Vol.2013-ARC-206 No.25
2013/8/1

ArchHDLで記述したハードウェアの論理シミュレーションの高速化

金子 達哉[1]　佐藤 真平[2]　吉瀬 謙二[2]

概要：RTL モデリングのための新しい言語として開発された，ハードウェアのレジスタを変数，ワイヤを関数として扱う ArchHDL の高速化を行う．ArchHDL を用いることで，Verilog HDL に近い RTL モデリングを実現できる．本稿では，ライブラリ中の分岐の削減，メモリ配置の工夫，OpenMP による並列化といった高速化手法を提案し，実装する．4096 個のカウンタ回路を用いた評価から，高速化手法を適用した ArchHDL のシミュレーションは商用の Verilog シミュレータである VCS より 56.7 倍高速であった．

1. はじめに

プロセッサなどのハードウェア設計は，アーキテクチャ設計・論理設計・回路設計・物理設計といったフローで行われる．アーキテクチャ設計と論理設計においては，RTL (Register Transfer Level) のシミュレーションが不可欠である．このために Verilog HDL などのハードウェア記述言語が用いられることが一般的である．

我々は C++ 言語上で RTL モデリングを行う新しい言語である ArchHDL を提案している [1]．ArchHDL を用いることで Verilog HDL に近い記述でハードウェアの論理検証を行うことができる．

ArchHDL で記述したハードウェアのシミュレーションはオープンソースの Verilog シミュレータである Icarus Verilog [2] と比較して高速である．しかし一部のハードウェアシミュレーションにおいて有償の Verilog シミュレータである Cadence 社の NC-Verilog より高速であったが，同じく有償の Synopsys 社の VCS [3] より高速ではないことがあった．

そこで ArchHDL に分岐の削減，メモリ配置の工夫，OpenMP による並列化といった高速化手法を適用する．

本稿では Verilog シミュレータの Icarus Verilog, NC-Verilog, VCS と比較して，今回の高速化手法の有用性を評価する．

本稿の構成は以下の通りである．2 章で，ArchHDL の概要を述べる．3 章で，ArchHDL の高速化手法を提案する．4 章で，様々な Verilog シミュレーションツールと比較して ArchHDL と今回の高速化手法の評価を行う．5 章でまとめる．

[1] 東京工業大学 工学部情報工学科
Department of Computer Science, Tokyo Institute of Technology
[2] 東京工業大学 大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology

```
1  class Counter : public Module {
2   public:
3    wire<uint> out;
4    reg<uint> counter;
5    void Init() {
6      out = [=]() { return counter(); };
7    }
8    void Always() {
9      counter <<= (counter() + 1) & 0xff;
10   }
11 };
```

図 1: ArchHDL による 8 ビットカウンタ回路の記述

2. ArchHDL の概要

2.1 ArchHDL による RTL モデリング

ArchHDL はハードウェアの RTL モデリングのための言語である．Verilog HDL に近い記述方法を目指している．ユーザはライブラリにより提供される Module クラス，reg クラス，wire クラスおよび C++11 のラムダ関数を用いてハードウェアを記述する．gcc では，バージョン 4.5 より標準ライブラリとしてラムダ関数が利用できるのでこれ以上のバージョンを要求する．

図 1 に，ArchHDL を用いて記述した 8 ビットカウンタ回路のコードを示す．また，**図 2** に，Verilog HDL で記述した 8 ビットカウンタ回路のコードを示す．ArchHDL において，Module クラスを継承して定義されるクラス（Module 子クラス）は，Verilog HDL におけるモジュールに相当する．同様に reg クラス，wire クラスは，それぞ

```
1  module Counter(CLK, out);
2      input CLK;
3      output [7:0] out;
4
5      reg [7:0] counter;
6      assign out = counter;
7      always @(posedge CLK) begin
8          counter <= counter + 1;
9      end
10 endmodule
```

図 2: Verilog HDL による 8 ビットカウンタ回路の記述

れ Verilog HDL におけるレジスタ，ワイヤに相当する．

Module 子クラスは，そのメンバ関数として Init 関数および Always 関数を定義する必要がある．これは，ライブラリにより強制されており，いずれかの関数が必要でない場合でも空の関数を定義する必要がある．

Init 関数には，モジュール内のすべてのワイヤへの継続代入の定義を記述する．これは，Verilog HDL においてモジュール内で定義されている assign 文をすべてこの関数内に記述することに相当する．wire クラスのインスタンスにラムダ関数を代入することでワイヤへの継続代入を記述することができる．図 1 では，6 行目で wire クラスの変数 out に reg クラス の counter の値を返すラムダ関数（`[=]() { return counter(); }`）を代入している．ここで，reg クラスのオブジェクトは関数として呼び出すことにより，そのサイクルにおける値を取得することができる．このため，先のラムダ関数における counter() によってレジスタの値が取得できる．これは，図 2 の 6 行目に相当する．

Always 関数には，モジュール内のすべてのレジスタへのノン・ブロッキング代入を記述する．ArchHDL では，単一クロックの立ち上がりエッジでの制御のみを対象としている．これは，Verilog HDL における `always @(posedge clock)` ブロック内の記述に相当する．reg クラスのインスタンスに **<<=** 演算子を用いて値を代入することでノン・ブロッキング代入を記述することができる．ArchHDL は，演算子オーバーロードを利用して **<<=** 演算子を Verilog HDL におけるノン・ブロッキング代入に相当する値の代入として実装している．図 1 では，9 行目で reg クラスの変数 counter に自身の値をインクリメントした値を代入している．これは，図 2 の 8 行目に相当する．

ArchHDL では，データ型として C++ の整数型を使用している．図 1 の例では unsigned int 型を利用しているため，8 ビットカウンタを実現するために値をマスクする必要がある．reg クラスと wire クラスのオブジェクトは関数として呼び出すことにより，そのサイクルにおける値を取得することができる．

```
1  class RegisterInterface {
2   public:
3    virtual void Update() = 0;
4  };
5
6  class ModuleInterface {
7   public:
8    virtual void Init() = 0;
9    virtual void Always() = 0;
10 };
11
12 class WireInterface {};
13
14 namespace ArchHDL {
15
16 class Singleton {
17  private:
18   std::vector<RegisterInterface*> registers_;
19   std::vector<ModuleInterface*> modules_;
20   std::vector<WireInterface*> wires_;
21  public:
22   static Singleton& GetInstance(void) {
23     static Singleton singleton;
24     return singleton;
25   }
26   void AddRegister(RegisterInterface* ri) {
27     registers_.push_back(ri);
28   }
29   void AddModule(ModuleInterface* mi) {
30     modules_.push_back(mi);
31   }
32   void AddWire(WireInterface* wi) {
33     wires_.push_back(wi);
34   }
35   void Init() {
36     for (uint i = 0; i < modules_.size(); i++) {
37       modules_[i]->Init();
38     }
39   }
40   void Exec() {
41     for (uint i = 0; i < modules_.size(); i++) {
42       modules_[i]->Always();
43     }
44     for (uint i = 0; i < registers_.size(); i++) {
45       registers_[i]->Update();
46     }
47   }
48 };
49
50 void Step() {
51   static bool init = false;
52   if (!init) {
53     init = true;
54     ArchHDL::Singleton::GetInstance().Init();
55   }
56   ArchHDL::Singleton::GetInstance().Exec();
57 }
58
59 }  // namespace ArchHDL
```

図 3: ArchHDL ライブラリにおける各インタフェースクラスと Singleton クラスと Step 関数の定義

2.2 ArchHDL の実装

ArchHDL のライブラリには，Module クラス，wire クラス，reg クラス，これらの 3 個のクラスのインタフェースクラス，Singleton クラスの 7 個のクラスが定義されている．本章では，標準ライブラリのインクルードをのぞくすべてのライブラリのコードを示しながら ArchHDL の実装について述べる．以降の説明では，ユーザが Module クラスを継承して作成したクラスを Module 子クラスと呼ぶ．

図 3 に RegisterInterface クラス，ModuleInterface ク

ラス，WireInterface クラス，Singleton クラスおよび Step 関数の定義を示す．

ModuleInterface クラス，WireInterface クラス，RegisterInterface クラスはそれぞれ Module クラス，wire クラス，reg クラスのインタフェースクラスである．Singleton クラスが，Module 子クラス，wire クラス，reg クラスのインスタンスをシングルトン・パターンにより一元管理する．これは，ArchHDL のライブラリにおいて核となるクラスである．

Singleton クラスは，メンバ変数として Module クラス，wire クラス，reg クラスのインタフェースクラスのポインタを格納する可変行列をもつ（18 〜 20 行目）．Module 子クラス，wire クラス，reg クラスのインスタンスが生成される際に，そのインスタンスへのポインタが Singleton クラスに渡される．また，ポインタは Singleton クラスに渡される際にそれぞれのインタフェースクラスに自動でアップキャストされる（26 〜 34 行目）．

Step 関数（50 〜 57 行目）は，1 サイクルのシミュレーションを行う関数である．Step 関数を呼び出すと，Singleton クラスの Exec 関数が呼ばれる．ただし，初回 Step 関数の呼び出しのみ Singleton クラスの Init 関数が呼ばれる．Step 関数を繰り返し呼び出すことにより，複数サイクルにわたるシミュレーションが行なわれる．

Init 関数（35 〜 39 行目）は，保持しているすべての Module 子クラスのインスタンスの Init 関数を呼ぶ（37 行目）．

Exec 関数（40 〜 47 行目）は，保持しているすべての Module 子クラスのインスタンスの Always 関数を呼び（42 行目），次に保持しているすべての reg クラスのインスタンスの Update 関数を呼ぶ（45 行目）．

Always 関数によりすべてのレジスタについて次のサイクルにおける値が計算される．Update 関数によりレジスタの値が更新される．この Always と Update の処理によりレジスタのノン・ブロッキング代入を実現する．

2.2.1 reg クラスの定義

図 4 に，reg クラスの定義を示す．reg クラスは，扱うデータ型をテンプレート引数にとるテンプレートクラスである．また，インタフェースクラスである RegiterInterface クラスを継承する．

ArchHDL ではレジスタを変数として扱うため，reg クラスはメンバ変数にテンプレート引数で与えられたデータ型の変数 curr_ と next_ を持つ（5，6 行目）．curr_ は，あるサイクルにおけるレジスタの値で，next_はその次のサイクルのレジスタの値である．Module 子クラスの Always 関数の呼び出しにより，next_に値が代入される．reg クラスのメンバ関数 Update を呼ぶことで，next_ の値は curr_ に反映される（15 〜 20 行目）．これによりレジスタへのノン・ブロッキング代入の挙動を実現する．

```
1  template <typename T>
2  class reg : public RegisterInterface {
3   private:
4    bool set_;
5    T curr_;
6    T next_;
7
8    // copy constructor
9    reg<T>(const reg<T>& other);
10   reg<T>& operator=(const reg<T>& rhs);
11  public:
12   reg(): set_(false), curr_(0), next_(0) {
13     ArchHDL::Singleton::GetInstance().AddRegister(this);
14   }
15   void Update() {
16     if (set_) {
17       curr_ = next_;
18       set_ = false;
19     }
20   }
21   void operator=(T val) {
22     curr_ =  val;
23   }
24   void operator<<=(T val) {
25     set_ = true;
26     next_ = val;
27   }
28   T operator ()() {
29     return curr_;
30   }
31 };
```

図 4: ArchHDL ライブラリにおける reg クラスの定義

reg クラスのオブジェクトへの値の代入をそのメンバ変数 next_ への値の代入とするために，演算子オーバーロードにより <<= 演算子を再定義している（24 〜 27 行目）．<<= 演算子により代入された値は，変数 next_ に格納され，set_ フラグがセットされる．

ArchHDL では，すべての Module 子クラスのインスタンスの Always 関数を呼び出した後に，すべての reg クラスのインスタンス Update 関数を呼び出す．したがって，Always 関数が呼び出されている間に取得できるレジスタの値 curr_ は Update 関数が呼ばれるまで保持されている．

reg クラスのコンストラクタ（12 〜 14 行目）では，メンバ変数を初期化し，自身のポインタを Singleton クラスに渡す処理が行われる．テスト記述や初期値設定のために = 演算子による値の代入も定義されている（21 〜 23 行目）．= 演算子による値の代入は，式が評価された時点で curr_ の値を変更する．reg クラスのインスタンスを関数として呼び出すことで curr_の値を取得することができる（28 〜 29 行目）．

2.2.2 wire クラスの定義

図 5 に，wire クラスの定義を示す．wire クラスは，テンプレート引数として扱うデータ型をとるテンプレートクラスである．また，インタフェースクラスの WireInterface クラスを継承している．

ArchHDL では，ワイヤは関数として扱うため，wire クラスはメンバ変数にラムダ関数 lambda_ を持つ（4 行目）

```
1  template <typename T>
2  class wire : public WireInterface {
3   private:
4    std::function<T ()> lambda_;
5
6    // copy constructor
7    wire<T>(const wire<T>& other);
8    wire<T>& operator=(const wire<T>& rhs);
9   public:
10   wire(): lambda_(nullptr) {
11     ArchHDL::Singleton::GetInstance().AddWire(this);
12   }
13   void operator=(std::function<T ()> lambda) {
14     lambda_ = lambda;
15   }
16   T operator()() {
17     return lambda_();
18   }
19 };
```

図 5: ArchHDL ライブラリにおける wire クラスの定義

```
1  class Module : public ModuleInterface {
2   private:
3    // copy constructor
4    Module(const Module& other);
5    Module& operator=(const Module& rhs);
6   public:
7    Module() {
8      ArchHDL::Singleton::GetInstance().AddModule(this);
9    }
10   virtual void Init() {}
11   virtual void Always() {}
12 };
```

図 6: ArchHDL ライブラリにおける Module クラスの定義

クラスとなっている．このラムダ関数は，テンプレート引数として与えられたデータ型を返す関数である．

コピーコンストラクタの禁止（7，8 行目）と演算子のオーバーロード（13，14 行目）により，wire クラスへの = 演算子による代入はラムダ関数に限定される．これにより，wire クラスのオブジェクトは Module 子クラスの Init 関数で記述されるラムダ関数を保持するクラスとなる．

wire クラスのコンストラクタ（10 〜 12 行目）では，メンバ変数を初期化し，自身のポインタを Singleton クラスに渡す処理が行われる．wire クラスのオブジェクトを関数として呼び出すと，自身の持つラムダ関数を呼び出した結果を返す（16 〜 18 行目）．これにより，wire クラスのオブジェクトを関数呼び出しすることで，そのサイクルでのワイヤの値が取得できる．

2.2.3 Module クラスの定義

図 6 に，Module クラスの定義を示す．Module クラスは，インタフェースクラスの ModuleInterface クラスを継承するクラスである．ArchHDL でハードウェアを記述する際に，このクラスを継承してモジュールを記述する．

コンストラクタ（7 〜 9 行目）では，自身のポインタを Singleton クラスに渡す．ModuleInterface クラスにおいて Init 関数と Always 関数が仮想関数として定義されているため，これらの関数を定義する必要がある．Module クラスは，Module 子クラスのインタフェースとして定義しているため，Init 関数と Always 関数として空の関数を定義している（10，11 行目）．

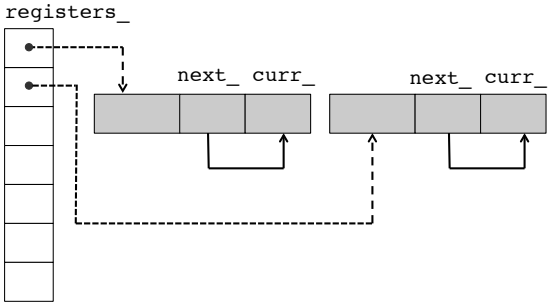

図 7: ArchHDL における reg クラスのインスタンスの処理の様子

3. ArchHDL の高速化手法の提案と実装

3.1 高速化の方針

高速化の方針として逐次プログラミングにおける最適化と並列化の両方を考える．

3.2 逐次プログラムにおける高速化手法

3.2.1 データ変更の有無による条件分岐の除去

図 4 に示した実装では，reg クラスのインスタンスの値を更新する方法としてブロッキング代入とノン・ブロッキング代入の 2 つが存在する．

ブロッキング代入について考える．reg クラスのインスタンスにブロッキング代入が行われた時に curr_ の値を書き換える．

一方でノン・ブロッキング代入について考える．reg クラスのインスタンスにノン・ブロッキング代入が行われた時にメンバ変数 set_ を true にし，メンバ変数 next_ に値を代入する．そして reg::Update 内では set_ が true の時だけ next_ をメンバ変数 curr_ に代入する．これは reg クラスのインスタンスの値を変更したサイクルのみで，その reg クラスのインスタンスの値を次サイクルに移る前に新しい値に更新することを意味する．

図 4 に示した実装では，更新されない reg クラスのインスタンスの curr_ と next_ の値が同じであるため，代入する処理を行う必要はない．よって set_ 変数を用いて不要な代入を避けている．reg クラスのインスタンスの更新頻度が低い回路であればこの実装が効率的である．

提案手法について述べる．この set_ 変数が true の時のみ代入するのではなく，次サイクルに移る前に next_ の値を curr_ に常に代入するようにする．こうすることによって分岐のオーバーヘッドが無くなるため，ノン・ブロッキング代入が頻繁に行われる回路で速度向上が期待できる．

3.2.2 値を配列として格納しメモリ配置を工夫

図 7 は ArchHDL における reg クラスのインスタンスの処理の様子である．図 3 の 44 行から 46 行の処理を表

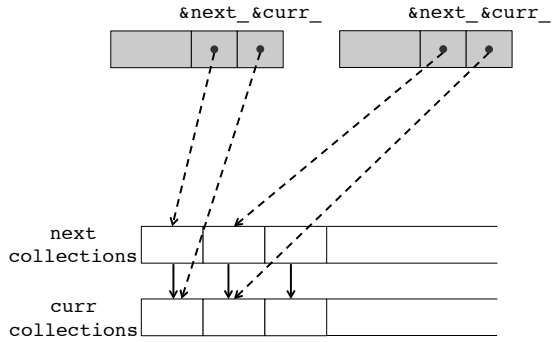

図 8: 値を配列として格納しメモリ配置を工夫した reg クラスのインスタンスの処理の様子

```
 1 void Exec() {
 2 #pragma omp parallel num_threads(8)
 3   {
 4 #pragma omp for
 5     for (uint i = 0; i < modules_.size(); i++) {
 6       modules_[i]->Always();
 7     }
 8 #pragma omp for
 9     for (uint i = 0; i < registers_.size(); i++) {
10       registers_[i]->Update();
11     }
12   }
13 }
```

図 9: Exec メソッド内の for 文を OpenMP で並列化したプログラム

している．reg クラスのインスタンスが灰色に塗られており，左からクラスのメタデータ，next_, curr_ を表している．左側の大きな枠が図 3 の 18 行の `std::vector` 型の registers_ である．実線矢印は代入を表し．点線矢印はポインタ参照を表す．

ArchHDL ではノン・ブロッキング代入をシミュレーションするために registers_ の値を上から順に辿り，reg クラスの各インスタンスのポインタを取得する．そして reg クラスの全インスタンスの reg::Update() メソッドを呼ぶ．

3.2.1 節で述べたデータ変更の有無による条件分岐の除去を行うと reg::Update() メソッド内で行なっている reg クラスのインスタンスの curr_ に next_ の値を代入する処理は毎サイクル全 reg クラスのインスタンスで実行されることになる．

この代入する処理と reg::Update() メソッドの関数呼び出しの 2 つのオーバーヘッドが ArchHDL の高速化を妨げている．

提案手法について述べる．**図 8** に値を配列として格納しメモリ配置を工夫した reg クラスのインスタンスの処理の様子を示す．図 8 は提案手法である．提案手法では全 reg クラスのインスタンスは現在の値と次サイクルの値の実体は持たず，ポインタを保持するように変更している．reg クラスのインスタンスが灰色に塗られており，左からクラスのメタデータ，&next_, &curr_ を表している．&next_, &curr_ は next_, curr_ のポインタである．下の枠が next_, curr_ の値をまとめた配列であり，ここでは next collections, curr collections と呼ぶ．実線矢印は代入を表し，点線矢印はポインタの参照先を表す．

ArchHDL の実装では次サイクルに移る前に行われる curr_ に next_ の値を代入する処理は reg クラスのインスタンスが存在するアドレスを調べる必要がある．しかし値を配列として格納しメモリ配置を工夫すると図 8 に示すように単純な代入となる．また今まで飛び飛びのアドレスに格納されていた next_ と curr_ のメモリ配置がまとまるのでメモリアクセスが連続的に行える．さらに reg::Update() の関数呼び出しが不要となり，関数呼び出しのオーバーヘッドもなくなる．これらの理由により高速化が期待できる．

提案手法の実装について述べる．next collections, curr collections として 2 つの十分大きな `unsigned int` 型の配列を用意する．記述された型に応じて，reg クラスのコンストラクタが next_, curr_ それぞれの領域を next collections, curr collections に確保する．確保する領域は参照の高速化のために 4 バイトの倍数とする．確保された next_ と curr_ のアドレスを取得し，インスタンス内の &next_, &curr_ がそれを保持する．これまで reg クラスの全インスタンスの reg::Update() メソッドを呼び出していたところを next_ collections から curr_ collections の値コピーに変更する．

3.3 並列化による高速化

これまで逐次プログラミングにおける高速化を考えてきたが，本節では並列化による高速化について考える．

図 3 の 40 行〜 47 行に示すように毎サイクル，Module クラスと reg クラスの全インスタンスの Module::Always() メソッドと reg::Update() メソッドが呼び出されている．

図 3 の 41 行〜43 行で実行される Module::Always() メソッドはユーザが自由に記述できる．そのため各 Module クラスのインスタンスで独立に Module::Always() メソッドが実行できる保証はない．しかしここでは独立に実行できると仮定する．独立に実行できる条件は今後の研究課題とする．この場合図 3 の 41 行〜43 行に示している Module::Always() メソッドの実行は並列化が可能である．

図 3 の 44 行〜 46 行で行われるレジスタの更新は図 7 の実線で表されている．各インスタンスで独立に行えるので並列化が可能である．

図 3 の 41 行〜47 行に示している Module クラスと reg クラスの全インスタンスの Module::Always() メソッドと reg::Update() メソッドの実行は並列化が可能である．提案手法について述べる．この部分を並列化する．並列化には OpenMP [4] を用いる．

図 9 は図 3 の 40 行〜 47 行が 8 スレッドで並列化が行われるように OpenMP 指示文を与えたソースコードであ

表 1: 実行環境

	Icarus Verilog, ArchHDL	NC-Verilog, VCS
OS	Ubuntu12.04	CentOS5.9
CPU	Core i7-3770K 3.50GHz	Core i7-3770K 3.50GHz
メモリ	16 GB	16 GB

```
1 unsigned int xor() {
2   static unsigned int y = 2463534242;
3   y ^= (y << 13);
4   y ^= (y >> 17);
5   return (y ^= (y << 5));
6 }
```

図 10: XORSHIFT 法に基づく乱数生成のアルゴリズム

る．2 行目は並列化を何スレッドで行うかを与えている．今回は 8 をスレッド数に指定している．この数字は環境によって変えることができる．4 行目と 8 行目は for 文の実行を並列化する OpenMP 指示文である．

一般に，並列化を行う場合は各スレッドに対して均等に負荷を与えることが重要である．OpenMP では for 文の負荷を分散するスケジュール方法として，静的に決定する static や，動的に決定する dynamic など複数の方法が存在する．一般に，スケジュール方法に dynamic を指定した場合，各スレッドに割り当てられる負荷が均等に近くなるメリットはあるが，オーバーヘッドが大きいデメリットがある．ArchHDL により記述されたハードウェア記述では各モジュールと各レジスタの負荷が大幅に変わらないのであれば，static を指定した方が効率が良いと考えられる．

他にも OpenMP にはオプションとしてチャンクサイズを指定できる．スケジュール方法を static にし，チャンクサイズを指定しなければ，チャンクサイズはループの反復数をスレッド数で割った商とほぼ同じ値になる．

今回の評価ではスケジュール方法は static でチャンクサイズは指定しないデフォルトの設定で行う．

4. 評価

本章では ArchHDL での論理シミュレーションの実行時間を評価し，Icarus Verilog, NC-Verilog, VCS での論理シミュレーションの実行時間と比較する．

表 1 に実行環境をまとめる．評価には同じ仕様の 2 台の計算機を用いる．一台は Icarus Verilog, ArchHDL の評価に用いる．もう一台は NC-Verilog, VCS の評価に用いる．CPU，メモリなどのハードウェアの仕様は同一であるがソフトウェアの制約により異なる OS を利用する．

異なる OS を用いる理由を述べる．NC-Verilog と VCS は RedHat 系のディストリビューションのみをサポートしている．今回は RedHat 系のディストリビューションである CentOS5.9 を用いる．しかし CentOS5.9 に含まれる gcc のバージョンは 4.1.2 である．2.1 節で述べたように，ArchHDL では C++11 のラムダ関数を用いて記述するため gcc のバージョンは 4.5 以上が必要である．その条件を満たす Ubuntu12.04 を評価に用いる．Ubuntu12.04 に含まれる gcc のバージョンは 4.6.3 である．gcc の最適化オプションとして `-O2` を用いる．Icarus Verilog はどちらのディストリビューションでも動作するが，今回は Ubuntu12.04 を用いる．Ubuntu12.04 に含まれる Icarus Verilog のバージョンは 0.9.5 である．VCS のバージョンは vcsC-2009.06 を用いる．NC-Verilog のバージョンは 06.20-s004 を用いる．

今回用いる計算機の CPU の物理コアは 4 コアであるため，OpenMP による並列化はスレッド数を 8 個にして評価する．

評価では，2 つのマイクロベンチマークと，現実的なハードウェアのベンチマークとしてステンシル計算回路 [5] を用いる．Verilog HDL と ArchHDL のためのハードウェアシミュレーションは手作業により作成した．両ハードウェアシミュレーションの出力は同様になるように作成した．

評価結果に用いているラベルの名前について述べる．オリジナルの ArchHDL は **ArchHDL** と表す．3.2.1 節で述べた条件分岐の除去を適用したものを **NO SET** と表す．3.2.2 節で述べたメモリ配置の工夫を適用したものを **MEM MAP** と表す．3.3 節で述べた並列化を行ったものを **PARA** と表す．メモリ配置の工夫と並列を同時に適用したものを **MEM MAP + PARA** と表す．

4.1 マイクロベンチマークによる評価

マイクロベンチマークとしてカウンタ回路と XORSHIFT による乱数生成回路を用いる．

カウンタ回路とは 図 1 に示した 1 サイクルごとに 1 を足す回路である．ハードウェアの規模を増やすためにカウンタの数を指定できるようにした．XORSHIFT による乱数生成回路とはシフトと XOR 演算のみで構成できる XORSHIFT 法に基づく乱数生成器をハードウェア記述によって実装した回路である．**図 10** に XORSHIFT 法に基づく乱数生成のアルゴリズムを C 言語によって実装したものを示す．

図 11 に 4096 個のカウンタ回路の実行時間を Icarus Verilog と比較した速度向上比を示す．縦軸は Icarus Verilog での実行時間を 1 とした速度向上比を示している．

ArchHDL は商用の NC-Verilog, VCS と比較してもかなり高速である．**MEM MAP + PARA** の論理シミュレーション実行時間は NC-Verilog の 58.8 倍，VCS の 56.7 倍高速である．

また今回提案している高速化手法はオリジナルの ArchHDL に比べていずれも効果が出ている．**MEM MAP + PARA** の論理シミュレーション実行時間はオリジナルの

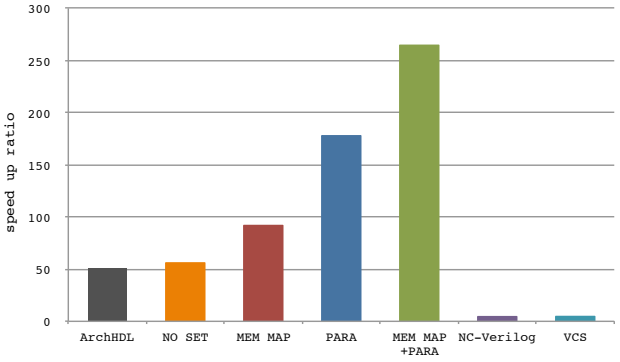

図 11: 4096 個のカウンタ回路の実行時間を Icarus Verilog と比較した速度向上比

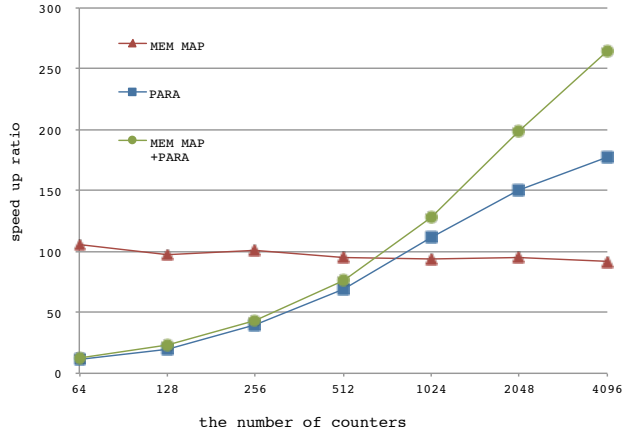

図 12: 高速化手法を適用した ArchHDL と OpenMP を適用したカウンタ回路の実行時間を Icarus Verilog と比較した速度向上比

ArchHDL の 5.23 倍高速である．

図 12 に高速化手法を適用した ArchHDL と OpenMP を適用したカウンタ回路の実行時間を Icarus Verilog と比較した速度向上比を示す．縦軸は Icarus Verilog での実行時間を 1 とした速度向上比を示している．横軸はカウンタの個数である．

MEM MAP は逐次に実行されているので Icarus Verilog と比較した速度向上比はカウンタの個数を変えてもほとんど変わらない．並列化を行った **PARA** と **MEM MAP + PARA** はカウンタの個数が 1024 個以上で **MEM MAP** よりも高速になる．**PARA** より **MEM MAP + PARA** の方が常に高速であるので今回提案している逐次処理での高速化手法は並列化を行った場合でも効果が出ている．カウンタの個数はハードウェアの規模とみなせるため，並列化が有効なのはある程度規模の大きい回路であると言える．

図 13 は XORSHIFT による乱数生成器での実行時間を Icarus Verilog と比較した速度向上比である．試行回数は 524,288 回である．初期値の異なる乱数生成器を 512 個用意している．

ArchHDL は商用の NC-Verilog, VCS と比較してもかなり高速である．**MEM MAP + PARA** の論理シミュレーション実行時間は NC-Verilog の 32.2 倍，VCS の 11.3 倍高速である．

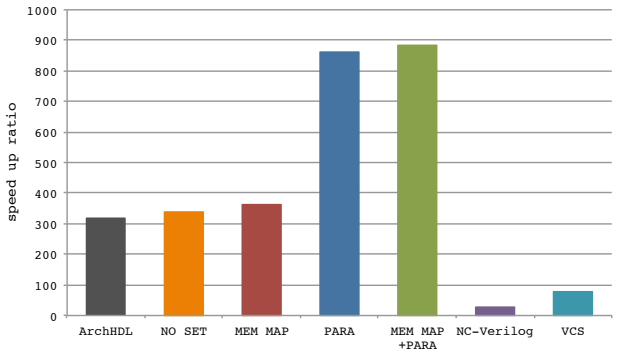

図 13: 512 個の XORSHIFT による乱数生成器の実行時間を Icarus Verilog と比較した速度向上比

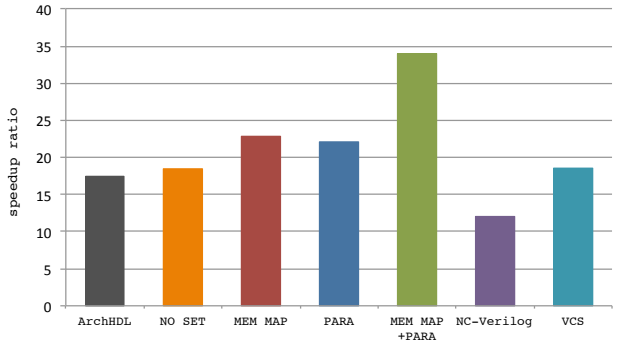

図 14: ステンシル計算回路の Icarus Verilog と比較した実行時間の速度向上比

また今回提案している高速化手法はオリジナルの ArchHDL に比べていずれも効果が出ている．**MEM MAP + PARA** の論理シミュレーション実行時間はオリジナルの ArchHDL の 2.78 倍高速である．

4.2 ステンシル計算回路による評価

図 14 はステンシル計算回路での実行結果である．縦軸は Icarus Verilog と比較したそれぞれの速度向上比である．

オリジナルの ArchHDL は商用の NC-Verilog より高速であったが，同じく商用の VCS はオリジナルの ArchHDL と **NO SET** より高速である．しかし逐次実行での高速化手法と並列化を共に適用した **MEM MAP + PARA** の論理シミュレーション実行時間は VCS の 1.83 倍高速である．

ステンシル計算回路の場合は Update() は 325,469,175 回呼ばれているのに対して，reg の値に更新がないのは 5,145,760 回である．つまり更新がないのは Update() メソッド呼び出し全体の 1.58% 程度に過ぎない．それにより条件分岐を無くす **NO SET** の論理シミュレーションはオリジナルの ArchHDL より高速である．また Update() のメソッド呼び出しを減らし，かつメモリ配置を工夫している **MEM MAP** の論理シミュレーション実行時間はオリジ

ナルの ArchHDL の 1.31 倍高速である．

また Module が 133 個，reg が 991 個存在する回路なので並列化の効果も大きい．逐次実行での高速化手法と並列化を共に適用した **MEM MAP + PARA** の論理シミュレーション実行時間はオリジナルの ArchHDL の 1.95 倍高速である．

5. まとめ

ハードウェアの RTL モデリングのための新しい言語として提案している ArchHDL の高速化手法を提案し，実装し，評価した．ArchHDL ではハードウェアのレジスタを変数，ワイヤを関数として扱うことで，C++ で RTL モデリングを実現する．

高速化手法として (1) データ変更の有無による条件分岐の除去，(2) 値を配列として格納しメモリ配置を工夫，(3) 並列化手法を提案した．

提案手法を実装し，ArchHDL を Icarus Verilog と商用ツールである VCS, NC-Verilog の実行時間と比較した．マイクロベンチマークである 4096 個のカウンタ回路を用いた評価では VCS より 56.7 倍高速であった．現実的なハードウェアであるステンシル計算回路を用いた評価では VCS より 1.83 倍高速であった．我々が知る限り最速な Verilog シミュレータである VCS よりも ArchHDL が高速にハードウェアシミュレーションが行えることを明らかにした．

謝辞

ArchHDL の開発に多大な貢献をしていただいた佐野伸太郎さんに感謝します．

参考文献

[1] 佐藤真平，吉瀬謙二：C++をベースとする新しいハードウェア記述の検討，情報処理学会研究報告. 計算機アーキテクチャ研究会報告 (ARC-205)，pp. 1–7 (2013).
[2] : Icarus Verilog. http://iverilog.icarus.com.
[3] : Synopsys VCS. http://www.synopsys.com/VCS.
[4] : OpenMP. http://openmp.org.
[5] 小林諒平，高前田（山崎）伸也，吉瀬謙二：多数の小容量 FPGA を用いたスケーラブルなステンシル計算機の開発，先進的計算基盤システムシンポジウム論文集，pp. 179–187 (2013).