Fuce言語HALの設計と実装

雨宮, 聡史
九州大学大学院システム情報科学府知能システム学専攻 : 博士後期課程

平号, 亮
九州大学大学院システム情報科学府知能システム学専攻 : 修士課程

越村, 三幸
九州大学大学院システム情報科学研究院知能システム学部門

藤田, 博
九州大学大学院システム情報科学研究院知能システム学部門

他

https://doi.org/10.15017/1516863

出版情報：九州大学大学院システム情報科学紀要. 11 (2), pp.103-108, 2006-09-26. 九州大学大学院システム情報科学研究院
バージョン：
権利関係：

九州大学大学院
システム情報科学紀要
第11巻　第2号　平成18年9月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.11, No.2, September 2006

Fuce言語HALの設計と実装

雨宮聡史*・平兮亮**・越村三幸***・藤田博***・長谷川隆三***

Design and Implementation of the Language HAL for Fuce Architecture

Satoshi AMAMIYA, Ryo HIRANA, Miyuki KOSHIMURA, Hiroshi FUJITA
and Ryuzo HASEGAWA

(Received June 16, 2006)

Abstract: In our previous works, we inroduced a higer intermidiate language called IML for the Fuce processor, and discussed special programming methods for stream processing on Fuce. However, without realization of the language HAL and its compiler, it is difficult to utilize the power of these tools. In this paper, we discuss the implementation techniques for constructing the HAL compiler including powerful optimization functionalities.

Keywords: Fuce architecture, Fine-grain multi-threading, Language design, Compiler

1. まえがき

Fuceプロセッサ[1),2)]はデータフロー計算モデルを発展させた継続[5),6)]という概念を直接実現したチップマルチプロセッサである．Fuceでは，'走りきり'スレッドが並列実行の単位であり，スレッドスタックは存在せず，プリエンプションという概念すら成り立たない．これは，現在広く利用されているマルチスレッドの実行モデルとは根幹から異なっており，既存の言語処理系では，Fuce向けのプログラム実行コードを生成することが難しいということを意味する．そこで，我々はFuce用の高級言語IML[3)]を開発し，また，Fuceの実行モデルを徹底的に活かしたストリーム処理プログラミング法を発案し[4)]，IML言語に導入した．このIMLを用いることで，排他制御等を含めたFuceオペレーティングシステムの記述が現実のものと成りつつある．しかし，Fuce言語系の最下位に位置するHAL言語のための質の良い最適化コンパイラが存在しなかったために上位言語IMLを十分活用することが難しかった．

HALコンパイラを作成する上で，一から最適化機能を含めて実装するのは困難を伴うため，我々は，既存の最適化C言語コンパイラを援用する方法を採用した．既存の処理系を流用する上で，Fuceの実行モデルからC言語の実行モデルへ変換が鍵となる．本稿では，HAL言語のための最適化コンパイラの実装法ならびに，Fuce向けのプリローディング最適化法について詳説する．

平成18年6月16日受付

* 知能システム学専攻博士後期課程
** 知能システム学専攻修士課程
*** 知能システム学部門

2. HAL言語

HAL言語はFuce用のもっとも低位な言語として設計されている．C言語の文法を流用し，ポインタを含めて変数，式や制御構造を備えており，かつ，Fuceの実行モデルを直接記述できる言語仕様となっている．また，**Table 1**にあるようなスレッド遷移等のFuceプロセッサ固有の命令をHAL処理系の組み込み関数として用意し，HALから直接プロセッサを操作できるようになっている．**Fig. 1**は再帰による階乗計算関数をHALで記述したものである．この**Fig. 1**のコードを用いながらHALの構文要素の概要を述べる．

Table 1 Instruction set for Fuce.

Arithmetic & Branch	
compliant with MIPS	

Thread handling	description
cont rs	thread continuation
delda rs	release data area (macro)
delins rs	release ACM instance
end	end of thread
newda rd, rs	acquire data area (macro)
newins rd, rs, rt	acquire ACM instance
plend	end of pre-loading
setacm rs, rt, imm	thread registration

関数定義

Fuceにおける関数はスレッドの集合とその実行環境として表される．これを表現するには，オブジェクト指向言

```
int fact(int n) {
  darea {
    int n;
    int m;
    int return_thid;
    int *return_val;
  }
  thread fact <1> {
    if (base->n > 0) {
      fact_darea *f1_da = newda(fact);
      int f1 = newins(fact, f1_da);
      f1_da->n = base->n - 1;
      f1_da->ret_thid = out_fact;
      f1_da->ret_val = &(base->m);
      cont(f1);
    } else {
      int ret_id = base->ret_thid;
      *(base->ret_val) = 1;
      cont(ret_id);
      delins(id);
      delda(base);
      end;
    }
  }
  thread out_fact <1> {
    *(base->ret_val) =
        base->n * base->m;
    cont(base->ret_thid);
    delins(id);
    delda(base);
    end;
  }
}
```

Fig. 1 An example of a function difinition in HAL.

語におけるクラス定義や関数型言語におけるクロージャを意識した構文が適当である．関数宣言部はC言語と同様であるが，関数本体定義部は唯一のデータエリア構造体定義とスレッド定義の列を記述する．例えば，**Fig. 1**の関数定義部ではdareaで始まるデータエリアと2つのスレッドfactとout_factを定義している．

データエリア定義

データエリアとは関数内局所変数および引数のためのメモリ領域である．このような領域は，逐次実行モデルの言語では，スタックフレームとして自動的に割り当てることができるが，Fuceの関数は並列起動が可能なので，一般的にはスタックを用いることは難しい．また，関数内の複数のスレッドから共通にアクセスできるような構造を成す必要がある．そのため，C言語の構造体風にdarea{...}と定義する．関数が戻り値を持つ場合は，戻り先のスレッドIDと戻り値へのポインタをデータエリアに用意する必要がある（**Fig. 1**のint ret_thidとint *ret_valに相当）．なお，このデータエリアのタイプ名は例えばfact_dareaというように“関数名 アンダースコア darea”となり，データエリアを確保する場合にはこのタイプ名を用いる．

スレッド定義

スレッドの定義はthread宣言子に続きスレッド名およびfan-in指定 < >を記述する．例えば，thread fact <1>{...}と記述すれば，fan-inが1であるfactという名のスレッドを定義したことになる．関数と同名のスレッドは関数の入り口スレッドである．入り口スレッドとは関数が起動されたときに最初に発火されるスレッドである．スレッド本体はC言語文法通りに記述できる．すなわち，変数定義，制御構造は自由に使用可能である．ただし，組み込みでない外部関数呼び出しはC言語のように直接は記述できない．また，スレッド本体内で使用可能な2つの組み込み変数idとbaseを用意している．idは自分自身のスレッドIDを表す変更不可能な変数である．baseは自分の属する関数のデータエリアへのポインタである．スレッド内では，データエリアへのアクセスは変数base経由で行う．

関数呼び出し

HALは高級アセンブラという位置づけの低位言語なので，関数呼び出しにはいくらかの煩雑さが伴う．以下は**Fig. 1**から，関数呼び出しに関連する箇所を抜粋したものである．

```
1:  fact_darea *f1_da = newda(fact);
2:  int f1 = newins(fact, f1_da);
3:  f1_da->n = n - 1;
4:  f1_da->return_thid = out_fact;
5:  f1_da->return_val = &(base->m);
6:  cont(f1);
```

この部分は，新規に関数factを呼び出している部分である．まず，関数factのデータエリアをnewdaで確保し(1行目)，それを基にnewinsでACMページを確保し，関数内の全スレッドをそのACMページに登録する．変数f1は関数factの入り口スレッドIDが割り当てられる（2行目）．3行目は実引数n-1を設定し，4，5行目は戻りスレッドを戻り値ポインタを設定している．これで関数のセットアップが完了したので，最後に入り口スレッドに遷移する（6行目）．

3. HALコンパイラの実装

HAL言語は，その大部分をC言語文法を流用している．

そのため，いちから最適化HALコンパイラを実装するよりも，HALを完全なC言語へ変換した上で，既存の現代的なCコンパイラの最適化能力を活用するほうが，コスト面を考慮しても現実的であり，より質の良いコードを得ることができる．以下にHALコンパイラの実装法を2通り述べるが，その一方はCコンパイラ自体にも若干の修正を必要とするので，ソースコードが公開されているgcc (v.4.1系)[7]の利用を前提として議論する．

3.1 GCCの改変を伴う実装法

HALからC言語への変換は比較的単純なので，変換系を実装することで自動化することができる．以下に変換方法を述べる．

ステップ0

Fuceのスレッド制御命令は，HALでは組み込み関数として使うことができるが，これらをC言語上でも組み込み関数のように見せかけるには，Cプリプロセッサおよびコンパイラ(gcc)のインラインアセンブラ構文を使ってマクロを定義すればよい．例として，組み込み関数contに対応するgcc用マクロ定義を以下に示す．

```
#define cont(x)  \
   asm volatile ("cont\t%0" : : "r" (x))
```

contは返り値は無く，読み込み専用の引数が一つなので‘"r" (x)’としている．‘%0’はgccのレジスタ割り付けによって変数xに対応した適切なレジスタ名に変換される．また，アセンブラ命令がgccの最適化処理によって無用命令扱いされるのを防ぐためにvolatileは必須である．組み込み関数newinsに対するマクロは次のように少し複雑になる．

```
#define newins(x)                              \
  ({ int _result;                              \
    asm volatile ("newins\t%0, %1, r0"  \
      : "=r" (_result) : "r" (x));             \
    _result; })
```

この記述では，複文を単一式として扱えるようにするためのgccの拡張文法を使っている．この式の値は最終文‘_result;’の評価値となり，これが関数newinsの戻り値となる．更に，出力値の指定に‘"=r" (_result)’を与え，デスティネーションレジスタ%0と変数_resultを対応させている．gcc拡張機能に関する詳細説明はgccのマニュアル[7]を参照されたい．

その他全てのFuceの固有命令はgccの拡張文法を用いてマクロとして定義することが可能である．全マクロ定義を記述したヘッダファイルfuce.hを用意し，後述のステップで生成されるC言語ソースファイルで，常にこれをインクルードするようにしておけばよいため，このステップの実行は一度だけである．

ステップ1

HALで定義された全ての関数からデータエリア構造体を抜き出し，それぞれをC言語の構造体として定義し直す．例えば，**Fig. 1**の関数factのデータエリアは次のようになる．

```
darea {                    typedef struct
  int n;                     _func_darea {
  int m;            ⇒        int n;
  int ret_thid;              int m;
  int *ret_val;              int ret_thid;
}                            int *ret_val;
                           } *func_darea
```

この変換により，C言語でfunc_dareaタイプの変数を宣言すれば，HALの関数funcのデータエリアを指せるようになる．

ステップ2

HALにおける各関数内で定義されたスレッド全てを抜き出し，個々をC言語のvoid型関数として定義し直す．このとき，スレッド名をスレッドIDとして参照している箇所(例えば，スレッド名some_thに対してcont(some_th))は，スレッドの抽出変換過程でスレッド名に対して番号を付けておき，変換後にその数値で置き換える．変換過程において，スレッド定義の出現順にスレッドIDを割り振っておけば簡単にスレッド番号を管理できる．なお，関数の入り口スレッドに関してはそのIDは0と決まっている．ところで，このような単純な方法ではスレッドIDの衝突が心配されるが，Fuce上では，関数インスタンスのID（これはnewins命令によって与えられる）とスレッドIDの2つ組で実行時のスレッドを識別するので問題は生じない．

Fig. 1のスレッドfactの定義をCの関数に変換すると**Fig. 2**のようになる．1行目では，HALでの組み込み変数idとbaseをC言語からもそのまま利用できるようにするために，変換後のC言語関数の仮引数として宣言する．第二引数baseのタイプ名はステップ1で変換したデータエリアのタイプ名になっていることに注意．そして，関数先頭の文として，Fuceのプリローディング終了命令plendを追加している（2行目）．これの作用は後述する．

6, 7行目では，setacm命令を使って，5行目で確保したACMページにスレッドを登録している．この命令は元のHALでは記述する必要はない．C言語への変換過程で，直前のnewins命令の引数のタイプから，関数名が分かり，更にそこから，HALの関数定義を見ることで，登録すべ

```
1:  void fact(int id, fact_darea base) {
2:    plend;
3:    if (base->n > 0) {
4:      fact_darea f1_da
            = newda(fact_darea);
5:      int f1 = newins(f1_da);
6:      setacm(f1, fact, f1_da, 1);
7:      setacm(f1+1, out_fact, f1_da, 1);
8:      f1_da->n = base->n - 1;
9:      f1_da->ret_thid = id + 1;
10:     f1_da->ret_val = &(base->m);
11:     cont(f1);
12:   } else { ....
```

Fig. 2 C function as a thread definition.

きスレッド全てを検出できるので，自動的に適切な個数のsetacm 命令を挿入することができる．なお，命令plendとsetacm はステップ0で定義しているものとする．

以上により，HALのスレッド定義を文法的には正しいCの関数へ変換することが可能となる．このときのC関数は，必ずリーフ関数となっているので，C言語としてのコンパイル時に最適化しやすいということを付け加えておく．

HAL言語からC言語への変換は以上で完了であるが，Fuceの実行モデルは通常のC言語の実行モデルと異なっているので，正しいFuceのオブジェクトコードを生成するためにはCコンパイラ自体を改変しなければならない．

ステップ3

Cコンパイラにおいて関数コード生成のプロローグおよびエピローグ処理を変更する．具体的には，これらの処理は通常，関数のスタックフレームやレジスタの待避復帰を行うが，Fuceのスレッドにはスタックフレームは存在せず，レジスタ待避という考えすら存在しない．Fuceではレジスタ待避が必要な場合つまり空きレジスタ枯渇した場合は，そこでスレッドを分割するという方法をとる．よって，スタックの伸縮操作やレジスタ待避回復処理を行わないようにし，更に，レジスタの枯渇を検知した場合には，コンパイルを中断しエラーを投げ，ユーザに対して，HALの関数およびスレッドの再定義を促す．

また，コンパイラで使われるレジスタ利用規約もFuce向けに再定義する必要がある．現在，Fuceの命令セットはMIPS命令セットを拡張して定義されているが，レジスタ利用規約はMIPSのそれとは全く異なる．Fuceではレジスタ1番から4番までがスレッド実行用に予約されているだけで，その他のレジスタは自由である．特に，スレッド実行開始時にはレジスタ1番にはスレッドIDが，レジスタ3番にはデータエリアポインタがセットされることになっているので，ステップ2で生成したC言語関数の第一引数idをレジスタ1番に，第二引数baseをレジスタ3番に割り当てるようにする．

以上ステップ0から3を実現することで，Fuceのアセンブラソースコードを生成することができる．ステップ3のgcc改変作業は，基本的にはgcc内部処理の最終段であるコード生成部に若干手を加えるだけなので，たいしたコストをかけずに，これら全ステップを実現することが可能である．

3.2 別法：ポストプロセスによるコード生成

前節で述べた方法は，gccの利用を前提とし，gcc自体を改変することでアセンブラコードの生成を可能とした．この方法の利点は，コンパイラの持つコード最適化機能を最大限活用できることであるが，一方でgccのヴァージョンアップ等に追随するために，gcc内部構造にある程度精通しておかなければならない．また，ソースコードの利用が難しい高性能な商用コンパイラを用いることも不可能である．

そこで，本節では任意のCコンパイラの利用を想定した方法を述べる．この方法の大部分は前節と共通であるため差分のみを説明する†1．

ステップ2b

HALのスレッド定義を，**Fig. 3**のようにC関数に変換する．変換後のC関数は無引数とし，関数本体の先頭で変数idとbaseをレジスタ番号指定付きの変数として宣言する（2,3行目）．その直後（4行目）に，スレッド本体の開始を表すマーカth_beginをインラインアセンブラ構文として追加する．同様に，スレッド定義の終端にも終了マーカth_endを追加する（7行目）．

```
1:  void fact() {
2:    register int id asm("%1");
3:    register fact_darea base asm("%3");
4:    asm volatile
          ("th_begin fact, fact, 1");
5:    plend;
6:    if (base->n > 0) { ... } else { ... }
7:    asm volatile ("th_end");
8:  }
```

Fig. 3 Translated thread definition for post-processing.

ステップ3b

前ステップで得られたC言語ソースを既存のCコンパイラでコンパイルしてアセンブラソースを生成する．このソースには，当然ながら，関数の入り口，出口付近にスタッ

†1　本節の例はgccのインラインアセンブラ構文を用いているが，他のCコンパイラでもgccと互換または類似機能が用意されていることが多い．

ク操作のための余計な命令列が追加されてしまうが，前ステップで追加したマーカth_beginからth_endの範囲にある命令列がスレッド本体に当たるので，この部分だけを抜き出すための後処理を行えばよい．

ただし，このポストプロセスによる方法では，利用するCコンパイラのもつレジスタ利用規約に従った制約を受けるので，Fuceプロセッサ向けの最適なレジスタ割り当ては期待できない．

4. プリローディング最適化

Fuceプロセッサにはプリローディングユニットと呼ばれるロード命令に特化した機能ユニットが備わっており，算術演算を開始する前に，あらかじめメモリからレジスタへデータを転送（先読み）しておくことで，効率的なスレッド実行を可能としている．

しかし，プロセッサが自動的に実行中のスレッドを解析してデータの“先読み”を行うわけではないので，この機能を活用するためには，先読み命令列をスレッドコードに追加する必要がある．先読み命令列を作るには，スレッドコード中に散在しているロード命令をスレッドコードの先頭からplend命令の間に可能な限り移動させればよい．本節では，典型的なパターン例を用いて，スレッドコードからのロード命令の抽出法を述べる．

なお，本手法はHAL言語からC言語への変換過程で行うことを前提にして議論する．すなわち，ロード命令を直接抽出するのではなく，ロード命令を生成する演算式を対象にする．以下の例では，演算子“->”を中心に議論を進めるが，その他のメモリアクセスを引き起こす演算子に関しても方法は同じである．

【パターン1－単純代入文の移動】

次のようなコードを考える．

```
plend;                          int v = a->b;
   ⋮        (A)     ⟹          plend;
v = a->b;                          ⋮        (A)
   ⋮                               ⋮
```

(A)の部分で変数v，aおよびcは宣言されているだけで，値は定義されていないとする．以降，この条件を条件Aと呼ぶ．条件Aのとき，代入文の右辺に含まれる演算子が“->”のみの場合は，この文をplend命令前に移動する．

【パターン2－部分式の置換と移動】

演算子“->”が複合式の一部として使われている場合を考える．条件Aのとき，新規変数を導入して，演算子“->”の式をこの変数で置き換える．そして，plend命令前に，この変数の宣言および定義を加える．例えば，式a->bを新変数v1で置き換えると，

```
                                int v1 = a->b;
plend;                          plend;
   ⋮        (A)     ⟹             ⋮        (A)
v = a->b + c;                   v = v1 + c;
   ⋮                               ⋮
```

このようになる．ここで，仮に“v = a->b->c;”であった場合はパターン1が適用されることに注意．

【パターン3－条件分岐】

else節の無いif文は次のようにする．ただし，条件Aおよび条件A’が成立しているとする．

```
                                int v1 = a->b;
plend;                          plend;
   ⋮        (A)                    ⋮        (A)
if (condition) {        ⟹      if (condition) {
   ⋮        (A')                   ⋮        (A')
 v = a->b + c;                   v = v1 + c;
   ⋮                               ⋮
} ...                           } ...
```

この場合，条件式の真偽によらず，挿入した代入文(ロード命令)は実行されるので，いわゆる投機的ロードとなり，この実行が無駄になる可能性もある．

【パターン4－選択的条件分岐】

```
                                int v1 = a->b;
                                int v2 = a->c;
plend;                          plend;
   ⋮        (A)                    ⋮        (A)
if (condition) {                if (condition) {
   ⋮        (A')                   ⋮        (A')
 v = a->b;              ⟹       v = v1;
   ⋮                               ⋮
} else {                        } else {
   ⋮        (A")                   ⋮        (A")
 v = a->c;                       v = v2;
   ⋮                               ⋮
} ...                           } ...
```

else節がある場合は，代入値の選択となるので，可能性のある式をすべて抜き出す．

【エイリアスがあるとき】

HAL言語自体はアドレス参照やポインタ変数定義を認めているので，エイリアスの問題は避けられない．例えば，

```
plend;
   ...
x = a;
   ...
x->b = c;
   ...
v = a->b;
   ...
```

のように，変数aとxが同一対象を指している場合，副作用があるために文“v = a->b;”を移動させることができない．当面は上記のような単純な場合のみを考慮することとし，完全なエイリアス解析は今後の課題としておく．条件Aの成立が判定できれば，これら基本パターンからプリローディング部分の生成は容易となる．この条件判定は，“Reaching Definition”[8),9),10),11)]と呼ばれるアルゴリズムを実現すれば良い．

5. コンパイラ性能評価

本方式で実装したHALコンパイラの性能を測定した．ベンチマークプログラムにストリームプログラミング法による2つのプログラム (sieve, stream_quicksort)，データ並列型プログラム（10queen）および通常のクイックソート（quicksort）を用いて，Java言語で実装されたFuceプロセッサシミュレータ上でそれぞれの実行時間（クロックサイクル数）を計測した．**Fig. 4**の縦軸は，プリローディング最適化行わなかった場合の実行時間と，最適化を施した場合の実行時間の比を表し，横軸はメモリアクセスレイテンシを表している．この図から，メモリアクセスレイテンシが90サイクル以上のとき，プリローディング最適化の恩恵によって2.5倍～6倍以上の性能向上が得られていることがわかる．我々は，Fuceプロセッサの動作周波数にも依存するのであるが，実際のメモリアクセスは最低でも100サイクル以上のレイテンシは生ずるとものと予想している．しかしながら，Fuceのプリローディングユニットおよび HAL処理系によるプリローディング最適化の効果は非常に大きいので，この程度のアクセスレイテンシは許容範囲であろう．

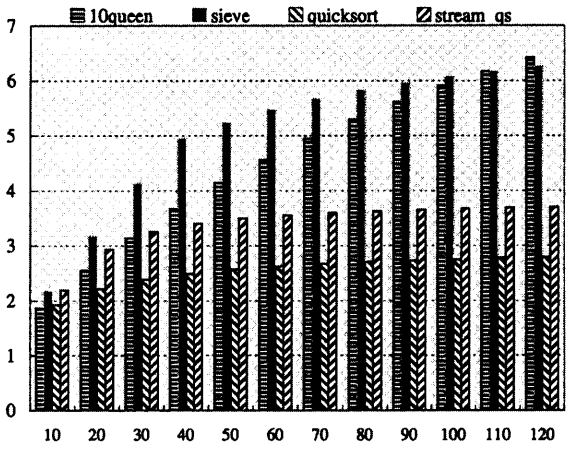

Fig. 4　Effect of pre-loading optimization.

ちなみに，手書きによるアセンブラとHAL処理系の生成したアセンブラでは，若干，手書きの方が速いのであるが，プログラム記述のコストを考えれば，速度差は無視できよう．

6. むすび

本稿では，既存のCコンパイラを援用したFuce用低位言語HALの処理系の実装方法を解説した．また，Fuceに特化した最適化法であるプリローディング最適化の手法を示した．処理系の開発コストを最優先に考えたため，単純で実装が容易な方法を選択したが，小規模なプログラムの実行結果を見ると，我々のコンパイラの生成したコードは十分な性能を持っていることが分かる．今後は，より複雑で規模の大きなベンチマークプログラムを用いて，プリローディング最適化の傾向を調査するつもりである．特に，深くネストした条件分岐を考慮した場合にどのレベルまでプリローディング最適化の対象にするのか，また，プリローディング用に最大で幾つの変数（レジスタ）を割り当てるのが最適か，などは多種多様な現実的なプログラムを用いなければ，一般傾向を考察することは難しいのである．

参考文献

1) 雨宮聡史，松崎隆哲，雨宮真人．“排他実行マルチスレッド実行モデルに基づくオンチップ・マルチプロセッサの設計”，情報処理学会 計算機アーキテクチャ研究会，2003-ARC-155, pp.51-56, (2003).
2) Satoshi Amamiya, Masaaki Izumi, Takanori Matsuzaki, and Makoto Amamiya. The Fuce Processor: The Execution Model and The Programming Methodologies, Proc. of The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'06), (2006).
3) 雨宮聡史，長谷川隆三，藤田 博，越村三幸，雨宮真人: Fuce言語とその処理系について，九州大学大学院システム情報科学紀要，第11巻，第1号，pp.23-30, (2006).
4) 長谷川隆三，藤田博，雨宮聡史，越村三幸，雨宮真人: Fuce上のストリーム処理とその記述言語，九州大学大学院システム情報科学紀要，第11巻，第1号，pp.31-38, (2006).
5) Makoto Amamiya and Rin-ichiro Taniguchi. Datarol: A Massively Parallel Architecture for Functional Language, Proc. IEEE 2nd SPDP, pp.726-735, (1990).
6) Makoto Amamiya and Tetuo Kawano. Design Principle of Massively Parallel Distributed-Memory Multiprocessor Architecture, In L. Bic and J-L. Gaudiot and G. R. Gao, editors, Advanced Topics in Dataflow Computing and Multithreading, pp.1-17, IEEE Press, (1995).
7) GCC, http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc
8) Alfred V. Aho and Jeffrey D. Ullman. Principles of Compiler Design, Addison-Wesley, (1977).
9) Matthew S. Hecht. Flow Analysis of Computer Programs, North-Holland, (1977).
10) Alfred V. Aho , Ravi Sethi and Jeffrey D. Ullman. Compilers - Principles, Techniques and Tools, Addison-Wesley, (1986).
11) 中田育男．コンパイラの構成と最適化，朝倉出版，(1999).