計算機アーキテクチャ　103－19
設　計　自　動　化　69－19
（1993. 12. 17）

# アーキテクチャ評価用コンパイラの自動生成

冨山 宏之†　赤星 博輝‡　安浦 寛人‡

†九州大学 工学部 情報工学科

‡九州大学 大学院総合理工学研究科 情報システム学専攻
〒816 春日市春日公園 6–1

*E-mail: {tomiyama, akaboshi, yasuura}@is.kyushu-u.ac.jp*

あらまし

ハードウェア記述言語によるアーキテクチャ記述をもとにコンパイラの自動生成を行うことは，アーキテクチャの性能評価に要する設計者の労力を軽減するだけでなく，ソフトウェア開発環境を提供する．様々な命令セットを持つアーキテクチャに対応したコンパイラを自動生成するため Tree Rewriting(木変換) という手法を採用し，コンパイラ・ジェネレータのプロトタイピングを行った．プロトタイプにより生成されたコンパイラはレジスタ割り当て法に改善の余地はあるが，ターゲットプロセッサの命令セットに応じてほぼ最適なマシン命令を選択することができた．

和文キーワード：コンパイラ生成，　計算機アーキテクチャ設計，　性能評価，　木変換

# Compiler Generation for Computer Architecture Evaluation

Hiroyuki TOMIYAMA†　Hiroki AKABOSHI‡　and　Hiroto YASUURA‡

†Department of Computer Science and Communication Engineering
Kyushu University

‡Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
6–1 Kasuga-koen, Kasuga-shi, Fukuoka 816 Japan

*E-mail: {tomiyama, akaboshi, yasuura}@is.kyushu-u.ac.jp*

**Abstract**

Compiler generation from a computer architecture description in an HDL(Hardware Description Language) makes a designer evaluate his architecture easily. It also provides a software development environment. We are implementing a prototype of a compiler generator using the Tree Rewriting algorithm. The compiler generator produces a compiler for a designed architecture with various kinds of instructions. Experimental results show that compilers generated by the compiler generator generate near-optimal object code while there is room to improve the register allocation method.

英文 **key words**: Compiler generation,　Computer architecture design,　Performance evaluation,　Tree Rewriting

# 1 はじめに

高性能な計算機アーキテクチャの設計を行うには，設計の早期において性能評価を行うことが不可欠である．性能評価を行う目的は，ターゲットプロセッサが設計者の意図通りの性能を発揮するか調べることである．アーキテクチャの性能評価の項目として我々は，1) ハードウェア単体の性能評価と，2) システム (ハードウェア上でソフトウェアを実行した場合) の性能評価を考えている [1]．我々が提案している計算機アーキテクチャ設計支援システム COACH (COmputer Architecture designer workbenCH) は，コンパイラの自動生成を行うことによりシステムの性能評価に対する支援を行う (図 1参照)．

現在我々は HDL (Hardware Description Language) によるアーキテクチャ記述からコンパイラの自動生成を行う研究を進めている．コンパイラの自動生成はシステムの性能評価に対する支援を行うだけでなく，チップの実現と同時にソフトウェア開発環境をユーザに提供することで応用プログラムの開発を支援する．

HDL によるアーキテクチャ記述からコンパイラを自動生成するためには，アーキテクチャ記述からコンパイラ生成に必要な情報を抽出する**情報抽出部**と，情報抽出部が抽出した情報をもとにコンパイラの自動生成を行う**コンパイラ・ジェネレータ**の 2 つのツールが必要である．

本論文ではコンパイラ・ジェネレータのプロトタイピングを行い，コンパイラ・ジェネレータの実現の条件，実現法，ならびにコンパイラの生成能力について評価と考察を行う．2 章で関連研究について述べた後，3 章でコンパイラ・ジェネレータの全体像，4 章で作成中のプロトタイプについて説明する．5 章でアーキテクチャの性能評価にプロトタイプを使用した例を示し，6 章でプロトタイプの評価，7 章で考察を行う．

# 2 関連研究

近年，アーキテクチャ設計支援の立場からいくつかのコンパイラの自動生成の研究が行われている．1 つのアプローチとして GCC (GNU CC)[7] などの既存のリターゲッタブル・コンパイラを使用する方法がある．赤星ら [2] は仮想的な命令セットを用意し，GCC の中間言語 RTL と仮想的な命令セット，仮想的な命令セットとターゲットプロセッサの命令セットとの対応をとることによってマシンコードの生成を行っている．博多ら [3] は C コンパイラの生成を容易にするため，RTL をベースにプロセッサの命令セットを決定している．しかしこれらの手法では，様々な命令セットを持つプロセッサに柔軟に対応したコンパイラを自動生成できない．

Marwedel[4] はリターゲッタブル・マッパの研究を行っている．Marwedel の手法は，コードを生成するためのコンパイラを生成するのではなく，ターゲットの記述をもとにアルゴリズムをターゲットの構造上にマッピングすることによって直接バイナリコードの生成を行うものである．

一方，言語処理の分野でも 1970 年代から盛んにリターゲッタブル・コンパイラ，あるいはコンパイラ生成の研究が行われてきた．代表的な手法として LR 構文解析を利用してマシン命令の選択を行う手法 [5] や，ツリーの書き換えによりコード生成を行う手法がある．ツリーの書き換えによるコード生成法は，A.V.Aho らが Tree Rewriting (木変換)[6] という手法として体系的にまとめている．Tree Rewriting はツリーを書き換えるための規則 Tree-Rewriting Rule を用意することで，ターゲットプロセッサの命令セットに柔軟に対応することが可能である[1]．しかし，これらの研究はコンパイラの作成を容易に行うことを目的にしており，コンパイラの作成を行うのは人間である．

我々の目標は，コンパイラの自動生成を行うコンパイラ・ジェネレータを実現することである．Tree Rewriting を採用することにより，様々な命令セットを持つアーキテクチャに対応可能なコンパイラ・ジェネレータの実現を図る．

# 3 コンパイラ・ジェネレータ

## 3.1 ターゲット・プロセッサ

我々がターゲットにしているのは，既存のプロセッサに対して命令互換性を持たない汎用計算機向けプロセッサ，および組込みなどの専用計算機向けプロセッサである．

[1] Tree-Rewriting Rule の集合を Tree-Translation Scheme (木翻訳スキーム) という．

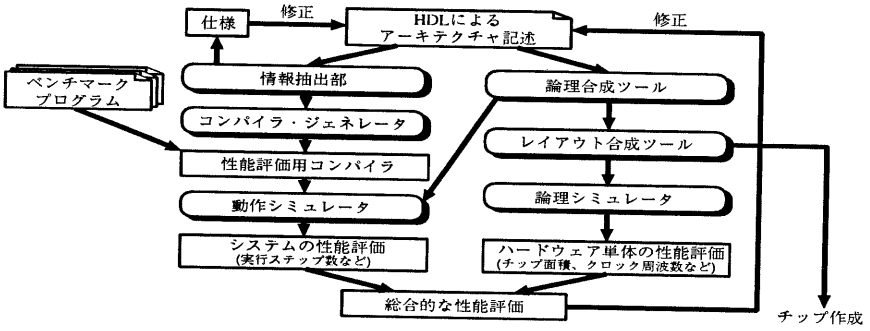


図 1: COACH

実現への第一段階としてターゲットプロセッサを逐次実行型プロセッサ，パイプラインプロセッサ，ならびにスーパースカラプロセッサとする．

### 3.2 構成

コンパイラ・ジェネレータは，

- 中間コード生成ルーチン生成部
- アセンブラコード生成ルーチン生成部
- コード・スケジューラ生成部
- アセンブラ生成部

から構成され，それぞれ

- 中間コード生成ルーチン
- アセンブラコード生成ルーチン
- コード・スケジューラ
- アセンブラ

を生成する (図 2参照).

### 3.3 要件

コンパイラ・ジェネレータは次の要件を満たさなければならない．

1. 短時間かつ容易にコンパイラを生成すること
2. ターゲットプロセッサに応じて最適なコードを生成するコンパイラを生成すること

中間言語がターゲットプロセッサに対して独立ならば，要件 2. は以下のように詳細化される．

- 以下の要件を満たすアセンブラコード生成ルーチンを生成すること
  - 命令セットに応じて最適なマシン命令を選択すること
  - レジスタ構成に応じて最適なレジスタ割り当てを行うこと
- 最適なコード・スケジューリングを行うコード・スケジューラ生成ルーチンを生成すること

## 4 プロトタイピング

### 4.1 方針

以下の手順に従って，コンパイラ・ジェネレータのプロトタイピングを行う．

1. コンパイラの機能をモジュール化
2. マシン独立なモジュールは C 言語の関数として実現
3. マシン依存のモジュールは次の手順で実現
   (a) マシン依存のモジュールをマシン独立なアルゴリズムとマシン依存の情報とに分割
   (b) (a) のマシン依存の情報をパラメータ化
   (c) (a) のアルゴリズムを，(b) の情報を入力とする (またはインクルードする)C 言語の関数として実現

コンパイラは上記の手順によって実現された関数をコンパイルすることにより生成される．

生成されたコンパイラがターゲットの命令セットに応じてマシンコードに変換できるかを検証するため，最初にアセンブラコード生成ルーチン生成部のプロトタイピングを行う．

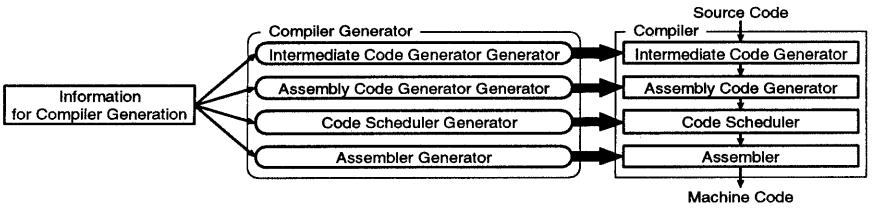


図 2: コンパイラ・ジェネレータとコンパイラ

## 4.2 中間言語

コンパイラ・ジェネレータのプロトタイピングを行う前に，アセンブラコード生成ルーチンの入力となる中間言語の設計を行った．設計した中間言語はソース言語とターゲットプロセッサの両方に対して独立である．基本的に 3 オペランド形式で，80 種類の原始的な命令を持つ．関数呼び出し，割り込みおよび浮動小数点演算についてはまだ完全にはサポートしていない．

## 4.3 Tree Rewriting

Tree Rewriting とは Tree-Rewriting Rule に従って行う，ツリーのパターンマッチングである．Tree-Rewriting Rule は Rewriting rule, Cost, Instruction から構成される (図 3参照)．Rewriting rule は対象のツリーを書き換えるための規則，Cost は Rewriting rule を適用するときのコスト，Instruction は Rewriting rule を適用したときに出力するマシン命令である．

一つのツリーに対して複数の Rewriting rule が適用可能な場合は，Cost の総和が最小となるように Rewriting rule を選択することにより，質の良いマシンコードを生成することができる．また，Tree-Rewriting Rule を変更することにより，様々な命令セットに対応することが可能である．

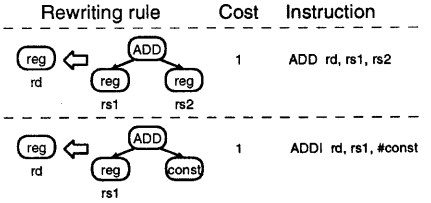


図 3: Tree-Rewriting Rule の例

## 4.4 実現法

アセンブラコード生成ルーチンは主に次の 3 つの機能を持つ (4参照).

1. 中間コードからのツリーの作成
2. Tree Rewriting によるマシン命令の選択
3. レジスタ割り当て

ツリーの作成はマシン独立であるため，C 言語による関数として実現した．Tree Rewriting によるマシン命令の選択とレジスタ割り当てはマシン依存である．マシン命令の選択を行うモジュールにおけるマシン依存の情報を，

1. 命令セット
2. 命令の実行に伴うコスト

とし，レジスタ割り当てモジュールにおけるマシン依存の情報を，

3. 使用可能な汎用レジスタ数とレジスタ番号
4. 1 ワードのサイズ

とした．

Tree Rewriting を行うプログラムの作成を支援する Twig というツールが既に実現されている．我々はコンパイラ・ジェネレータの一部に Twig を使用することにより，コンパイラの生成を容易にする．Tree-Rewriting Rule を記述した Twig 仕様記述を Twig に与えると，Twig は Tree Rewriting を行う C プログラムを生成する．

Tree-Rewriting Rule として記述された情報 1, 2 が Twig 仕様記述への入力である．一方，レジスタ割り当てアルゴリズムを C 言語の関数で実現した．このレジスタ割り当て関数は情報 3, 4 をインクルードすることにより完成される．

また，Makefile を作成することにより，これらの関数からコンパイラの生成を容易に行う環境を作成した．

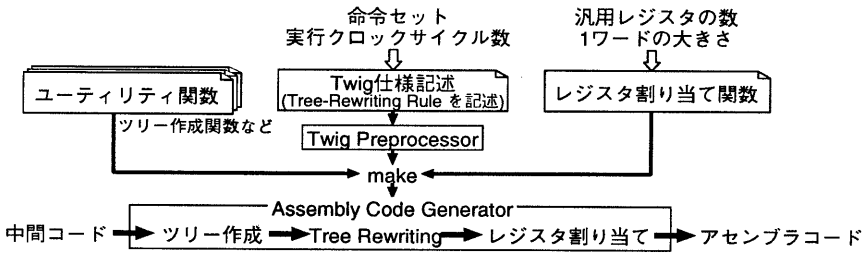


図 4: アセンブラコード生成ルーチン生成部

# 5 アーキテクチャ設計例

ASIC Design System(Tsutsuji)[8][9] を使用して異なった命令セットを持つ 2 つのプロセッサ A, B の設計を行った．本章ではプロトタイプにより生成されたコンパイラを用いて性能評価を行うことによりコンパイラ・ジェネレータの有効性を示す．プロトタイプの評価は次章で行う．

## 5.1 プロセッサの仕様

プロセッサ A, B 共に，逐次実行方式，汎用レジスタ方式，3 オペランド方式であり，命令長は 32 ビットである．32 ビットの汎用レジスタ数を 8 本持ち (うち 1 本は常に 0)，1 本をフレームポインタとして使用する．プロセッサ A, B の命令セットを表 1に示す．分岐命令の実行には 2 クロックサイクル数を要し，分岐命令以外は 1 クロックサイクル数を要する．

表 1: 命令セット

| | プロセッサ A | プロセッサ B |
|---|---|---|
| データ転送 | LD, ST | LD, ST |
| 算術演算 | ADD, ADDI | ADD, ADDI |
| 比較 | SGE | |
| 条件分岐 | BNEZ | BGE |
| 無条件分岐 | J | J |

## 5.2 コンパイラ生成

今回，回路図入力により設計を行ったこと，情報抽出部がまだ実現していないことから，コンパイラ生成に必要な情報は人間の手によりコンパイラ・ジェネレータに与えた．コンパイラ・ジェネレータに与えた情報は 4.4節で述べた通りである．コストとして各命令の実行クロックサイクル数を与えた．また，アセンブラは人間が作成した．

## 5.3 アーキテクチャの性能評価

### 5.3.1 ハードウェア単体の性能評価

論理合成を行った結果得られたプロセッサ A, B の動作周波数 (標準値) およびゲート数を表 2に示す．これらの値は富士通の AU シリーズにマッピングした場合である．ゲート遅延時間は 1 ゲートあたり 0.8 [ns] である．

表 2: ハードウェア単体の性能評価

| | プロセッサ A | プロセッサ B |
|---|---|---|
| 動作周波数 | 15.65 MHz | 15.14 MHz |
| ゲート数 | 7,335 | 7,269 |

### 5.3.2 システムの性能評価

今回，ベンチマークプログラムとして次の 2 種類のループを使用した．

```
loop1:
    sum = 0;     n = 5;
    for ( i=0 ; i<n ; i++ ) {
        sum = sum + i;
    }

loop2:
    for ( k=1 ; k<20 ; k++ ) {
        x[k] = y[k] + x[k-1];
    }
```

中間コードからマシンコードに変換する部分しかコンパイラの生成を行わなかったため，loop1, 2 を中間コードにハンドコンパイルした後，生成されたコンパイラを用いてマシンコードに変換した．動作シミュレーションを行うことにより求めた実行ステップ数を表 3に示す．

表 3: 実行ステップ数

| | プロセッサ A | プロセッサ B |
|---|---|---|
| loop1 | 76 steps | 70 steps |
| loop2 | 349 steps | 329 steps |

#### 5.3.3 総合的な性能評価

ハードウェア単体の性能評価とシステムの性能評価の結果から，プロセッサ A, B 上で loop1, 2 を実行したときの実行時間を表 4に求めた．プロセッサ B はプロセッサ A と比較して，loop1, 2 においてそれぞれ 5.0%, 2.5% 高速である．一方，ゲート数もプロセッサ B の方が少ないため，今回のベンチマークプログラムから判断するとプロセッサ B の方が高性能であるといえる．

表 4: 実行時間

| | プロセッサ A | プロセッサ B |
|---|---|---|
| loop1 | 4.855 $\mu$s | 4.622 $\mu$s |
| loop2 | 22.29 $\mu$s | 21.73 $\mu$s |

### 5.4 コンパイラ・ジェネレータの有効性

中間言語で記述されたベンチマーク・プログラムをプロトタイプが自動生成したコンパイラでマシンコードに変換することにより，容易にシミュレーションを行うことができた．また，ゲート数や動作周波数などのハードウェアの性能の概算が求まった時点で，ハードウェアとソフトウェアの両方を考慮した性能評価を行うことができた．

コンパイラ・ジェネレータはアーキテクチャ設計の早期において信頼性の高い性能評価を行うことを可能にする．

## 6 評価

本章ではコンパイラ・ジェネレータのプロトタイプの評価を行う．評価は 1) プロトタイプにより生成されたコンパイラのコード生成能力と，2) コンパイラ生成の容易さの 2 項目について行う．

### 6.1 生成されたコンパイラの能力

#### 6.1.1 人間との比較

プログラムをハンドコンパイルした場合との実行ステップ数の比較を行った．loop1, 2 をハンドコンパイルしてシミュレーションを行った結果得られた実行ステップ数を表 5に示す．ただし，アセンブラコード生成ルーチンだけの評価を行うため，ハンドコンパイルする際には基本ブロックを越えるコード移動は行っていない．

表 5: 実行ステップ数の比較

| | 自動生成 | ハンドコンパイル |
|---|---|---|
| loop1 | 76 steps | 41 steps |
| loop2 | 349 steps | 252 steps |

(プロセッサ A を使用した場合)

プログラムをハンドコンパイルした場合は，自動生成されたコンパイラを使用した場合と比較して，loop1, 2 で実行ステップ数がそれぞれ 46%, 27%削減された．生成されたアセンブラコードを見ると，ハンドコンパイルした場合はレジスタをほぼ最適に割り当てており，ロード/ストア命令が大幅に削減されたため両者に差が生じたことがわかった．

#### 6.1.2 既存のコンパイラとの比較

GCC(バージョン 1.36) との比較を行った．ターゲットプロセッサには DLX[10] を使用し，DLX のマシン命令のうち 27 命令をサポートした．ベンチマークプログラムには要素数 100 のバブルソートプログラムを使用した．プロトタイプはコンパイラの一部しか生成しないこと，C 言語のフルセットをコンパイルできないこと，使用している中間言語が異なっていることから，以下の手順で比較を行った．ただし，GCC を使用する際は最適化を行わなかった．

1. ソースプログラムを GCC の中間言語である RTL に変換する．
2. RTL による中間プログラムから，実際にソーティングを行う部分のコードを抽出し，プロトタイプで使用している中間コードに変換する．
3. 2. で得られた中間コードを，コンパイラ・ジェネレータが生成したコンパイラを用いて DLX のアセンブラコードに変換する．
4. GCC を用いてソースプログラムを DLX のアセンブラコードに変換し，実際にソーティングを行う部分を 3. で得られたアセンブラコードと入れ換える．

5. 4. で得られたアセンブラコードと，ソースプログラムをGCCによりコンパイルしたアセンブラコードとを用いてシミュレーションを行い，実行ステップの比較を行う．

シミュレーション結果を表6に示す．自動生成されたコンパイラによって生成されたコードは，GCCが生成したコードに比べてステップ数が30%少ない(ソーティング部)．両者のステップ数の差は，レジスタ割り当て法の違いと使用した中間コードの違いに起因していた．中間コードからマシン命令を選択する能力に関して，プロトタイプが生成したコンパイラは人間が作成したGCCと対等の能力を持っていると言える．

表6: 実行ステップ数の比較

| | 自動生成 | GCC |
|---|---|---|
| プログラム全体 | 181,966 steps | 232,434 steps |
| ソーティング部 | 167,448 steps | 217,916 steps |

## 6.2 コンパイラ生成の容易さ

コンパイラ生成の容易さについて評価を行うため，コンパイラ生成に必要な情報量(Tree-Rewriting Ruleの数)とコンパイラの生成時間の2項目について評価を行った．

表7にプロセッサA,Bおよびサポートした DLXのマシン命令の数，コンパイラ・ジェネレータに与えたTree-Rewriting Ruleの数[2]，Twigにより生成されたCプログラムの行数，ならびにアセンブラ生成ルーチンの生成時間を示す．表の生成時間はTwig使用記述をTwigプリプロセッサによってC言語のプログラムに変換し，それをレジスタ割り当て関数，およびその他のユーティリティ関数群とともにCコンパイラでクロスコンパイルし，リンクするのに要した時間である．生成時間はSPARC Station10 (SunOS 4.1.3) 上で5回測定を行った平均値である．

質の良いコードを生成するために必要なTree-Rewriting Ruleの数は命令セットとレジスタ構成により決定される．例においてマシン命令の数に比べてTree-Rewriting Ruleの数が多いのは，ターゲットプロセッサがゼロレジスタを持っていたため，定数0をレジスタにロードして演算を行う場合にはゼロレジスタを使用するようにTree-Rewriting Ruleの記述を行ったためである．Tree-Rewriting Ruleの記述を簡略化するため，類似した命令に対してTree-Rewriting Ruleの共有化を行うなどの処理が必要である．

プロセッサBはプロセッサA, DLXに対してそれぞれ4%, 10%，プロセッサAはDLXに対して6%生成時間が短い．アーキテクチャが複雑になる従ってTree-Rewriting Ruleの数が増加し，生成時間が増大する．しかし，コンパイラの生成時間は論理/レイアウト合成に要する時間と比べると非常に短いため，アーキテクチャの設計を行う上で問題にならないと言える．

[2]他にターゲットプロセッサに独立なTree-Rewriting Ruleが4つ存在する．

# 7 考察

コンパイラ・ジェネレータにより生成されるコンパイラの質は与えられたTree-Rewriting Ruleの質とレジスタ割り当て能力に依存する．プロトタイプではTree-Rewriting Ruleを手で与えることにより，中間コードに対して最適なマシン命令を選択することができた．一方，グラフ彩色法などの強力なレジスタ割り当て法を採用していないため，ターゲットプロセッサのレジスタを有効に使用できず，ハンドコンパイルした場合に比べるとプログラムの実行ステップ数が多い．強力なレジスタ割り当てを行うことにより改善可能であるがTree Rewritingの性質上，次のような問題点がある．

- ツリー表現からアセンブラコードの生成を行うため，中間コードで並んでいる順番で，対応するマシン命令が出力されるとは限らない．このことは，中間言語レベルにおいて変数などの値の生存期間を正確に知ることができないことを意味する．
- ターゲット・プロセッサがLoad&Add命令のような複合命令を持つとき，spill処理により挿入されたロード/ストア命令は，前後の命令と複合させることによって，より良いコードに変換できる可能性がある．

人間がコンパイラを作成する場合はこれらの問題を考慮した上で作成を行う．しかし，コンパイラの自動生成を行う場合にはこれらの問題を解決，自動化する必要がある．

表 7: アセンブラコード生成ルーチンの生成時間

| | プロセッサ A | プロセッサ B | DLX |
|---|---|---|---|
| マシン命令の数 | 7 命令 | 6 命令 | 27 命令 |
| 与えた Tree-Rewriting Rule の数 | 17 rules | 17 rules | 42 rules |
| Twig が生成した C プログラムの行数 | 1007 lines | 1010 lines | 1319 lines |
| 生成時間 | 5.8 sec | 5.6 sec | 6.2 sec |

## 8 おわりに

ターゲットプロセッサに制限を加えることにより，コンパイラ・ジェネレータのプロトタイピングを行った．Tree Rewriting を採用することによりほぼ最適なマシン命令の選択を行うことができたが，レジスタ割り当ての能力に改善の余地がある．

今回，情報抽出部がまだ実現していないため，コンパイラ生成用の情報は人間がコンパイラ・ジェネレータに与えた．与えた情報は 1) 命令セット，2) 命令のコスト，3) 汎用レジスタ数，4) 1 ワードのサイズである．HDL によるアーキテクチャ記述からコンパイラを自動生成するためには，これらの情報は情報抽出部が抽出しなければならない．その際に問題となるのは Tree-Rewriting Rule (命令セットとコスト) の抽出であろう．高性能なコンパイラを生成するためには Tree-Rewriting Rule の詳細な記述を必要とするからである．

今後，コンパイラ・ジェネレータ全体のインプリメントを目指すと同時に，コンパイラ生成に必要な Tree-Rewriting Rule がアーキテクチャによってどのように変化するか解析することにより，ソフトウェアとハードウェアの境界を探りたい．

## 謝辞

Tsutsuji の提供など御協力頂く YHP の皆様に感謝致します．Twig を提供していただいた AT&T Bell 研究所の皆様に感謝致します．日頃御討論頂く九州大学大学院総合理工学研究科の村上和彰講師，ならびに安浦研究室の諸氏に感謝致します．

本研究は一部，京都高度技術研究所の新産学交流事業 EAGL の支援による．

## 参考文献

[1] 赤星博輝, 安浦寛人, "計算機アーキテクチャ設計支援に関する考察," 信学技報, VLD93-57, pp.47-54, 1993 年 10 月.

[2] 赤星博輝, 安浦寛人, "アーキテクチャ評価用ワークベンチ — コンパイラの自動生成 —," 信学技報, VLD92-86, pp.9-16, 1993 年 1 月.

[3] 博多哲也, 佐藤淳, A.Y. Alomary, 今井正治, 引地信之, "ASIC CPU 向きソフトウェア開発環境生成系の実現," 信学技報, VLD92-25, pp.43-48, 1992 年 5 月.

[4] Peter Marwedel, "Tree-Based Mapping of Algorithms to Predefined Structures," *IEEE/ACM International Conference on Computer-Aided Design*, pp.586-593, Nov 1993.

[5] R.S. Glanville and S.L. Graham, "A new method for compiler code generation," *Fifth ACM Symposium on Principles of Programming Languages*, pp.231-240, 1978.

[6] A.V. Aho, M. Ganapathi and S.W.K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming," *ACM Transactions on Programming Languages and Systems*, Vol.11, No.4, pp.491-516, Oct 1989.

[7] Richard M. Stallman, "Using and Porting GNU CC for version 1.39," Free Software Foundation, Inc., Jan 1991.

[8] "ASIC Design System User's Manual," 株式会社 YHP システム技術研究所.

[9] "ASIC Design System Reference Manual," 株式会社 YHP システム技術研究所.

[10] J.L. Hennessy and D.A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.