の通りである．
(1) SystemBuilder の Vivado HLS 対応
(2) Vivado HLS の SystemC 生成機能の評価
(3) SystemBuilder の SystemC 生成機能の実現
(4) RP ライブラリの設計
以降，各項目について説明する．

4.3 SystemBuilder の Vivado HLS 対応

SystemBuilder はこれまで，有償の高位合成ツールである eXcite[11]，CWB[12] に対応しており，Vivado HLS に対応していなかった．Vivado HLS は eXcite，CWB とは異なる外部インタフェース記述があるため，SystemBuilder のプロセス間通信機構に対応した Vivado HLS 向けの通信ライブラリを作成する必要がある．通信ライブラリとは送受信するデータの値や ack 信号，stb 信号などを制御するためのクロックレベル記述である．そこで，本研究では以下の 2 点に注目して，Vivado HLS 向けの通信ライブラリの検討を行った．

- 各ポートの生成と初期化
  生成したポートに対して，入出力の指定や初期化が必要である
- クロックの挿入方法と位置
  BC 通信では，各ポートの値の確認，代入に順序が決まっており，クロックを適切な箇所に適切な長さ挿入する必要がある

4.3.1 Vivado HLS 向けの通信ライブラリ

Vivado HLS 向けの通信ライブラリを，以下の流れで検討した．作成した Vivado HLS 向けの通信ライブラリのうち，例として BC の READ 記述を**図 6** に示す．

- ポートの生成方法
  - 宣言方法
    グローバル変数として宣言，ポートを生成する．
  - ポートに対する適切な指示子の選択
    Vivado HLS では生成されたポートに対して，ハンドシェイクを目的として複数のプロトコル信号が同時に生成される．しかし，これらのプロトコル信号の一部は SystemBuilder の通信機構との接続において不要である．そこで #pragma HLS INTERFACE ap_none 指示子を使用して不要なプロトコル信号の生成を抑制した．さらに，Vivado HLS では自動最適化により，同一のポートに対する複数回のアクセスが 1 回にまとめられる．これを回避するため，各ポートの宣言時に volatile を指定した．

```
1  volatile uint1 *bc_ack_i;
2  volatile uint1 *bc_stb_o = 0;
3  volatile uint32 *bc_dat_i;
4  volatile uint1 *bc_we_o = 0;
5  #pragma HLS INTERFACE ap_none port=bc_ack_i
6  #pragma HLS INTERFACE ap_none port=bc_stb_o
7  #pragma HLS INTERFACE ap_none port=bc_dat_i
8  #pragma HLS INTERFACE ap_none port=bc_we_o
9
10 void BC_READ(uint32 *data_ptr){
11 BC_READ:{
12      #pragma HLS protocol fixed
13      volatile uint32 data;
14      volatile uint1 ack_i = 0;
15      *bc_stb_o = 1;
16      ap_wait();
17      ack_i = *bc_ack_i;
18      while (ack_i == 0){
19          ack_i = *bc_ack_i; }
20    data = *bc_data_i;
21    *bc_stb_o = 0;
22    ap_wait();
23    *data_ptr = data;
24 }}
```

図 6 Vivado HLS 向け通信ライブラリ

- クロックの挿入方法および挿入位置
  ap_wait() により，図 6 の 16，22 行目に 1 クロック挿入することで，通信順序を固定した．
- 実行順序の固定
  BC 通信はブロッキング通信であり，通信に関する以下の処理の間，それ以降の処理が実行されてはならない．
  (1) READ が呼び出されると， stb が立ち上がる
  (2) その後有効なデータがあることを示す ack が立ち上がるまでデータの待ちが発生する
  そのために，指示子 #pragma HLS protocol fixed を使用し，READ 処理と他のコードの並列実行を抑制する．

NBC，MEM の通信ライブラリに関しては，BC の通信ライブラリを以下のように変更することで実現した．

- NBC： プロセスの待ちを行うポートの削除
- MEM： プロセスの待ちを行うポートの削除，およびアドレスを指定するポートの追加

4.4 Vivado HLS の SystemC 生成機能の評価

Vivado HLS では SystemC 生成に関するドキュメントがほぼ存在しない．そこで Vivado HLS で生成された SystemC を協調シミュレーションに利用できるか確かめるため，C ソースコードを Vivado HLS で高位合成し SystemC を生成，コンパイルして PC 上でシミュレーションした．

シミュレーション結果

PC 上でシミュレーションを実行すると，SL-Sim と同

じ結果，実行時間を取得することができた．

しかし評価を進める中で，生成された SystemC にはサブプロセスの定義が記述されたヘッダファイルが不足し，実行できないことがあった．これに関して調査したところ，公式のリファレンスには説明がないが，Vivado HLS では単純なサブプロセスの定義ファイルの生成を省略することがあるとわかった．そこで，この不足しているサブプロセスに関しては，HDL で記述された同等の動作をするファイルが生成されるため，それをもとに手作業でサブプロセスの定義を作成した．手作業で作成したサブプロセスを用いて PC 上で実行した結果，SL-Sim と同じ結果，実行時間を取得することができた．この結果から，サブプロセスが不足することがあるものの，不足するサブプロセスの定義は単純なものであり，手作業で作成して補うことができるため，Vivado HLS が生成した SystemC を協調シミュレーションに利用することに問題はないと判断した．

4.5 SystemBuilder の SystemC 生成機能の実現

SystemBuilder では HDL で記述された，HW プロセス間をつなぐ HW トップやプロセス間の通信記述を自動生成している．SystemC 実行環境での性能評価には，SystemC で記述された HW トップやプロセス間の通信記述が必要になる．本節ではその SystemC 生成について説明する．

記述内容

SysGen により生成される HDL ファイルは以下である．

- SL-Sim 実行時に最上位となる main ファイル
- 各プロセスとその間の通信について記述されている HW トップ
- HW トップと SW の間の通信について記述されている CORE モジュール
- その他使用している通信プリミティブに応じた記述を持つ通信記述ファイル

なお，プロセス間の MEM 通信に伴う RAM の定義ファイルは生成されなかった．SystemBuilder では市販ツールのライブラリを使用しているためである．そこで本研究では，新たに SystemC で RAM の定義ファイルを作成した．

各 HDL ファイルを参考に，SystemC ファイルを生成した．このとき注目するべきポイントは以下の 4 点である．

(1) 信号および変数のデータ型
(2) シミュレーション制御構文の利用
(3) SC_METHOD
(4) SC_SIGNAL_WRITE_CHECK

それぞれについての概要，および生成した記述内容について説明する．

(1) 入出力信号および変数のデータ型

Vivado HLS が生成する各 HW プロセスの SystemC の各信号は，クロックを除き sc_logic もしくは sc_lv で宣言されている．これらは 0，1，x，z の 4 値を扱うデータ型で，使用するとシミュレーション時間が長くなる．また，x，z が存在すると，論理合成時のシミュレーションと結果が一致しない可能性があるため，x，z は存在しないことが望ましい．そこで本研究では，sc_logic もしくは sc_lv の使用を極力控えることにした．しかし，Vivado HLS が生成する SystemC の宣言の変更は自動化の際の障害になると判断し，Vivado HLS で生成された SystemC 記述で使用されている入出力信号のみを sc_logic および sc_lv で宣言し，それ以外は高速化のために， sc_uint を使用することにした．このとき，SystemC の仕様として，クロックは sc_in_clk を用いて記述する．

(2) シミュレーション制御構文の利用

SystemC ではシミュレーション記述のためにシミュレーション制御構文を持っている．SystemC 生成に使用したシミュレーション構文は以下である．

- sc_clock

  以下のように記述して 10ns 周期のクロック信号"CORE_CLK"を生成する．

  sc_clock CORE_CLK("CORE_CLK", 10, SC_NS);

- sc_start，sc_stop

  シミュレーションの開始，終了のための制御構文で，sc_start() でシミュレーションを開始し，sc_stop() が呼ばれるまで実行し続ける．

- sc_trace

  波形情報をファイル出力する制御構文である．以下のように記述した信号は，シミュレーション後，波形ビューワを使用して波形を確認できる．

  sc_trace(＜ファイル名＞, ＜信号名＞, "＜ファイルでの表示名＞");

(3) SC_METHOD

SystemC 生成の重要な点として，assign 文と always 文の書き換えがある．ここで assign 文と always 文に関して簡単に説明する．assign 文は，1 ステートメントで記述できる組合せ回路の記述に使用する．信号同士を接続するため，右辺の値が変化した場合，その変更は左辺にも伝わる．always 文も assign 文と同様，組合せ回路の記述に使用する．assign 文とは異なり，値を代入するタイミングを指定する必要がある．そのため，@(入力信号，…) と記述して，入力信号のいずれかに変化があるときに always 文内部の処理が実行される．

本研究では，assign 文，always 文を再現するために，SC_METHOD を用いた．このとき，SC_METHOD は SC_CTOR マクロを使って定義したコンストラクタ内に記述し，SC_METHOD の引数にはその処理内容を定義した関数名を指定する．SC_METHOD は，sensitive リストに信号を追加することで，その処理が動くきっかけを指定できる．なお，各 sensitive リストは直前

の SC_METHOD に対して有効であるため，1 つのコンストラクタ内で複数の SC_METHOD と sensitive リストを登録できる．

(4) SC_SIGNAL_WRITE_CHECK

SystemC では，ある変数に対して複数のプロセスからアクセスする記述があると実際の競合の有無に関わらずエラーを出力する．このエラーに対しては，環境変数 SC_SIGNAL_WRITE_CHECK を DISABLE に設定することで回避する．

4.6 RP ライブラリの設計

QEMU と SystemC 実行環境との接続に関する記述について説明する．

4.6.1 インタフェース記述検討の概要

QEMU を用いた協調シミュレーションには，QEMU からの信号をクロックレベルに変換する仕組みが必要になる．そこで本研究では，2.2 節で紹介した，Xilinx によって GitHub リポジトリに公開されている libsystemctlm-soc[13] の利用を検討した．

libsystemctlm-soc では QEMU と SystemC 実行環境との通信に RP と TLM を使用している．そのため, SystemC 実行環境で協調シミュレーションを実行すると，QEMU から read/write 要求を出して， SystemC のモデルに要求が受け付けられるまで，RP と TLM の時間が必要となる．この RP と TLM を使用した通信は低速であると述べている既存研究 [10] もある．そこで本研究では，RP に比べて TLM 変換の実行オーバヘッドの方が支配的であると予想し，TLM への変換をしない方式での実現を検討した．そこで SL-Sim と同様に，RP からクロックレベルへ直接変換するモジュールを利用することで，TLM へ変換せず QEMU と SystemC 実行環境を接続した．

4.6.2 記述内容

SL-Sim では libsystemctlm-soc のうち, HDL シミュレータと QEMU の通信のために，SystemC と Xilinx が提供している SoC に依存している部分を取り除いた RP ライブラリを実装していた．SL-Sim では，この RP ライブラリを共有ライブラリとして，QEMU と通信している．

ここで，QEMU から RP パケットを受信，バス要求への変換，RP での応答送信までの流れを次に示す．なお，この処理の間 QEMU は停止している．1〜3 と 4〜6 は並行して別スレッドで動作しており，1〜3 は RP ライブラリ，4〜6 は バス接続モジュールで行われる．

(1) RP からのコマンドを受信する．
(2) read/write コマンドの場合，コマンドごとに対応した処理を行う．この時実際の処理はコールバック登録された関数に委譲する．
(3) HW のバスに read/write 要求が受け付けられる．
(4) read/write のバストランザクションを出力し，HW からの信号を待つ．
(5) HW からの信号を受け付けたら，RP 応答の処理を行う関数を呼び出す．read の場合はリード値を渡す．
(6) RP 応答を送信する．

そこで本研究でも, SL-Sim の共有ライブラリを利用する．SL-Sim 向けに SystemBuilder で生成されていた，HW との接続処理が記述された HDL ファイルをもとに SystemC のバス変換モジュールの実装を行った．SL-Sim と提案手法は次の点で異なる．RP パケットを受信してからバス要求への変換，RP 応答送信まで手順のうち，手順 4 の HW からの信号生成に関して，SL-Sim では HDL シミュレータが生成していたが，提案手法では CORE モジュールからの信号を受けて，バス変換モジュールが生成する．

5. 評価

本章では RP と TLM の実行オーバヘッド，および本研究で提案している協調検証環境の評価について説明する．

5.1 RP と TLM の実行オーバヘッド

本研究では，QEMU と HW を接続する際の，RP と TLM の実行オーバヘッドのうち，TLM への変換オーバヘッドが支配的と予想し，バス変換モジュールを実装した．本節では，その予想が正しいか確認するために，RP と TLM の実行オーバヘッドをそれぞれ計測した．

5.1.1 計測方法と計測環境

評価対象を 2.2 節で紹介した libsystemctlm-soc[13] とし，CPU i7-8700K，Ubuntu 16.04.7 の環境で計測した．測定範囲は以下の通りである．

- RP

  QEMU から要求を出して，RP で処理するまでの時間を計測する．そのために，QEMU からデータ送信を 100,000 回行った時間をストップウォッチで計測し 1 回あたりのデータ送信時間を算出した．この計測時間には QEMU-HW 間の TCP/IP 通信の時間も含んでいる．

- TLM

  RP で要求を処理してから，TLM へ変換, SystemC モデルに要求が受け付けられるまでを計測する．コード中の read コマンドの前後に時間を取得する関数 sc_time_stamp を挿入して計測した．

5.1.2 計測結果

1 回あたりの平均実行オーバヘッドを**表 1** に示す．RP は TCP/IP の通信時間を含んでいるにも関わらず，TLM への変換の方が実行オーバヘッドの方が大きかった．この結果から，libsystemctlm-soc において，TLM への変換は実行オーバヘッドが大きく，TLM への変換を行わないことで高速化が可能であるといえる．

Vol.2021-ARC-244 No.46
Vol.2021-SLDM-194 No.46
Vol.2021-EMB-56 No.46
2021/3/26

表 1　RP と TLM の 1 回あたりの平均実行オーバヘッド

	RP	TLM
平均実行オーバヘッド（µs）	32.05	38.00

表 2　評価環境

Vivado HLS	2019.1
SystemC	2.3.3
g++	7.4.0
Questa	2019.4
Vivado Sim	2019.1
OS	Ubuntu 18.04.5 LTS
CPU	AMD Ryzen(TM) 9 3950X 16 コア

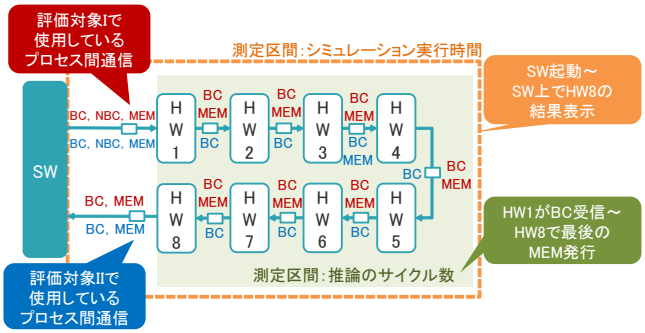

図 7　評価対象の構成

5.2　協調検証環境

CNN 手書き数字推論ソースコード [4] を評価対象として，本研究で提案した SystemC を用いた協調シミュレーションを実行する．シミュレーション実行時間や測定精度，測定粒度，ツールコストに関して SL-Sim と比較し，目的を達成できたか確認する．評価環境を**表 2** に示す．

5.2.1　評価対象

先行研究 [4] で紹介されている CNN の手書き数字推論ソースコードのうち，以下の高速化手法が適用されている 2 つのソースコードを評価対象とした．いずれの評価対象も C 言語で記述された 8 つの HW プロセスを持つ HW で，**図 7** の構成で通信している．

- 評価対象 I：パイプライン処理
  各 HW プロセスを並列実行させることで推論の高速化を実現している．プロセス間の通信は，MEM でデータを，BC で次の HW プロセスの起動信号を送信している（図 7 の赤色）．
- 評価対象 II：パイプライン処理 + FIFO + データ通信の効率化
  評価対象 I の高速化手法に加えて，次の 2 つの高速化手法を適用することで推論の高速化を実現している．
  – プロセス間の通信を MEM から BC に変更（図 7 の青色），FIFO 化することで各 HW の並列性を高める
  – 複数のデータをまとめて一度に送信することでデータ通信の効率を向上させる

本研究では，次の 2 つの測定区間での計測を行った．

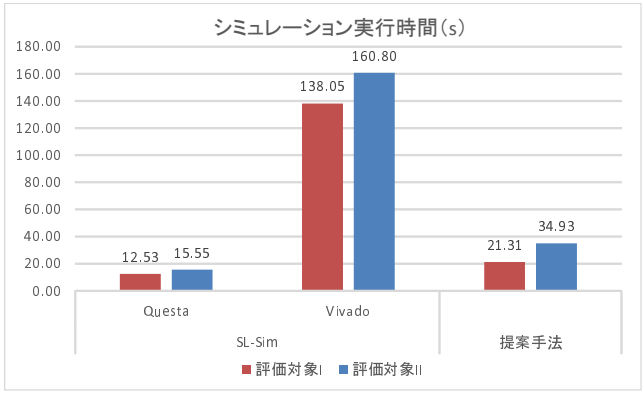

図 8　評価対象によるシミュレーション実行時間の比較

- シミュレーション実行時間
  SW を起動してから，SW 上で HW8 の結果が表示されるまでの，実際にユーザが待っている時間
- 推論のサイクル数
  HW1 が 最初の BC を受信してから，HW8 で最後の MEM が発行されるまでにかかった，シミュレーションされている HW の実行サイクル数

5.2.2　評価手順

提案手法における評価手順は次の通りである．

(1) SystemBuilder の SysGen を用いて合成を行う
(2) 4 章で述べた HW トップや通信記述などの SystemC を手作業で作成する
(3) 高位合成により生成される SystemC と，HW トップ，通信記述などのファイルを，g++ を用いてコンパイル，実行ファイルを生成する
(4) SW の記述をコンパイラを用いてコンパイル，QEMU を起動する
(5) 実行ファイルと QEMU を用いて協調シミュレーションを実行する

5.2.3　評価結果と考察

前述の手順で実行したところ，いずれの評価対象に対しても提案手法は SL-Sim と同等の HW の実行結果として期待する結果を得られた．

図 8 に両評価対象のシミュレーション実行時間を比較した結果を示す．評価対象 I のシミュレーション実行時間比は，有償の HDL シミュレータである Questa よりは 1.70 倍低速ではあるが，無償の HDL シミュレータである Vivado Sim より 6.48 倍高速であった．また，評価対象 II でも Questa よりは 2.25 倍低速ではあるが，Vivado Sim より 4.60 倍高速であった．この結果より，提案手法は無償の HDL シミュレータ使用時より高速化できたといえる．これは提案手法において，HDL シミュレータを使用せず，SystemC によりシミュレーションしているため，高速化できたと考えられる．ただし，Questa はグローバル・コンパイル/シミュレーション最適化アルゴリズムを適用しているため，HDL シミュレータを使用していても高速であ

表 3 1 回の設計空間探索にかかる時間（s）

	SL-Sim		提案手法
	Questa	Vivado Sim	
設計空間探索時間	299.52	461.45	335.91 （+150.58）

ると考えられる．

さらに SL-Sim と提案手法のシミュレーション実行時間比を比較すると，提案手法により，評価対象 I は評価対象 II よりも高速化できている．この理由として，評価対象 I は評価対象 II と比較して BC ではなく MEM を多く使用していることが挙げられる．SL-Sim では MEM 通信時，論理合成ツールが提供している RAM を使用している．論理合成ツールが提供している RAM は，実機に実装する際に FPGA 内部のブロック RAM（BRAM）として実装されるよう，実機向けの記述になっている．しかし，提案手法で MEM 通信時に使用している RAM はシミュレーション向けに記述しているため，高速化できたのだと考えられる．

C ソースコードの変更から高位合成，シミュレーション実行終了までにかかったツール実行時間を設計空間探索時間として，**表 3** に示す．提案手法の時間については，HW トップなどの SystemC 生成にかかった時間を除いたものであり，() 内の時間は不足しているサブプロセスの作成にかかった時間を示している．C ソースコードのみの変更であれば，変更後の C ソースコードに対して高位合成を行うだけで性能評価が可能であり，SystemBuilder による新たな SystemC 生成は必要ない．しかし，2.1 節で説明した SDF の書き換えを伴う変更の場合，高位合成だけでなく，新たに SystemC 生成を行う必要があり，提案手法による設計空間探索時間は 30 分以上になる．そのため，これらの SystemC 生成時間の短縮のために，SystemBuilder による SystemC 生成の自動化が求められる．

次に，測定精度について確認するため，2 つ目の測定区間である，推論のサイクル数の結果について述べる．Questa，Vivado Sim，提案手法のいずれも推論にかかったサイクル数は一致していた．この結果から，測定精度を維持出来ていると確認できた．また評価粒度に関しても，各プロセスの信号情報を波形として，各プロセス・信号をサイクル精度で取得するという目標を達成できた．

以上の結果より，測定粒度，精度を維持したまま，ツールコストを 0 にできた．さらにツールコストが同じく無償である，Vivado Sim 使用時より 4.60 倍高速化を達成した．

6. おわりに

本研究では，システムレベル設計における，高速かつサイクル精度で性能評価可能な手法の低コストでの実現を目的として，QEMU と SystemC を用いた協調検証環境を検討した．SL-Sim と比較して，有償ツール使用時よりは 2.25 倍低速であったが，無償ツール使用時より 4.60 倍高速に実行できた．このとき測定精度および測定粒度は SL-Sim と同等であった．また，有償ツールを使用していないため，ツールコストを 0 に削減した．これにより，無償ツールのみを用いた協調検証環境を実現し，サイクル精度での HW 性能評価の高速化を達成した．

今後は通信記述の抽象度引き上げ等によるさらなる高速化を目指す．また，シミュレーションに必要な SystemC で記述されたファイルの自動生成を可能にし，SystemBuilder へ導入したい．

参考文献

[1] 久米寛司，飯間　豊，本田一成，細谷雅寿：SoC 組込みソフトウェアの開発, 沖テクニカルレビュー, Vol. 70 (2003).
[2] 本田晋也，冨山宏之，高田広章：システムレベル設計環境: SystemBuilder，電子情報通信学会論文誌 D，Vol. 88, No. 2, pp. 163–174 (2005).
[3] Chi, Y., Choi, Y.-k., Cong, J. and Wang, J.: Rapid cycle-accurate simulator for high-level synthesis, Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 178–183 (2019).
[4] 岡本卓也，山本椋太，本田晋也，中本幸一，若林一敏：高位合成による小規模 FPGA 向け DNN 推論器の設計，第 50 回組込みシステム合同研究発表会（ETNET2019）(2019).
[5] Xilinx/qemu: Xilinx QEMU. `https://github.com/Xilinx/qemu`（参照 2021 年 2 月 17 日）.
[6] Xilinx: Xilinx Wiki / QEMU / QEMU SystemC and TLM CoSimulation. `https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842109/QEMU+SystemC+and+TLM+CoSimulation`（参照 2021 年 2 月 17 日）.
[7] Choi, Y. and Cong, J.: HLScope: High-Level Performance Debugging for FPGA Designs, 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 125–128 (2017).
[8] Choi, Y., Zhang, P., Li, P. and Cong, J.: HLScope+,: Fast and accurate performance estimation for FPGA HLS, 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 691–698 (2017).
[9] Charif, A., Busnot, G., Mameesh, R., Sassolas, T. and Ventroux, N.: Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration (2019).
[10] Díaz, E., Mateos, R. and Bueno, E.: Virtual Platform of FPGA based SoC for Power Electronics Applications, 2019 IEEE 28th International Symposium on Industrial Electronics (ISIE), pp. 1371–1376 (2019).
[11] Y Explorations: eXCite 公式ホームページ. `http://www.yxi.com/old/Japanese/productsJ.html`（参照 2021 年 2 月 17 日）.
[12] NEC: CyberWorkBench 公式ホームページ. `https://jpn.nec.com/cyberworkbench/index.html`（参照 2021 年 2 月 17 日）.
[13] Xilinx/libsystemctlm-soc: libsystemctlm-soc. `https://github.com/Xilinx/libsystemctlm-soc`（参照 2021 年 2 月 17 日）.