FUCE言語とその処理系について

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

長谷川, 隆三
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員知能システム学部門

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

他

https://doi.org/10.15017/1516219

出版情報：九州大学大学院システム情報科学紀要. 11 (1), pp.23-30, 2006-03-24. 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1号 平成18年3月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.11, No.1, March 2006

FUCE言語とその処理系について

雨宮聡史* ・ 長谷川隆三** ・ 藤田博** ・ 越村三幸** ・ 雨宮真人**

On a FUCE Language and Its Processing

Satoshi AMAMIYA, Ryuzo HASEGAWA, Hiroshi FUJITA,
Miyuki KOSHIMURA and Makoto AMAMIYA

(Received December 9, 2005)

Abstract: In this paper, a language family designed for a FUCE machine is described. The FUCE hardware is viewed as a macro data-flow machine which actually executes "uninterruptable threads" on rather conventional but specially configured multiple processing elements. From the view point of programming languages, such architecture enables us to design and implement high level languages with minimum effort by utilizing conventional techniques or existing compiler tools. Intermediate languages of suitable abstraction in terms of some attached instructions and syntax sugars are very useful both for application programmers and system programmers. Also, a technique to automatically extract FUCE oriented threads from a usual C program is presented.

Keywords: FUCE architecture, Fine-grain thread processing, Language design, Language processing, Compiler

1. まえがき

FUCE[1]は，あらゆるユーザプログラム，および外部割り込み等を扱うOSを含めて，一切の処理を"走り切りスレッド"を単位として効率よく実行するための計算機構である．FUCEプロセッサは，単純なRISC型CPUコアを採用し，これに必要最小限の拡張を施すという設計思想に基づいて開発されている．

我々は，FUCEをターゲットとしたOS記述，および応用プログラム記述を支援するための言語とその処理系について検討と試作を進めてきた．当面，プロトタイプ開発の段階にあるため，できる限り既存の処理系を援用し，開発コストを抑えるよう工夫した．モデルとなる従来型高位言語として，C言語を選んだ．高位から低位までのソフトウェア開発に最も広範に使用され，既存ツールの援用も容易であることを勘案した．

ラピッドプロトタイピングの観点からは，低位の仕様変更が高位まで影響するのを極力避けるために，C言語に準じた機械非依存の高位言語とアセンブラ言語との間に，HAL, IML, SHLという3つの中間言語を置くことにした．HALはもっとも低位な中間言語で，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．IMLはHALの上位中間言語であり，OS記述者が直接ハンドコーディングできることを念頭に置き，HALの冗長な記述を隠蔽し，非常に簡潔にプログラムを記述できるように設計されている．SHLはC言語からFUCEスレッドを自動抽出する際の作業言語として設けた．

本稿では，これらの中間言語を用いて，従来型実行モデルを想定したC言語を，従来とは極めて異なった実行モデルを持つFUCE向けに如何に変換を施すかを詳説する．

平成17年12月9日受付

* 知能システム学専攻博士後期課程
** 知能システム学部門

2. FUCEプロセッサ

FUCEはマクロデータフロー計算モデルを基盤とした効率のよいマルチスレッディング実行を実現するために設計された．似たような計算モデルに基づいたアーキテクチャは過去にいくつもある[10),4),9),5),2),3)]が，あくまでユーザレベルプログラムを実行させるためのアーキテクチャであってOSレベルの低位な処理の効率的実行まで踏み込んで設計されたとは言い難い．FUCEはOSの実行も可能とするように設計されている．さらに現在における半導体技術の進歩を鑑みて，複数のCPUコアを1チップに集約したチップマルチプロセッサとして実装されている．FUCEにおけるスレッドは"走りきり"であり，走行中は他のいかなる干渉も受け付けない．スレッドは後続のスレッドにデータを渡し終えた時に終了する．

Fig. 1にFUCEアーキテクチュアの概要を示す．各スレッド実行ユニット(TEU)は，メインユニットと呼ばれるMIPS型命令セットにいくつかのスレッド制御命令(**Table 1**参照)を追加した非常に単純なRISC型CPUコアと，プリロードユニットと呼ばれるロード命令のみを実行できるCPUコアの対を搭載している．これらCPUコアは2つのレジスタファイルをダブルバッファ方式で利用することでロード命令の遅延を隠蔽し，できるかぎりパイ

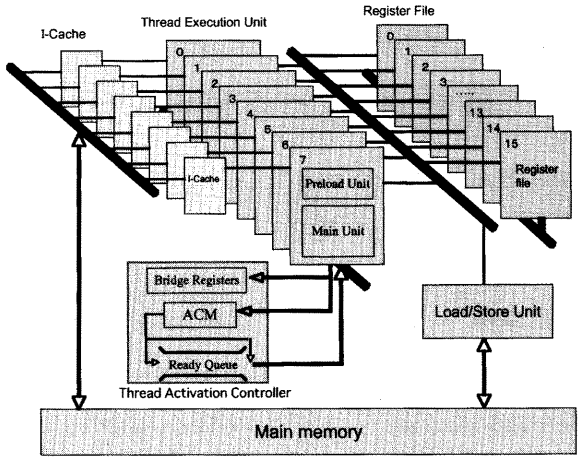

Fig. 1　FUCE architecture.

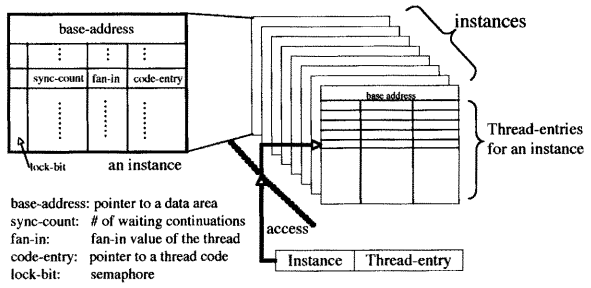

Fig. 2　Activation Control Memory.

プラインストールを起こさずに動作するように設計されている．これに伴って，スレッドとして定義された命令列も２つの部分から成り，先頭部分はロード命令列のみ，本体となる後半部分は任意の命令の列となっている．この先頭部分はコンパイラの命令スケジュールによって構成されることを想定しているが，空でもかまわない．

FUCEにおけるプログラムは，複数の関数群として定義されることを想定しており，実行時には，Activation Control Memory (ACM)と呼ばれる高速メモリと主記憶上に関数インスタンスとしてマップされる．各関数インスタンスはその内部構造として，いくつかのスレッドの組を含んでいるが，ACMはこれらのスレッドに関する情報を保持している．**Fig. 2**に示すように，ACM内部はページングシステムと同様の構造をしていて，各ページ(ブロック)はそれぞれ関数インスタンスに一対一対応している．base-addressとは関数インスタンスが利用するメモリ領域(データエリア)のポインタである．fan-inはスレッドが他のスレッドから受け付けるシグナル(continuation, 継続と呼ぶ) の最大値であり，sync-countはスレッドの現在の残りのcontinuationの数である．code-entryはスレッドの開始アドレスである．sync-countが0になった時，このスレッドは実行可能になり待ち行列に投入される．

Table 1 Instruction Set for FUCE.

arithmetic and 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

TEUが空くと(つまりあるスレッドの実行が終了すると)，待機中の実行可能スレッドがTEUに割り当てられる．

3. FUCE言語系

FUCE言語系は，いくつかのレベルの中間語，ならびにそれらの間の変換系から構成される．本節ではこのうちIMLとHALについて詳細に述べる．

3.1 FUCE中間言語IML

IMLの基本構文はANSI標準のC言語に準拠し，若干の拡張構文要素を導入して設計されている．拡張構文の概要を以下に記す．

関数定義

関数宣言の型名の前に `function` 修飾子を付ける．

データエリア変数

関数は，呼出しごとに異なるインスタンスが生成される．関数インスタンスの仮引数，および局所変数は，主記憶上に置かれる．これをデータエリア（以後，DAと略記）と呼ぶ．局所変数がDAに置かれるべきとき，型名の前に`darea` 修飾子を付ける．`darea` 修飾子を付されない局所変数は，レジスタ上にとられる．

スレッド定義

関数本体はスレッド定義の集合である．各スレッド定義は，`thread` 修飾子を付け，関数内で唯一のスレッド名に続いて本体定義ブロックを書く．ただし，関数の本体定義にあたるブロックは明示的スレッド定義の後に，C言語の通例どおりに置かれる．これは，関数呼出しの際，最初に起動されるスレッド（当該関数の入口スレッドと呼ぶ）の本体定義であり，名前は関数名と同一である．

スレッドのfan-in指定

スレッド名の直後に，`<2>` のようにfan-in指定を書くことができる．省略した場合は `<1>` と書いたものとみなされる．入口スレッドに対しては，関数の仮引数並びの直後にfan-in指定を書く．

```
function int fact(int n) <2> {
  darea int m;
  thread outfact {
    return n * m;
  }
  if (n>0) {
    int p = n-1;
    m := fact(p) => outfact;
  } else {
    return 1;
  }
}
```

Fig. 3 IML code for a factorial function.

DA代入

DA変数への代入演算子は := を用いる.

スレッド遷移

スレッド名を指定し，継続する.

=> スレッド名;

関数呼出し

関数の戻り値は，DA変数に代入されなければならない．また，呼出された関数は別スレッドで走行するから，これに継続スレッド名を通知する必要がある．したがって，関数呼出しでは，実引数渡し，DA変数代入，継続スレッド名指定の3つをひとまとまりとして記述する必要がある．次のように記述する.

DA変数 := 関数呼出し => スレッド名;

自己再帰呼出し

通常，関数呼出し形式で自己再帰を記述した場合，関数インスタンスが新たに生成されることになるが，比較的重たい処理である領域確保を避けるために同じ関数インスタンスを再利用したい場合もある．このような場合は自己再帰呼出し：

recur(実引数並び);

を記述すればよい．このとき，実引数は現インスタンス上の仮引数を上書きすることになる.

最終実行文

各スレッドの最終実行文は，通常の関数の戻り return か，スレッド遷移文か，関数の自己再帰呼出しのいずれかでなければならない.

Fig. 3に階乗関数のIMLコード記述例，**Fig. 4**に同HALコード例を示す.

3.2 IMLからHALへの変換

中間言語HALでは，DA変数は関数ごとにまとめられ，継続スレッドの識別子(return_thid)と関数の戻り値を格納すべきメモリアドレス(*return_val) とともに，関数名を付した構造体定義に準じた形式が作られる.

各スレッドの定義ブロック内では，通常の文および式はC言語に準拠しているが，DA変数構造体を変数baseで参照することができ，また，自スレッド番号を変数idで参照できるようにしているため，これらは予約されている.

```
int fact(int n) <2> {
  fact_darea {
    int n;
    int m;
    int return_thid;
    int *return_val;
  }
  thread out_fact {
    int n;
    int m;
    int ret_id;
    n = base->n;
    m = base->m;
    ret_id = base->return_thid;
    *(base->return_val) = n*m;
    continue(ret_id);
    delins(id);
    delda(base);
    end;
  }
  thread fact <2> {
    int n;
    n = base->n;
    if (n>0) {
      fact_darea *f1_da;
      int f1;
      int p;
      p = n-1;
      f1_da = newda(fact);
      f1 = newins(fact,f1_da);
      f1_da->n = p;
      f1_da->return_thid = fact;
      f1_da->return_val = &(base->m);
      continue(f1);
    }
    else {
      int ret_id;
      ret_id = base->return_thid;
      *(base->return_val) = 1;
      continue(ret_id);
      delins(id);
      delda(base);
      end;
    }
  }
}
```

Fig. 4 HAL code for a factorial function.

Fig. 3と**Fig. 4**を見比べるとわかるように，IMLコードからHALコードへの変換は逐語的に行える．その変換テンプレートを**Table 2**に示す.

4. スレッドの自動抽出

関数型言語を対象としたスレッド抽出手法[6),7),8)] が提案されているが，OS等の低位な処理の記述を念頭に置くと，関数型言語では不十分である．C言語を対象にするのが適切である．一般のC言語のプログラムをFUCE上で実行可能な形式に変換するためには，まず走り切りスレッドに分解することが必要である．本節では，中間言語

Table 2 Templates for translate IML into HAL.

IML code	HAL code
DA variable := exp ;	assignment to a member of DA structure
=> *thread_name* ;	set actual parameters; continue to *thread_name*;
DA variable := *function_name(...)* => *thread_name* ;	acquire a data area; acquire a ACM instance; set actual parameters; set return thread as *thread_name*; set pointer to the returned value; continue to *function_name*;
`return` *exp* ;	set `ret_id`; assign the value of *exp* to `return_val`; continue to `ret_id`; release the ACM instance; release the data area; end of thread;

SHLを用いて，これを自動化する手法について述べる．

4.1 スレッド番号付与法

SHLでは，関数呼出しの実引数を変数に限定する．実引数の値は関数呼出し以前に決定しておく必要がある．

プログラム中の各要素がどのスレッドに属するべきかを区別するため，以下のようにスレッド番号(thid)を付与することにする．(thidは上付き数字で表す.)

手続き中，基礎番号(baseid)，現番号(curid)，最大番号(maxid) を各々カウンタに保持する．

初期値

ブロックの開始時，通常，baseid=curid=0

定数

定数のthidは0.

関数呼出し

baseid，ならびに引数のthidのうち最大の値．

例 1　baseid = 0 のとき $\overset{2}{f}(\overset{2}{a},\overset{1}{c})$，baseid = 3 のとき $\overset{3}{f}(\overset{2}{a},\overset{1}{c})$，など

curid=maxid=maxid+1

表記法：たとえば，関数fの呼出しthid= m，maxid=n のとき，$\overset{m:n}{f}$ と表す．

関数値代入文

$y = f(a)$の左辺の変数yのthidは，fの呼出し時のmaxidである．

例 2　$\overset{3}{y} = \overset{0:3}{f}(\overset{0}{a})$

一般代入文

$y = expr$の左辺の変数yのthidは，右辺のthidによらずcuridである．代入自体は左辺のthidのスレッドで実行される．†1

†1 変数は単一代入ではないので，変数のthidとは，正確には変数出現のthidのことである．したがって，$\overset{0}{a}=1;\ \overset{1}{a}=\overset{0:1}{g}(\overset{0}{a});\ \overset{1}{b}=\overset{1}{a}+2$ のように，各文のthidは，最新の変数出現のthidを使って計算される．

例 3　curid = 3のとき，$\overset{3}{e} = (\overset{0}{a} * \overset{1}{b}) \ / \ (\overset{1}{c} * \overset{2}{d})$

if文

$if\ (\ P\ )\ \{\ Q\ \}\ else\ \{\ R\ \}$のthidは，if文に入る直前のcuridである．これをif文のbaseidと呼ぶ．then部，else部の各ブロックの開始時にcurid=baseidである．

例 4　$\overset{0}{a}=1;\overset{0}{n}=2;$
$\overset{0}{if}\ (\overset{0}{n}>0)\ \{\overset{1}{y}=\overset{0:1}{g}(\overset{0}{a});\}\ else\ \{\overset{2}{y}=\overset{0:2}{h}(\overset{0}{a});\}$

if文の直前のcuridは0なので，ifのthidもbaseidも0となる．$g(a)$の呼出しthidは$\max(\overset{0}{a},\overset{0}{\text{baseid}})=0$ である．$h(a)$も同様．

例 5　$\overset{0}{a}=1;\overset{0}{n}=2;\ \overset{1}{b}=\overset{0:1}{f}(\overset{0}{a});$
$\overset{1}{if}\ (\overset{0}{n}>0)\ \{\overset{1}{t}=\overset{1}{b}*\overset{0}{a};\ \overset{2}{y}=\overset{1:2}{g}(\overset{1}{t});\}\ else\ \{\overset{3}{y}=\overset{1:3}{h}(\overset{0}{a});\}$

if文の直前のcuridは1なので，ifのthidもbaseidも1となる．$t = b * a$のtのthidはcuird=1である．$g(t)$の呼出しthidは$\max(\overset{1}{t},\overset{1}{\text{baseid}})=1$，$h(a)$の呼出しthidは$\max(\overset{0}{a},\overset{1}{\text{baseid}})=1$ となる．

例 6　$\overset{0}{a}=1;\overset{0}{n}=2;\ \overset{1}{b}=\overset{0:1}{f}(\overset{0}{a});$
$\overset{1}{if}\ (\overset{0}{n}>0)\ \{\overset{2}{t}=\overset{1:2}{g}(\overset{0}{a});\ \overset{2}{y}=\overset{1}{b}*\overset{2}{t};\}\ else\ \{\overset{3}{y}=\overset{1:3}{h}(\overset{0}{a});\}$

if文の直前のcuridは1なので，ifのthidもbaseidも1となる．$t = g(a)$において，$g(a)$の呼出しthidは$\max(\overset{0}{a},\overset{1}{\text{baseid}})=1$，curid=maxid=maxid+1=2となり，tのthidは2である．また，$y = b * t$において，yのthidはcurid=2である．

if文の条件部(P)に関数呼出しがある場合は，SHLでは，これをif文の外側で処理し，条件部には関数呼出しのない関係式のみを置く．

例 7　$a = 1; n = 2;$
$if\ (f(n) > 0)\ \{y = g(a);\}\ else\ \{y = h(a);\}$
$\overset{toSHL}{\Rightarrow}$
$\overset{0}{a}=1;\overset{0}{n}=2;\overset{1}{p}=\overset{0:1}{f}(\overset{0}{n});$
$\overset{1}{if}\ (\overset{1}{p}>0)\ \{\overset{2}{y}=\overset{1:2}{g}(\overset{0}{a});\}\ else\ \{\overset{3}{y}=\overset{1:3}{h}(\overset{0}{a});\}$

配列代入

$A[i] = expr$の左辺の配列参照のthidは，右辺のthidによらずcuridである．代入は左辺のthidのスレッドで実行される．

例 8 curid = 1のとき $\overset{1}{A}[\overset{1}{i}] = \overset{0}{a} * 2$，curid = 3のとき $\overset{3}{A}[\overset{2}{j}] = A[\overset{3}{i}]$，

while文

$while$ (P) { Q } は，SHLでは，$wbegin$ $\{P0\}$ $(P1)$ $\{Q\}$ $wend$ に細分化する．$\{P0\}$をwhile開始部(wbegin)，$(P1)$をwhile条件部 (wcond)，$\{Q\}$をwhile本体部 (wbody) と呼ぶ．Pが関数呼出しを含まなければ，$P0 = \emptyset$, $P1 = P$である．さもなければ，Pは関数呼出し処理$P0$と関数呼出しを含まない関係式$P1$ に分解される．

wbeginに対し，新スレッド番号(curid=curid+1)を付与する．これがwhile開始部$\{P1\}$のbaseidとなる．

1. while文の条件部(P)が関数呼出しを含まない場合を以下に例示する．

   例 9

   $i = 0;$ $while$ $(i > 0)$ $\{...\}$

   $\overset{toSHL}{\Rightarrow}$ $\overset{1}{wbegin}$ $\{\}$ $(\overset{0}{i} > 0)$ $\{...\}$ $wend$

   wbeginに新スレッド番号1を付与する．元の条件部が関数を含まないのでwhile開始部は{}である．この場合，wbeginのthidをwhile本体部のbaseidに設定する．

2. while文の条件部(P)に関数呼出しがある場合は，SHLでは，これをwhile開始部で処理し，while条件部には関数呼出しのない関係式のみを置く．

   例 10

   $i = 0;$ $while$ $(f(g(i))) > 0)$ $\{...\}$

   $\overset{toSHL}{\Rightarrow}$

   $\overset{0}{i} = 0;$ $\overset{1}{wbegin}$ $\{\overset{2}{y1} = \overset{1:2}{g}\ (\overset{0}{i});$ $\overset{3}{y2} = \overset{2:3}{f}\ (\overset{2}{y1});\}$

   $(\overset{3}{y2} > 0)$ $\{...\}$ $wend$

   wbeginに新スレッド番号1を付与する．これがwhile開始部のbaseidとなる．$y1 = g(i)$において，$g(i)$の呼び出しthidは$\max(\overset{0}{i}, \text{baseid}) = 1$, curid=maxid=maxid+1 =2となり，$y1$のthidは2である．また，同様に，$y2 = f(y1)$における$y2$のthidは3である．while開始部から最大スレッド番号3が得られた．これがwhile本体部のbaseidとして設定される．

3. while本体部が関数呼出しを含まなければスレッド分割は生じないが，さもなければ，関数呼出しの度に分断されることになる．wendを実行するwhile本体最後のスレッドは，wbeginスレッドに遷移することに注意．

   例 11

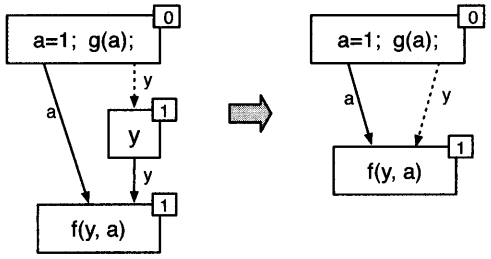

Fig. 5 Dependency graph for actual parameters.

$i = 10; j = 2;$

$while$ $(i > 0)$ $\{j = j * 2; i = i - 1\}$

$\overset{toSHL}{\Rightarrow}$

$\overset{0}{i} = 10; \overset{0}{j} = 2;$

$\overset{1}{whilebeign}$ $\{\}$ $(\overset{0}{i} > 0)$ $\{\overset{1}{j} = \overset{0}{j} * 2; \overset{1}{i} = \overset{0}{i} - 1;\}$

$wend$

条件部も本体部も関数呼出しを含まない例である．wbeginのthidは1．これがwhile本体部のbaseidとなる．この例の場合，wbeginからwendまでの全体が1番スレッドで実行される．

例 12

$i = 10;$

$while$ $(f(i) > 0)$ $\{j = g(i); i = i - 1;\}$

$\overset{toSHL}{\Rightarrow}$

$\overset{0}{i} = 10;$

$\overset{1}{wbeign}$ $\{\overset{2}{p} = \overset{1:2}{f}\ (\overset{0}{i});\}$ $(\overset{2}{p} > 0)$

$\{\overset{3}{j} = \overset{2:3}{g}\ (\overset{0}{i}); \overset{3}{i} = \overset{0}{i} - 1;\}$ $wend$

wbeginのthidは1．while開始部から最大スレッド2が得られ，これがwhile本体部のbaseidとなる．$j = g(i)$において，$g(i)$の呼出しthidは$\max(\overset{0}{i}, \overset{2}{\text{baseid}}) = 2$, curid=maxid=maxid+1=3となり，jのthidは3である．また，$i = i - 1$において，左辺のiのthidはcurid=3である．例12からは，

$\{\overset{0}{i} = 10;\}$, $\{\overset{1}{wbegin}$ $\{\overset{2}{p} = \overset{1:2}{f}\ (\overset{0}{i});\}\}$,

$\{(\overset{2}{p} > 0)$ $\{\overset{3}{j} = \overset{2:3}{g}\ (\overset{0}{i})\}\}$, $\{\{\overset{3}{j};$ $\overset{3}{i} = \overset{0}{i} - 1;\}$ $wend\}$

が，それぞれ0〜3番スレッドとして抽出される．

4.2 依存性グラフ

スレッド間の変数参照関係を表したグラフを依存性グラフと呼ぶ．図中大きな箱はスレッドを，小さな箱はスレッド番号を表す．また，関数値代入において，関数呼出しを行うスレッドと関数値を受け取るスレッド間の変数参照は破線矢で表し，それ以外の変数参照は実線矢で表す．

Fig. 5は，$\overset{0}{a} = 1;$ $\overset{1}{y} = \overset{0:1}{g}\ (\overset{0}{a});$ $\overset{2}{z} = \overset{1:2}{f}\ (\overset{1}{y}, \overset{0}{a})$ に対する依存性グラフである．0番スレッドで定義されたaを1番スレッドで参照しているので，0から1へ実線矢を引き，関数値

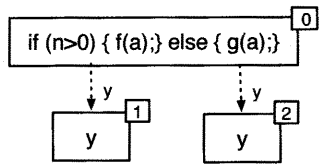

Fig. 6　Dependency graph for an if-statement.

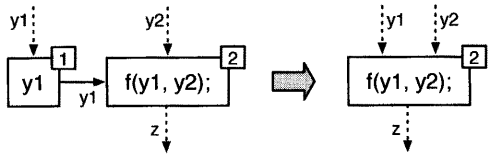

Fig. 7　Example of thread Fusing.

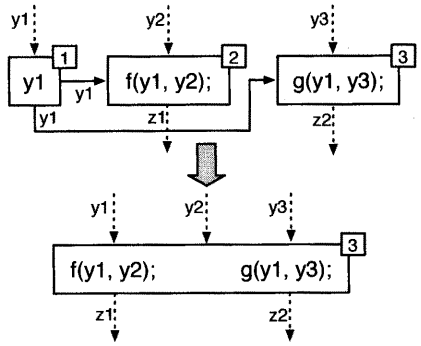

Fig. 8　Bad Example of thread Fusing.

代入$\overset{1}{y}=\overset{0:1}{g}\ (\overset{0}{a})$; によって定義される値$y$には破線矢を引く．$y$と$f(y,a)$の間も実線矢で結んだが，これらは同じ1番スレッドで実行されるので，**Fig. 5** 右側のように，一つのスレッドとしてまとめる．

Fig. 6は，$\{\overset{0}{if}\ (\overset{0}{n}>0)\ \{\overset{1}{y}=\overset{0:1}{f}\ (\overset{0}{a});\}\ else\ \{\overset{2}{y}=\overset{0:2}{g}\ (\overset{0}{a});\}$ に対する依存性グラフである．if文のthen部およびelse部の出力先（もしあれば）に対してそれぞれから矢をひく．この例の場合，then部の$f(a)$，else部の$g(a)$から同じyの名で出力されるが，これらはいずれか一方にしか値が送出されない排他的出力である．

4.3 スレッド融合

変数名のみからなるスレッドは，もしそれを参照しているスレッドが唯一であればそのスレッドに融合する．例えば，**Fig. 7**のように，スレッド1で定義された$y1$をスレッド2が唯一参照している場合，1を2に融合する．**Fig. 8**上部において，スレッド2,3がスレッド1の$y1$を参照しているが，2と3の間に参照関係はない．これを**Fig. 8** 下部のように融合してしまうと，$f(y1,y2)$と$g(y1,y3)$は$y1,y2,y3$が全て揃った後にしか計算されず，並列性を損なうことになる．よって，参照関係のない独立スレッド2,3は融合しないことにする．このような融合不可の場合

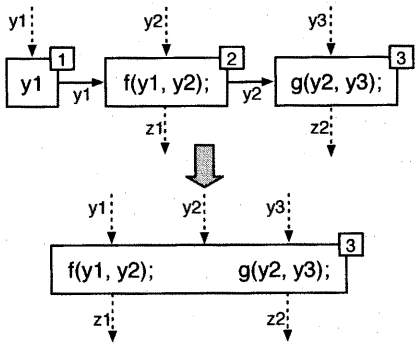

Fig. 9　Another example of thread fusing.

に対応するSHLの例として，

(a) $\overset{1}{y1}=\overset{0:1}{h1}\ (\overset{0}{a});\ \overset{2}{y2}=\overset{0:2}{h2}\ (\overset{0}{a});\ \overset{3}{y3}=\overset{0:3}{h3}\ (\overset{0}{a});\ \overset{4}{z1}=\overset{2:4}{f}\ (\overset{1}{y1}\ ;\overset{2}{y2});\ \overset{5}{z2}=\overset{3:5}{g}\ (\overset{1}{y1};\overset{3}{y3});$ や，

(b) $\overset{1}{y1}=\overset{0:1}{h1}\ (\overset{0}{a});\ \overset{1}{if}\ (\overset{0}{n}>0)\ \{\ \overset{2}{y2}=\overset{1:2}{h2}\ (\overset{0}{a});\ \overset{3}{z1}=\overset{2:3}{f}\ (\overset{1}{y1}\ ;\overset{2}{y2});\}\ else\ \{\ \overset{4}{y3}=\overset{1:4}{h3}\ (\overset{0}{a});\ \overset{5}{z2}=\overset{4:5}{g}\ (\overset{1}{y1};\overset{4}{y3});\}$

がある．

スレッド$th_1,\ldots,th_n$間に実線矢だけの参照関係の連鎖$th_1\rightarrow th_2, th_2\rightarrow th_3,\ldots,th_{n-1}\rightarrow th_n$が存在すれば，これら$n$個のスレッドを$th_n$に融合する．例えば，**Fig. 9** はSHL

$\overset{1}{y1}=\overset{0:1}{h1}\ (\overset{0}{a});\ \overset{2}{y2}=\overset{0:2}{h2}\ (\overset{0}{a});\ \overset{3}{y3}=\overset{0:3}{h3}\ (\overset{0}{a});\ \overset{4}{z1}=\overset{2:4}{f}\ (\overset{1}{y1}\ ;\overset{2}{y2});\ \overset{5}{z2}=\overset{3:5}{g}\ (\overset{2}{y2};\overset{3}{y3});$

を表しているが，スレッド1,2,3の間には，$1\rightarrow 2, 2\rightarrow 3$ という参照関係があるので，1,2を3に融合する．

4.4 Fan-in数の決定

> 融合操作後の依存性グラフにおいて，スレッドのfan-in数は関数値定義変数参照を表す破線矢入力の総数となる．

[証明] 今，変数およびスレッドに対し，依存関係によるスレッド連鎖の段数を示すレベル決定関数lを次のように定義する．始スレッドの入力Iに対し，$l(I)=0$．あるスレッドT_iの入力を$I_1,\ldots,I_n$としたとき，$l(T_i)=max\{l(I_1),\ldots,l(I_n)\}+1$．$T_i$内で定義される変数および$T_i$の破線出力のレベルを$l(T_i)$ に設定する．例えば，T_1からT_2，T_2からT_3が，それぞれ破線y,zで，T_1からT_3が実線xで結ばれている場合，$l(T_1)=l(x)=l(y)=1, l(T_2)=l(z)=2, l(T_3)=3$となる．各レベルの入出力は必ず定義されることをレベルに関する帰納法により示す．

[ベース] 始スレッドの入力は定義されており，スレッド内で行われる演算はすべて定義される．スレッド内では，関数呼び出しの後にはそれ以外の文は続かない．従って，関数値（破線出力）が定まった時点では，始スレッド内で定義される変数はすべて参照可能である．

```
int sieve(int p, int s[]) {
    int i;
    i=p+1;
    while (i<MAX) {
        if (i%p==0) s[i]=1;
        i=i+1;
    }
    p=p+1;
    while (p<MAX && s[p]==1) p=p+1;
    return p;
}

void main(void) {
    int s[MAX];
    int i, p;
    s[0]=1;
    s[1]=1;
    i=2;
    while (i<MAX) {
        s[i]=0;
        i=i+1;
    }
    p=2;
    while (p<MAX) p=sieve(p, s);
}
```

Fig. 10 An example C program.

[帰納ステップ] スレッドT_iの入力が破線ならば，帰納法の仮定よりこれは定義されている．入力が実線ならば，次のことに注意．融合操作後は，(始以外の)スレッドは必ず破線入力を持ち，実線入力のみをもつことはない．よって，実線入力は破線入力以前のレベルで定義されている．T_iの入力はすべて参照可能なので，T_iの破線出力も定まる． ∎

以上より，あるスレッドの破線入力が全て揃えば，そのスレッド内で参照している実線入力の値も全て定まっている．よって，当該スレッドのfan-in数は，破線入力の総数に設定すればよい．

4.5 Cプログラムからの抽出例

以上の方法にしたがって，CプログラムからFUCEプログラム（IMLコード）を導くことができる．**Fig. 10**にCプログラムの例を示す．これは，エラトステネスの篩に基づいて素数を求めるプログラムの主要部分である．関数sieveが素数ごとの篩の動作を行い，main関数は，新たな素数が求まるごとに関数sieveを呼出す．**Fig. 11**にスレッド抽出例を示す．元のCプログラムにおける関数呼出しは1箇所のみであるが，while文が含まれるため，6個のスレッド（そのうち2個は入口スレッド）が生じる．こうして，各while文は，繰返し実行部がスレッド化され，繰返し制御は，if文とrecur文，および別スレッドへの遷移文の形に変換されている．

```
function int sieve(int p, int s[]) {
    darea int i;
    thread seive_1 {
        if (i<MAX) {
            if (i%p==0) s[i]:=1;
            i:=i+1;
            recur;
        } else {
            p:=p+1;
            => sieve_2
        }
    }
    thread seive_2 {
        if (p<MAX && s[p]==1) {
            p:=p+1;
            => sieve_1;
        } else return p;
    }
    i:=p+1;
    => sieve_1;
}

function void main() {
    darea int s[MAX];
    darea int i, p;
    thread main_1 {
        if (i<MAX) {
            s[i]:=0;
            i:=i+1;
            recur;
        } else {
            p:=2;
            => main_2;
        }
    }
    thread main_2 {
        if (p<MAX)
            p:=sieve(p, s)=>main_2;
        else end;
    }
    s[0]:=1;
    s[1]:=1;
    i:=2;
    => main_1;
}
```

Fig. 11 IML code as a result of thread-extraction.

5. むすび

本稿は，FUCEのための言語の設計と実装について述べた．C言語文法にほぼ準拠した中間言語HALやIMLを設けることにより，高位言語から機械語に至る変換過程でgccといった既存の最適化Cコンパイラを援用でき，開発コスト懸けずに済んだ．多段の中間語を経るため，プログラムごとの変換コストが問題となり得るが，今後，生成コードの評価や言語自体の修正／拡張の発展性を考慮すると，ラピッドプロトタイピング容易性のメリットの方が大きい．また，従来型実行モデル用のプログラムから，走り切りスレッドを抽出してFUCEプログラムに変換する手法に関しても，簡単なデータフロー解析で自動化できることを示した．本稿で触れられなかった，OS

記述等に不可欠な並行プログラミング等の技法に関する詳細については，別稿にて報告する．

謝　辞

本研究は，日本学術振興会 科学研究費補助金 基盤研究(A)「細粒度マルチスレッド処理原理による並列分散処理カーネルウェアの研究」(課題番号: 15200002)の一環として行った．

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) Makoto Amamiya and Rin-ichiro Taniguchi. Datarol: A Massively Parallel Architecture for Functional Language, Proc. IEEE 2nd SPDP, pp.726-735, (1990).
3) 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).
4) David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine, In Proc. of 4th ASPLOS, pp.164-175, (1991).
5) Herbert H.J. Hum et al. A Design Study of the EARTH Multiprocessor, In the Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'95), pp.59-68, (1995).
6) Kentaro Inenaga, Shigeru Kusakabe, Tetsuro Morimoto, and Makoto Amamiya. Hybrid Approach for Non-strict Dataflow Program on Commodity Machine, Proc. of Intl. Symp. on High Performance Computing (ISHPC'97), pp.243-254, (1997).
7) Shigeru Kusakabe et al. Implementation of a Non-strict Functional Programming Language V on a Threaded Architecture EARTH, Proc. of Intl. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'98), p.95, (1998).
8) Shigeru Kusakabe et al. Implementing a Non-strict Functional Programming Language on a Threaded Architecture, Fourth Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99), pp.138-152, (1999).
9) Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture, Proc. of Intl. Symp. on Computer Architecture (ISCA'92), pp.156-167, (1992).
10) Gregory M. Papadopoulos, and David E. Culler. Monsoon: an explicit token-store architecture, Proc. of Intl. Symp. on Computer Architecture (ISCA'90), pp.82-91, (1990).