Committed-Choice 型言語 Fleng における粒度制御法の評価

荒木拓也[†]　田中英彦[†]

Committed-Choice 型言語 Fleng は，データフロー同期の機構を用いることにより，容易に大量の並列性を抽出することが可能である．しかし，実行の粒度が非常に細かいため，同期やゴール（Fleng における実行の単位）の起動といった，細粒度実行に由来するオーバヘッドが大きい．プログラムの粒度を大きくすることができれば，オーバヘッドを低減することができる．しかし，Fleng においてプログラムの意味を変えずに粒度を大きくすることは容易ではない．我々は Fleng における粒度制御法としてゴール融合と強制インライン展開を提案している．ゴール融合法は複数のゴールを 1 つのゴールにまとめることにより，粒度を大きくする手法である．安全にゴール融合を行うためには精密なデータフロー解析が必須であるため，解析に時間がかかったり，大きなプログラムでは十分粒度を大きくできない可能性がある．また，強制インライン展開は同期部分のコードを含めてインライン展開することにより粒度を大きくする手法である．強制インライン展開ではコード量が増大したり，同期部分のコードが残ることによりオーバヘッドが完全に排除できない可能性がある．本研究では実行時間，コンパイル時間，出力コードの大きさなどについて，このこの 2 つの手法の比較評価と検討を行う．

Evaluation of Granularity Control Methods of a Committed-Choice Language Fleng

Takuya Araki[†] and Hidehiko Tanaka[†]

A committed-choice language Fleng can extract much parallelism easily from any programs using dataflow synchronization. However, there is large overhead such as synchronization and goal (a unit of Fleng computation) invocation, because the granularity of execution is very fine. If granularity of a program is coarsened, such overhead can be reduced; but it is not easy without changing semantics of the program. We proposed goalfusion and force inlining as granularity control methods of a Fleng program. Goalfusion method coarsens granularity by fusing goals into one goal. In order to fuse goals safely, precise dataflow analysis is required, which may take a lot of time and may make it impossible to coarsen granularity of a large program. Force inlining method coarsens granularity by inlining goals including synchronization code. Force inlining method may explode the size of a program and may not delete overhead completely because synchronization codes still remain after inlining. In this study, we evaluate these two methods by execution time, compilation time, output code size, and so on.

1. はじめに

Committed-Choice 型言語 Fleng[2] は，データフロー同期の機構を用いることにより，容易に大量の並列性を抽出することが可能である．しかし，実行の粒度が非常に細かいため，同期やゴール（Fleng における実行の単位）の起動といった，細粒度実行に由来するオーバヘッドが大きい．プログラムの粒度を大きくすることができれば，オーバヘッドを低減することができる．しかし，Fleng においてプログラムの意味を変えずに粒度を大きくすることは容易ではない．

本来，粒度と並列度はトレードオフの関係にあり，粒度を大きくする際には並列度の低下に注意する必要がある．しかし，Fleng では粒度を大きくすることが大変困難であるため，現在は，並列度の低下には注意を払わず，粒度を大きくすることだけを目的として研究を進めている．

我々は Fleng における粒度制御法としてゴール融合[7] と強制インライン展開[6] を提案している．それぞれの手法は実行時間やコンパイル時間，コードサイズなどの面で利点，欠点があると思われる．

本研究では，強制インライン展開手法の実装を行い，文献 7) で実装，評価を行ったゴール融合法との比較，

† 東京大学工学系研究科
School of Engineering, The University of Tokyo

評価を行う．両方の手法を同時に適用した場合も評価の対象とした．

本論文の構成は以下の通りである．2章で，対象言語である Fleng について説明し，3章で Fleng の基本的な実装技術について説明する．4章で，ゴール融合法について簡単に説明する．5章で，強制インライン法について説明し，6章で，強制インライン法の実装について説明する．そして，7章で両手法の比較評価を行い，8章で関連研究について述べる．最後に 9章でまとめを行う．

2. Committed-Choice 型言語 Fleng

Fleng は論理型言語を祖先とする並列記号処理言語である．シンタックスは Prolog のものとよく似ているが，バックトラックを行わないという点で，セマンティクスは大きく異なる．

Fleng は，

- すべてのゴールを並列に実行する
- 単一代入変数を用いたデータフロー同期をとる

ことにより，高並列なプログラムの実行が可能である．この点が Fleng の大きな特徴になっている．以下に例をあげながら説明する．

```
foo(A,R):- add(A,1,B), mul(B,2,R).
```

これは，R = (A + 1) * 2 を実行する述語 foo の定義である．述語は 1 つまたは複数の定義節で定義される．この場合は 1 つの定義節で定義されている．定義節は “:-” で区切られており，“:-” の左側をヘッド，右側をボディという．

Fleng における計算の単位はゴールと呼ばれる．このプログラムの場合，foo(A,R) という初期ゴールが与えられると，add(A,1,B)，mul(B,2,R) という 2 つのゴールに書き換えられる．この書き換えの操作をリダクションという．書き換えられた add と mul はそれぞれが並列に実行される．しかし，この場合，mul の方は B の値が決定するまで実行することはできない．このような場合，add の実行が終了し B の値が決まるまで，mul は実行を中断（サスペンド）し，add の実行が終了し B の値が決定すると，実行を再開（アクティベイト）する．この機構を矛盾なく実現するため，変数は単一代入であり，書き換えることはできない．

分岐は次のように表す．

```
foo(true,R):- R = 1.
foo(false,R):- R = 0.
```

この述語は 2 つの定義節から構成されており，第一引数が true ならば R = 1，false ならば R = 0 を実行する．ここで，true や false のように小文字で始まるものはシンボルを表す．変数は大文字で始まる．この場合，さきほどと同様に，第一引数がきまるまでサスペンドし，値が決まったところでアクティベイトされ，その値によって分岐を行う．

また，算術演算は

```
add(#A,#B,R):- compute(+,A,B,R).
```

のように定義されている．ここで “#” のついた変数は具体化されるまで待つことを表す．また，“compute” はサスペンドせずに実行し，ほぼアセンブリ言語の add 命令にコンパイルされる．したがって，compute を用いる際は#によって値が具体化されていることを保証しなければならない．

compute と=はサスペンドせずに実行可能なため，ゴールのフォークではなく，リダクション時に直接実行される．また，複数の compute，=があった場合は，記述順に逐次に実行される．

また，ゴール融合や強制インライン展開は Fleng プログラムから Fleng プログラムへのトランスレータとして実現されている．そのため，ボディー内で分岐が扱えるよう，コンパイラを拡張してある．ボディー内での分岐は (Cond--> Then ; Else) で表す．この分岐は条件部の Cond を評価するときに必要な変数はすべて値が決まっていることを仮定し，サスペンドを起こさずに実行する．このため，この分岐を Non-Blocking If-Then-Else と呼び，コンパイルされると，手続き型言語と同様，単なる分岐命令にコンパイルされる．

先ほどの分岐の例を Non-Blocking If-Then-Else を用いて表現すると次のようになる．

```
foo(#A,R):-  (A == true --> R = 1; R = 0).
```

3. 基本的な実装技術

データフロー同期を実現するため，ゴールは計算機の中では**図 1** のような形で表されている．この図は先ほどの例

```
foo(A,R):- add(A,1,B), mul(B,2,R).
```

で，foo(3,R) を呼びだし，mul が add よりも先に実行されようとしたため，サスペンドしている状態である（説明のため，簡略化してある）．

foo(3,R) を呼び出した時点で add と mul の 2 つのゴールにリダクションされる．この時点で，述語名，引数といったゴールに関する情報がメモリ上に保存される．これをゴールフレームと呼ぶ．変数 B はリダクション時は未定義なので，UDF（UNDEF: 未定義）型の変数としてメモリ上に確保し，ゴールフレームにはその領域へのポインタ（VAR で表す）を書き込む．

図のように mul がサスペンドした場合は，サスペ

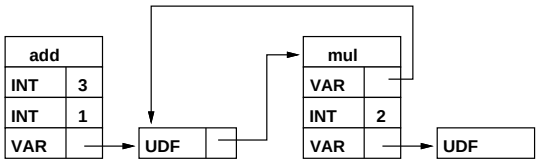


図 1 ゴールの表現
Fig. 1 Representation of goals.

ンドした原因の変数 B に mul へのポインタを保持する☆．add の実行が終り，B の値を決定する際には，B を待ってサスペンドしているゴール（mul）のゴールフレームをたぐり，アクティベイトする．

新たにゴールを実行する際は，ゴールフレームから引数などの情報を取り出して実行する．ここで，リダクションが終了した後，次に実行するゴールは今リダクションした最後のゴールであることを考えると，最後のゴールのゴールフレームの作成は冗長である．直前にゴールフレームにセーブした情報と同じ情報をロードすることになるためである．このため，最後のゴールはゴールフレームを作成せず，レジスタに引数を設定した後，その述語の定義にジャンプすることで実行を行う．これを末尾呼び出しの最適化と呼ぶ．末尾呼び出しをする述語が自分自身の場合（末尾再帰の場合）は，ループの実行と同様になる．

以上に述べたような機構のみを用いて実装した場合，

- ゴールフレームの確保
- ゴールフレームへの情報のセーブ/ロード
- 値が決定しているかのチェック

などのオーバヘッドがかかる．特に add などの算術演算の場合，C などの逐次言語ではほぼアセンブラ 1 命令で済むことを考えると，大変なオーバヘッドだといえる．

4. ゴール融合

このようなオーバヘッドを削減するために，文献 7) で，ゴール融合という手法を提案/実装した．この手法は複数のゴールを 1 つにまとめることで，オーバヘッドを削減するという手法であり，一種のプログラムの逐次化である．先ほどあげた例

```
foo(A,R):- add(A,1,B), mul(B,2,R).
```

の場合は次のようになる．

```
foo(A,R):- add_mul(A,R).
add_mul(#A,R):-
  compute(+,A,1,B), compute(B,2,R).
```

このように 2 つのゴール add と mul を 1 つにまとめている．

このようにすることで，add_mul を末尾呼び出しすれば，ゴールフレーム作成の手間ははなくなる．また，mul は add の後に実行するということが分かっているので，値が決まっているかどうかをチェックする同期コードも不要になる．また，2 つの compute の間の値の授受に使われている変数 B は生成時に値が決まっていることが保証されるので，メモリをアロケートせず，レジスタを使えるようになるというのも重要な点である．

ただし，このような融合が常に可能なわけではない．安易なゴール融合は，プログラムのデッドロックを引き起こす．そのため，プログラムのデータフロー解析を行い，プログラムの意味を変えないことを保証して行う必要がある．

依存関係の解析は,「存在するかも知れない」といった潜在的なものも含めて解析し，保守的に行う必要がある．データフロー解析の精度が融合できるゴールの数に大きな影響を与えるため，プログラム全体にわたる大域的なデータフロー解析を行っている．

大きなプログラムでは十分な解析が行えなかったり，複数のファイルで構成されるプログラムでは，完全な情報の伝搬が行えなかったりするため，十分な数のゴールが融合できない可能性がある．また，大きなプログラムの場合は特に，データフロー解析に長い時間がかかるため，コンパイル時間が長くなるという問題があった．

5. 強制インライン展開

そこで，我々は，データフロー解析の必要のない粒度制御手法として，強制インライン展開を提案した[6]．通常のインライン展開では，ヘッド部に同期がない（サスペンドしない）述語しかインライン展開できないが，強制インライン展開では，ヘッドによる同期部分も含めてインライン展開を行う．たとえば，

```
add(#A,#B,R):- compute(+,A,B,R).
```

という算術演算は，A，B の値を待たなければ計算をしてはならないので，compute をそのままインライン展開することはできない．ここで，このコードのヘッド部にある同期部分（#で値を待っている部分）を Non-blocking If-Then-Else を用いてボディー部に移動することを考える．たとえば，上記の add は

```
add(A,B,R):-
  (isvar(A)--> suspend(add(A,B,R),A);
```

☆ 実際には複数のゴールが一つの変数を待ってサスペンドする可能性があるので，このポインタはサスペンドしたゴールのリストをさすようになっている

```
  (isvar(B)--> suspend(add(A,B,R),B);
   compute(+,A,B,R)).
```

のようになる．ここで，isvar は変数の値が決定しているかどうかの判定することを表し，suspend(Goal,Var) は，変数 Var に対して Goal をサスペンドすることを表す．このコードは,「A または B が決定していなければサスペンドし，そうでなければ計算を行う」というコードであるため，元のプログラムと同じ意味である．

このように，同期部分をボディー部に移動してしまえば，どのようなゴールでもインライン展開が可能になる．先ほどの例

```
foo(A,R):- add(A,1,B), mul(B,2,R).
```

だと次のようになる．

```
foo(A,R):-
  (isvar(A)--> suspend(add(A,1,B),A);
   compute(+,A,1,B)),
  (isvar(B)--> suspend(mul(B,2,R),B);
   compute(*,B,2,R)).
```

これにより，foo(A,R) を呼び出す時点で A が決定していれば，ゴールフレームを作らずに計算を行うことができる．

この手法は複雑なデータフロー解析を行わなくてもよいという利点があるが，同期用の分岐命令（isvar）は削除することができない，値が決まっていなかった場合はオーバヘッドを削減できないという問題点がある．また，むやみに展開するとコード量が爆発する可能性がある．

6. 強制インライン展開の実装

前章で述べたような強制インライン展開を行うプログラムを実装した．プログラムは Fleng 自身で記述した，約 3,300 行程度のプログラムである．Fleng プログラムを入力とし，展開後の Fleng プログラムを出力する．ゴール融合のプログラムが約 6,000 行程度の Fleng プログラムであったことを考えると，ゴール融合より実装が容易であったことが分かる．

また，コンパイラを拡張して，suspend(Goal,Var) の機能を実装し，ボディー部でゴールのサスペンドが出来るようにした．

プログラムは，トップゴールを与え，そこから呼び出し木を深さ優先でたどり，葉の部分から順に展開を行う．展開する際，任意のゴールを展開の対象にできるため，コード量を爆発させないよう，何らかの形で展開するゴールを抑制する必要がある．これは次のように行った．

- 展開しようとしている述語のサイズを推定し，ある一定以上の大きさなら展開しない．
  もっとも単純な戦略である．サイズは compute ならいくつ，ゴールの起動ならいくつ，というように決めているが，後段のコンパイラに負担をかけないよう，行数でも評価している．
- 展開しようとしている述語が再帰呼び出しを行っている場合は展開しない．
  これは，ループを繰り返しの 1 回分だけ展開することになるのであまり効果がないと考え行わない．
  たとえば，

  ```
  foo(...):- bar(...), baz(...).
  bar(...):- ..., bar(...).
  ```

  のような場合，bar は再帰呼び出しを行っているので，foo のボディー中の bar は展開しない．
- 展開しようとしている述語が自分自身の場合，基本的には展開しない．
  ただし，これを展開することはループアンローリングに相当し，効果がある可能性があるので，オプションで指定できるようにした．たとえば，

  ```
  append([A|B],C,D):-
    D = [A|E], append(B,C,E).
  append([],C,D):- D = C.
  ```

  の場合，

  ```
  append([A|B],C,D) :-
   D = [A|E],
   (isvar(B)--> suspend(append(B,C,E),B)
   ;(islist(B)-->
     compute(car,B,F), compute(cdr,B,G),
     E = [F|H], append(G,C,H)
    ;C = E)).
  ```

  のようになる．ただし，今回の評価ではほとんど効果が見られなかったので，このような展開は行っていない．
- 展開しようとしている述語が必ずサスペンドすることが分かる場合は展開しない．これはたとえば，

  ```
  foo:- bar(X), baz(X).
  baz(#X):- ...
  ```

  のような場合，foo において baz の引数 X は bar(X) で初出である．したがって，bar の実行が終了しないかぎり，X の値が定まることはない．このような場合（bar が展開されない限り），baz は展開しない．

7. 2 つの粒度制御手法の評価

7.1 評価条件

これら，2 つの粒度制御手法を並列推論エンジン

PIE64[1]上で評価した．PIE64 は Fleng の高速実行を目的に設計された並列計算機であり，64 台の要素プロセッサを持つ分散共有メモリマシンである．プロセッサは RISC アーキテクチャであり，タグサポートや幾つかの特殊命令以外は一般の RISC プロセッサと同様の命令セットを持つ．

また，相互結合網は自動負荷分散の機能を持ち，負荷値が最小のプロセッサに負荷を割り振ることができるため，評価において，負荷分散の影響は無視できる．

評価対象のプログラムとしては，以下のものを用いた．

primes エラトステネスのふるいを用いた素数生成プログラム．5000 までの素数を求めた．36 行のプログラムである．

queens N-Queens 問題を解くプログラム．10-Queens で評価した．大きさは 41 行である．

fme Fleng プログラムのマクロを展開するプログラム．約 1000 行の実用規模のプログラムである．入力には fme 自身を用いた．

fc Fleng コンパイラ．入力にはマクロ展開後の Queens を用いた．約 3800 行であり，10 ファイルに分割されている．

fme と fc については，I/O 時間を省くため，出力を行わない形で評価を行った．

また，1章でも述べたように，本来は粒度を大きくすれば並列度が低下するため，粒度と並列度はトレードオフの関係にある．しかし，Fleng の場合はこれらの手法を用いて出来るだけ粒度を大きくしても，まだオーバヘッドの方が大きいため，粒度はできるだけ大きくして評価した．

また，fme，fc のような大きなプログラムでは，以下のような制限のもとで粒度制御を行った．

まず，ゴール融合では，定義節内の変数の数が 20 を越える場合は解析を行わないようにした．これは，定義節内の変数の数が増えると依存解析のための情報が増大し，メモリ不足で処理できないためである．また，強制インライン展開では，大きさが 100 行を越える述語は展開しないようにした（述語のサイズは行数のみで評価した）．これは，展開後のプログラムのサイズが増大し，メモリ不足でコンパイルできなくなるのを防ぐためである．

これらの制限の影響を見るため，制限の値による影響を**表 1** と**表 2** に示す．

実行時間の測定値は 3 回実行した値の平均値を用いた（ガーベジコレクションの時間を含む）．コンパイル時間は Sun Enterprise 3000 上の Fleng 処理系で CPU を 1 台だけ用い，測定した．速度は粒度制御を行わない場合を 1 とした相対値，コンパイル時間の単位は秒，バイナリサイズの単位はキロバイトである．

表 1 ゴール融合における制限の影響
Table 1 Effect of limitation (goalfusion).

	節内変数の上限数	17	18	19	20
fc	相対速度（1 台）	1.30	1.29	1.32	1.30
	相対速度（64 台）	1.35	1.35	1.34	1.34
	コンパイル時間	1422	1630	1759	1969
	バイナリサイズ	896	897	898	900
fme	相対速度（1 台）	1.36	1.36	1.37	1.35
	相対速度（64 台）	1.03	1.05	1.04	1.05
	コンパイル時間	368	391	414	571
	バイナリサイズ	201	201	201	200

両手法とも実行速度に対する影響はほとんど見られない．それぞれの制限値による速度差は誤差の範囲内である．これは，実行速度に大きく影響するのは計算木の葉の部分であり，そのような部分は比較的小規模なので，制限が厳しくても融合，展開が行われたからではないかと推測される．ただし，制限が大幅に緩和された場合に，実行速度に影響が出てくる可能性はある．

ゴール融合では，制限を厳しくすることによってコンパイル時間がかなり短くなっている．これは，制限を厳しくすることで，データフロー解析をしない部分が増え，解析時間が短くなったためである．

強制インライン展開では，制限値によってコンパイル時間，バイナリサイズにそれぞれ影響が出ている．これは展開が抑制されることによりプログラムサイズが減少し，それが後段のコンパイラにおけるコンパイル時間にも影響しているためである．

以上のような影響があるが，以降の評価では，最大限に粒度を大きくした場合の評価を行うため，ゴール融合における定義節内の変数の数は 20 個を上限に，強制インライン展開における展開する述語の行数は 100 行を上限とした．これが現在の処理系での限界値である．他の小規模なプログラムはこれらの制限は影響しないため，このような制限はかけていない．

7.2 実行速度

それぞれのプログラムの実行速度を**図 2**〜**図 5** に示す☆．実行時間の測定値は 3 回実行した値の平均値を用いた（ガーベジコレクションの時間を含む）．

それぞれのグラフの縦軸は粒度最適化を行わない場合の 1 台の速度を 1 とした相対速度である．横軸はプ

☆ グラフは文献 7) と若干異なるが，これは測定条件の違いと，バックエンドのコンパイラのバージョンの違いによる．

表 2　強制インライン展開における制限の影響
Table 2　Effect of limitation (force inlining).

	述語の上限行数	70	80	90	100
fc	相対速度（1 台）	1.68	1.67	1.65	1.67
	相対速度（64 台）	1.70	1.70	1.70	1.70
	コンパイル時間	1732	1775	1840	1965
	バイナリサイズ	1495	1519	1545	1584
fme	相対速度（1 台）	1.44	1.44	1.44	1.44
	相対速度（64 台）	1.08	1.07	1.07	1.07
	コンパイル時間	468	471	471	497
	バイナリサイズ	348	348	350	354

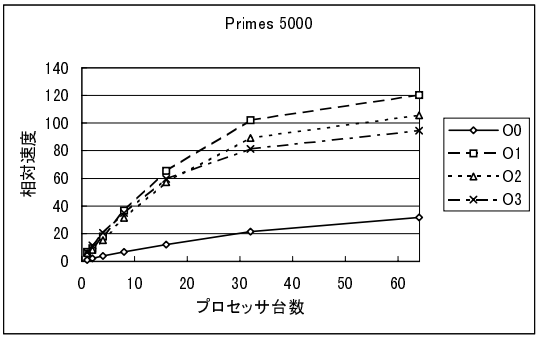


図 2　primes 5000 の実行速度
Fig. 2　Execution speed of primes 5000.

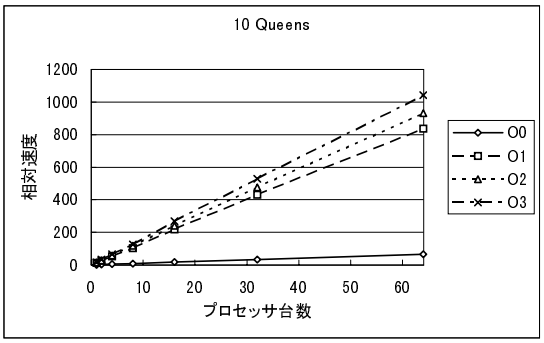


図 3　10 Queens の実行速度
Fig. 3　Execution speed of 10 Queens.

ロセッサ台数を表す．

粒度制御を行わない場合を O0，強制インライン展開を行った場合を O1，ゴール融合を行った場合を O2，ゴール融合後に強制インライン展開を行った場合を O3 として示す．これは以後の評価でも共通である．

fme で若干効果が低いことを除けば，どのプログラムでも粒度制御の効果が見られている．

粒度は出来るだけ大きくして評価したが，台数を増やした際にオリジナルの方が速くなるということはない．これにより，並列度の低下による速度低下はないということが分かる．

queens では，O2 の方が O1 よりも速い．これは強

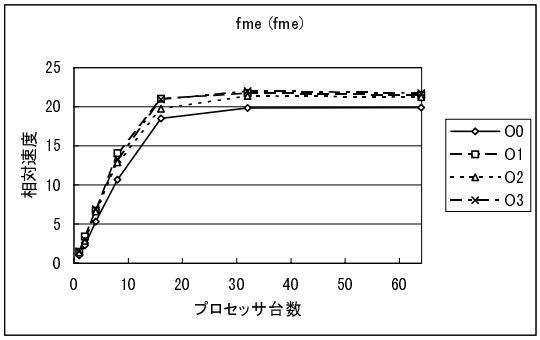


図 4　fme の実行速度
Fig. 4　Execution speed of fme.

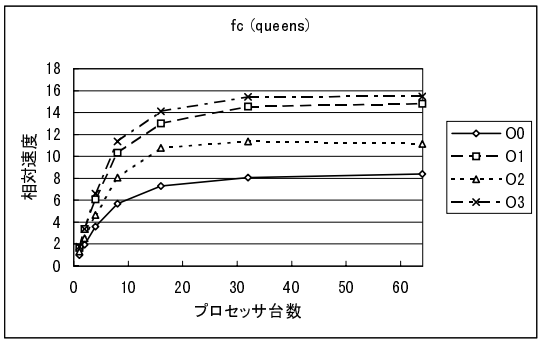


図 5　fc (Fleng compiler) の実行速度
Fig. 5　Execution speed of fc (Fleng compiler).

制インライン展開で取り切れなかった条件分岐が影響しているものと思われる．しかし，その差はあまり大きなものではない．その他のプログラムでは O1 は O2 よりも速い．これはゴール融合では融合しきれなかった部分を強制インライン展開で拾えているのだと考えられる．primes 以外は O3 がもっとも速い．両手法を併用することによる効果があることが分かる．primes で O3 の速度が若干低下しているのはスケジューリングが悪くなったためである．

O3 では primes プログラム中の

```
sift([P|Xs], R) :-
 R = [P|Zs1],
 filter(P,Xs,Ys),
 sift(Ys,Zs1).
```

という部分で，filter 部がインライン展開されて以下のようになっている．

```
sift([P|Xs], R) :-
 R =  [P|Zs1],
 (isvar(Xs)--> ... ;
  ... ,
  subgoal_of_filter(...,Ys,...),
 ),
```

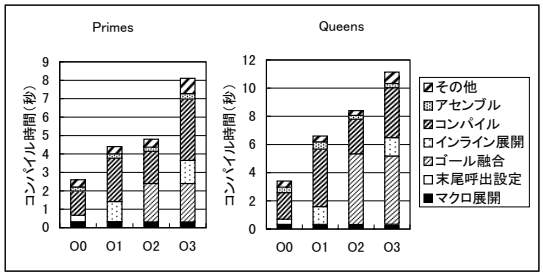


図 6　primes, queens のコンパイル時間
Fig. 6　Compilation time of primes and queens.

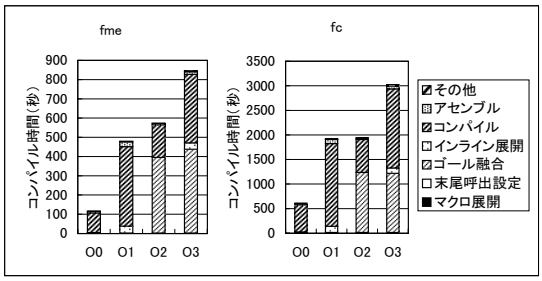


図 7　fme, fc のコンパイル時間
Fig. 7　Compilation time of fme and fc.

```
sift(Ys,Zs1).
```

ここで，展開されない場合は末尾呼び出しの最適化を filter に対して行っている．これは，プログラマはデータフロー順にプログラムを書く傾向があるので，filter を先に実行した方がサスペンドがおきずに高速なためである．ところが，展開後は filter が展開されているので，展開された中の subgoal_of_filter を末尾呼び出しするわけにはいかず，sift の方を末尾呼び出ししている☆．sift は subgoal_of_filter で決定される値を待つので，ここで必ずサスペンドが起こってしまう．これが性能低下の原因である．実際にこの部分の展開を手で抑制すると，O1 同様の高い性能がえられた．このような場合は他のプログラムでも起こりうるが，primes のような小さなプログラムでは，一部分のスケジューリングの悪さが性能に大きな影響を与えうる．

ただ，このような悪いスケジューリングがおこっているかどうかはサスペンドした回数だけを見ても分からない．実際，primes の場合は O3 のサスペンド回数がもっとも少ない．サスペンドが実行時間のクリティカルパスに乗っているかどうかが実行時間に影響するためである．暇なプロセッサがいくらサスペンドしていても，実行時間には影響しない．

7.3　コンパイル時間

図 6 と**図 7** に primes，queens，fme，fc のそれぞれのコンパイルにかかった時間を示す．コンパイル時間は Sun Enterprise 3000 上の Fleng 処理系で CPU を 1 台だけ用い，測定した．

強制インライン展開の方がゴール融合よりもコンパイル時間は短くすんでいる．しかし，圧倒的というほどでもないのは，強制インライン展開では変換後のプログラムの行数が増えるため，後段のコンパイラに負担がかかるためである．fme や fc のような大きなプログラムだと，この傾向がより顕著に現れている．図 7 を見ると，強制インライン展開にかかっている時間はゴール融合にかかっている時間に比べると圧倒的に短いが，後段のコンパイラにかかっている時間は強制インライン展開の方が長い．

また，コンパイル時間は表 1，表 2 に示したように，粒度制御における制限にも依存する．制限の値によっては，ゴール融合と強制インライン展開のコンパイル時間が逆転することもありうる．

強制インライン展開とゴール融合を共に行った場合は，やはりもっとも長い時間がかかっている．

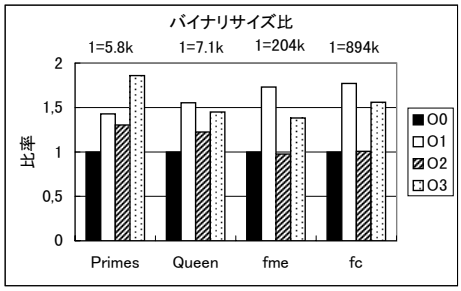


図 8　バイナリサイズ
Fig. 8　Binary size.

7.4　バイナリサイズ

図 8 に各プログラムの最適化によるバイナリサイズの比を表す．最適化を行わない場合のバイナリサイズを 1 とする（最適化を行わない場合の実際のバイナリサイズは，グラフの上に示している）．

バイナリサイズはゴール融合の方が強制インライン展開よりも小さくなる傾向がある．これは，ゴール融合の場合，融合によって参照されなくなった定義を削除するためである．強制インライン展開では，サスペンドに備えて元の定義を残しておく必要があるので，バイナリサイズは必ず増加する．

ゴール融合の場合，単純な融合だけが行われるとすると，直観的にはバイナリサイズはオリジナルよりも

☆ この場合は工夫しだいで subgoal_of_filter の方を末尾呼び出しすることは可能だが，sift の方も展開されてしまうと，かなり困難になる．

減少するように思われるが，実際にはそうではない．これは，たとえば再帰する述語どうしが融合されることによって繰り返しが展開され，初期値の与えられた1回目とそうでない2回目以降で異なる融合が行われたりすることがあるためである．Primes の O3 でバイナリサイズが O1 よりも大きくなっているのは，このような複雑な融合が行われた後に強制インライン展開が行われたためである．

今回の評価では，強制インライン展開の際，展開する述語の行数は100行を上限とした．バイナリサイズはこの制限に依存するが，今回の制限に関する限り，バイナリサイズの増加は2倍以内に収まっており，許容範囲内であると考えられる．

7.5 考　　察

強制インライン展開は複雑なデータフロー解析を行わないのにもかかわらず，多くの場合で十分な速度向上を示していることが注目される．バイナリサイズは若干大きめになる傾向があるが，これが問題にならない限り，粒度制御手法としては十分な効力があると考えられる．

単一代入変数を用いてデータフロー同期をとるという実行機構を持つ言語は，Fleng 以外にもたくさんある．Fleng のように全てがそうであるという言語は多くないが，たとえば他の Committed-Choice 型言語（KL1 など）やデータフロー関数型言語（Id など）がそうであるし，一部にこのような機構を採り入れた言語としては future を使った Lisp やオブジェクト指向言語，Compositional C++ などがある．これらの言語でも同様の問題が指摘されているが，これらの言語に今回紹介したような粒度制御手法を適用することを考えると，実装の容易さからいって強制インライン展開が有利であると考えられる．

もちろん，ゴール融合と強制インライン展開を併用することで，さらなる効率の向上が見られているため，可能であれば併用することが望ましいであろう．

8. 関 連 研 究

データフロー関数型言語の分野でも，このような研究はさかんに行われている．データフロー関数型言語の一種 Id では，データフローグラフの複数のノードを一つのスレッドにまとめることでオーバヘッドを削減する研究が行われている．Separation Constraint Partitioning[4] と呼ばれるアルゴリズムが有名であり，本研究のゴール融合も大きな影響を受けている．

また，日下部らの研究[9] では V のような非ストリクトなデータフロー言語の実装において suspensive という概念を提案している．この実行機構では，関数を実行する際，本来はサスペンドに備えてヒープ上にフレームを作成しなければならないが，値が決まってると仮定して，スタック上にフレームを作成して実行を行う．もし値が決まっていなくてサスペンドしなければならない場合は，その時点でヒープ上にフレームを作成する．インライン展開ではなく，スタックフレームを用いた関数実行であるという違いはあるが，我々の強制インライン展開に近い考え方である．Separation Constraint Partitioning との併用も考えられている．しかし，文献 9) では大きなプログラムでの評価はまだ紹介されていない．

また，大山らは future を使った並列オブジェクト指向言語 Schematic において，強制インライン展開と同様の手法を独立に提案している[3]．彼らの評価でも，このような手法が効果的であることが示されている．

Committed-Choice 型言語における研究としては，KLIC における伊川らによる研究がある[5]．あるゴール b があるゴール a に依存している場合，ゴール a がサスペンドすると，ゴール b をスケジューリングしても必ずサスペンドしてしまう．この研究ではこれを防ぐため，依存関係のあるゴールを一つのスレッドという単位にまとめ，スレッドを単位にしてスケジューリングを行う．この場合，ゴール a とゴール b は一つのスレッドにまとめられるので，a がサスペンドすると，b がスケジューリングされることはない．この手法では，ゴール a が実行可能になった場合でもゴール b が実行可能であるとは保証されないので，ゴール融合よりもゆるい結合である．また，この研究ではサスペンドコストの低減を目的としているという点で，我々の研究とは異なるが，プログラムの逐次化によって効率向上を目指しているという点は共通している．

また，強制インライン展開の考え方は，オブジェクト指向言語における，メソッド呼び出しの最適化[8] との類似性があると思われる．この手法では，メソッド呼び出しの最適化を行う際に，レシーバクラスを予測して，インライン展開を可能にする．たとえば，`r.draw()` というコードにおいて，r のクラスがほとんどの場合 Box であると予測できるならば，以下のようにコンパイルできる（記法は C++に従う）

```
if(r.class == Box)
  Box::draw(); // インライン展開できる
else
  r.draw();
```

強制インライン展開の場合は，値が決定していると予測しているわけだが，ディスパッチと関数呼び出しを

削除している点で類似性があると考えられる．

9. おわりに

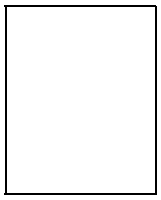
本研究では，Committed-Choice 型言語 Fleng において，細粒度実行のオーバヘッドを削除する 2 つの手法，ゴール融合と強制インライン展開について，強制インライン展開の実装を行った上で，比較評価と検討を行った．強制インライン展開は，容易に実装できるが，かなりの程度のオーバヘッドを削減できることが分かった．また両手法を併用することにより，さらなる速度向上がえられることを示した．

本研究は，PIE64 上での評価のみを示した．今後の課題として，ワークステーション上の Fleng 処理系の改善を行った上で，マルチプロセッサワークステーション上で評価を行い，C などの言語と Fleng 処理系の比較を行うことがあげられる．

謝辞 有益な御意見を頂いた査読者の方に感謝致します．

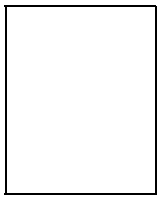
参考文献

1) Araki, T., Hidaka, Y., Nakada, H., Koike, H. and Tanaka, H.: System Integration of the Parallel Inference Engine PIE64, *Workshop on Parallel Logic Programming attached to International Symposium on Fifth Generation Computer Systems 1994*, pp. 64–76 (1994).
2) Nilsson, M. and Tanaka, H.: Fleng Prolog - the language which turns supercomputers into Prolog machines, *Logic Programming '86, LNCS 264*, Springer-Verlag, pp. 170–179 (1989).
3) Oyama, Y., Taura, K. and Yonezawa, A.: An Efficient Compilation Framework for Language Based on a Concurrent Process Calculus, *Euro-Par'97 Parallel Processing, LNCS 1300*, Springer-Verlag, pp. 546–553 (1997).
4) Schauser, K. E., Culler, D. E. and Goldstein, S. C.: Separation Constraint Partitioning – A New Algorithm for Partitioning Non-strict Programs into Sequential Threads, *POPL '95*, ACM, pp. 259–271 (1995).
5) 伊川雅彦, 大野和彦, 五島正裕, 森眞一郎, 中島浩, 富田眞治: KLIC におけるゴール・スケジューリング最適化, 情報処理学会研究報告 プログラミング研究会, 96-PRO-10, Vol. 96, No. 107, pp. 43–48 (1996).
6) 荒木拓也, 田中英彦: Committed-Choice 型言語 Fleng のインライン展開による粒度制御手法, 情報処理学会第 53 回全国大会, Vol. 1, No. 3E-8, pp. 347–348 (1996).
7) 荒木拓也, 田中英彦: Committed-Choice 型言語 Fleng における静的粒度最適化, 情報処理学会論文誌, Vol. 38, No. 9, pp. 1771–1780 (1997).
8) 小野寺民也: オブジェクト指向言語におけるメッセージ送信の高速化技 法, 情報処理学会誌, Vol.38, No. 4, pp. 301–310 (1997).
9) 日下部茂, 森本徹雄, 雨宮真人: 非ストリクトデータフロープログラム実行におけるスタッ クフレームの利用, 情報処理学会研究報告 プログラミング研究会, 97-PRO-14, Vol. 97, No. 78, pp. 73–78 (1997).



荒木 拓也（学生会員）

1971 年生．1994 年東京大学工学部電気工学科卒業．1996 年同大学院情報工学専攻修士課程修了．現在，同大学院博士課程に在学中．並列プログラミング言語の最適化の研究に従事．並列論理型言語，並列関数型言語，並列オブジェクト指向言語，並列化コンパイラなどに興味を持つ．



田中 英彦（正会員）

1943 年生．1965 年東京大学工学部電子工学科卒業．1970 年同大学院博士課程終了．工学博士．同年東京大学工学部講師．1971 年助教授，1978 年〜1979 年ニューヨーク市立大学客員教授，1987 年教授現在に至る．計算機アーキテクチャ，並列処理，人工知能，自然言語処理，分散処理，CAD 等に興味を持っている．「非ノイマンコンピュータ」,「情報通信システム」著,「計算機アーキテクチャ」,「VLSI コンピュータ I, II」,「ソフトウエア指向アーキテクチャ」共著，New Generation Computing 編集長，情報処理学会，電子情報通信学会，人工知能学会，ソフトウエア科学会，IEEE，ACM，各会員．