計算機アーキテクチャ 134－26
（1999．8．4）

自動並列化コンパイラの統一的中間表現とインターフェースを用いたコード変換の実装

羽田 昌代 * 神戸 和子 * 中西 恒夫 † 城 和貴 *

masayo1@ics.nara-wu.ac.jp

* 奈良女子大学理学部情報科学科
† 奈良先端科学技術大学院大学情報科学研究科

概要

自動並列化コンパイラで逐次プログラムを並列化する時の手法としてコード変換がある．従来のコード変換はそれぞれの対応する中間表現上で実装が行われており適用順序が任意ではない等の不都合があった．これらの中間表現を統一し，その一つの中間表現に対して変換を行うことが提案されている．本稿では，この統一的中間表現とそのインターフェースを用いて，コード変換手法の一つであるインライン展開を実装し，その結果を検討し統一的中間表現を用いることの意義について議論を行う．

An Implementation of a Code Transformation with the UIR and Interface of a Parallelizing Compiler

Masayo Haneda * Kazuko Kambe * Tsuneo Nakanishi † Kazuki Joe *

* Fuculty of Sciences. Nara Women's University
† Nara Institute of Science and Technology

Abstract

Code transformations are used to parallelize sequential programs for parallelizing compilers. Conventional parallelizing compilers do not allow the code transformations, which are implemented on individual intermediate representations, to be applied in arbitrary order. The idea of universal intermediate representations by which various intermediate representations are integrated in a unified way has been proposed. In this paper, we implement an inline expansion with using a UIR and its interface, and discuss the implementation result for future UIRs.

1 はじめに

自動並列化コンパイラは逐次プログラムを並列化するために様々な解析と変換を行う．解析によって得られた情報はコード変換を実装するのに適した形である中間表現に格納される．一般的に中間表現には様々な形があり，各変換はそれぞれが必要とする中間表現に対してのみ操作が行われる．従来の自動並列化コンパイラはこのような中間表現を複数保持している．この構造には次の欠点がある．各コード変換手法で異なる中間表現を参照するため，(1) 変換手法を任意の順序で適用できない，(2) 同じ変換手法なのに中間表現の粒度やレベルによって個別に実装しなければならない，(3) 中間表現の変換時に情報損失の可能性が高い．これらの問題を解決するた

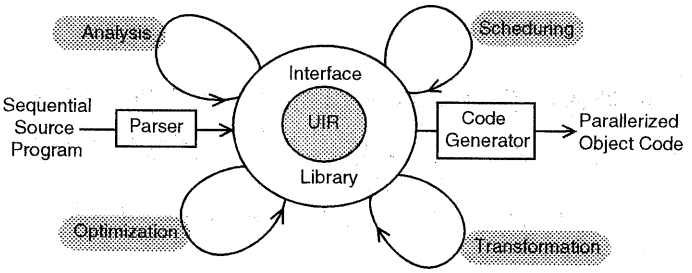

図 1: 統一的中間表現を核とした自動並列化コンパイラ

めには，コンパイラに各コード変換手法が使用する中間表現を統合・共通化した統一的中間表現（UIR : Universal Intermediate Representation)[2] を持たせ，これに対して全てのコード変換手法を任意順序で適用させることが必要である．自動並列化コンパイラである SUIF[4], PROMIS[8] などがこの UIR を実装しており，今日では中間表現の実装形態の主流になっていると言える．

奈良女子大学，奈良先端技術大学院大学，和歌山大学が共同開発している Narafrase[7] は，この UIR を用いた分散メモリ型並列処理システムのための自動並列化コンパイラである．Narafrase では UIR としてデータ分割グラフ DPG(Data Partitioning Graph)[6] を採用している．

UIR には変換に必要な全ての情報が格納されているため，情報の一貫性を保ちながら変換を行うことは容易である．反面，UIR を操作するためにはその複雑な構造を理解しなくてはいけない．この操作をより簡単にするために，インターフェースライブラリを整備する必要がある．UIR そのものは各コンパイラにおいて自由に設計し実装することが可能であるが，インターフェースの仕様はコンパイラ間で共通にすることが望ましい．インターフェースを共通にすることにより，共通のプログラムでどのような UIR でも操作することができる．本研究グループではこのような汎用性の高いインターフェースライブラリの設計および実装を行っている．

現在 Narafrase ではループ変換手法である Loop Distribution および Loop Permutation が実装されている [7]．本稿ではコード変換手法の一つであるインライン展開の実装に関する報告を行う．

以下 2 章では UIR とインターフェイスについて，3 章ではインライン展開の概要と手法について説明する．4 章では実装した結果と他のコンパイラでのインライン展開との比較と分析，5 章ではまとめとこれからの課題について述べる．

2 UIR とインターフェース

2.1 UIR

UIR はプログラムの並列化，最適化のために必要な中間表現を統合・共通化したものである．図 1 は UIR を核とした自動並列化コンパイラの概念図を示す．Narafrase では次の中間表現の情報を使うが，全ての情報は UIR としての DPG に保持される．

- シンボルテーブル
- 抽象構文木 (AST)
- コールグラフ
- データ依存グラフ

DPG は分散メモリ型並列システムで並列化を行うために，HTG を拡張して変数参照の情報を明示的に

示している中間表現である．HTG[1] は Parafrase-2 ならびに PROMIS で UIR として用いられているプログラムのループネストに基づいて階層化されたグラフである．Narafrase では最終的に DPG を実装する予定であるが，本稿では Girker らの HTG を拡張したものを使用している．

2.2 インターフェース

先にも述べたように，インターフェースがあれば，ユーザは UIR の詳しい構造や，取り扱い方を知らなくても，プログラムの変換手法を実装することができる．また，インターフェースは UIR の情報が欠落しないようにその操作を管理する役割がある．Narafrase では，C++言語を使用し，UIR に含まれる中間表現および情報と，インターフェースとなる関数をクラスとして一体化して実装している．これによって次の利点が得られる．

1. UIR はそのインターフェースライブラリを通してのみ操作されるため整合性を保って更新が行われる．
2. UIR の仕様が変更されても，インターフェイスを修正するだけでよい．そのため UIR の保守と拡張が容易である．
3. 複雑なコンパイラ内部のデータ構造が取り扱いやすくなる．

3 インライン展開

インライン展開とは，プログラムコード中で関数呼び出しを行っている部分を関数本体で置き換えるコード変換手法である．関数呼び出しの部分の並列化を行う時に，呼び出す関数と呼び出される関数の間の関係を解析する必要がある．この場合複雑なプロシージャ間解析が必要になる．プロシージャ間解析は 2 つの関数間の変数の依存関係の解析などを行う必要があるため，解析に時間がかかる．それに比べてインライン展開は，呼び出される関数を呼び出す関数の中で展開するため，1 つのプログラムの解析をするだけでよい．また関数呼び出しのオーバーヘッドがなくなる．本節ではインライン展開の UIR 上への実装について述べる．

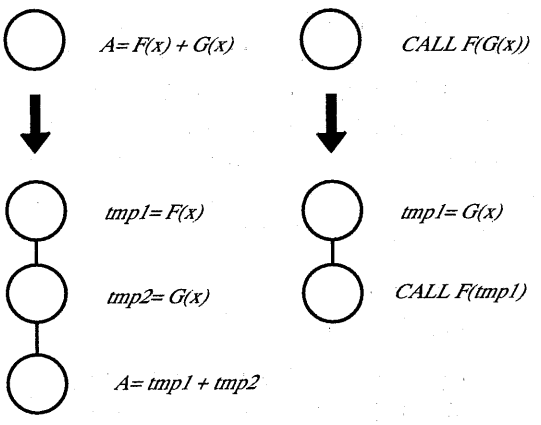

図 2: 複数の関数に対するインライン展開の処理法

3.1 HTG における実装方法

一般的な展開の手順は次の通りである．

1. 関数呼び出しの部分を探す．
2. 呼び出されている関数をコピーする．
3. コピーされた関数の中で，変数のリネームや戻り値の処理をする．
4. コピーと関数呼び出しの部分を置き換える．

議論を簡潔にするために本稿では再帰呼出の展開については考えない．再帰呼出の展開はスタックを使用すれば可能であり，これによって議論の一般性を欠くことはない．

3.2 実装の手順

ここではインライン展開における HTG の操作の手順について述べる．

1. コールグラフをメイン関数からトップダウンに

調べ展開する関数を決定する．

2. 展開する関数内で，関数呼び出しのHTGノードが2つ以上関数を呼び出しているならば，ノード内の呼び出しが1つになるようなノードの分割を行う．これは全てのHTGノードについて行う．分割を行わなければならない例として次のようなものが挙げられる．

   - A=F(x)+G(x)
   - CALL F(G(x))

   上の2つの例はどちらも1つのHTGノードが2つの関数を呼び出している．1つ目はF(x), G(x)の結果をそれぞれ新しい変数の中にいれてから，それらの変数を元の式に代入して計算をさせる．2つめは同じようにG (x)の結果を新しい変数に入れてから元の式に代入するという方法が考えられる．このようにノード内のコードを関数呼び出しが1つずつになるようなコードに分割し，それに従ってノードを分割する．(図2参照)
3. 展開するコールグラフのノードから出るエッジを調べ、展開される関数を決定する．ただしコールグラフの強連結成分は再帰呼び出しにあたるため，展開しない．
4. 展開される関数のHTGを全てコピーする．
5. コピーしたものと呼び出し側の環境を展開に適した形に変える．コピーしたものの中では次の作業を行なう．
   (a) 呼び出される関数の仮引数を調べて対応した実引数と置き換える．
   (b) その関数内で行われる代入文の変数名が元の関数の変数名と重複しないように，リネームする．HTGは色々なタイプのノードを持つが，代入文は式タイプのノードで表される，したがってその関数内の式ノードを全て調べて，大域変数以外の変数を新しい変数名で置き換えるとよい．
   (c) HTGを変更する時にデータ依存の情報も変更する必要がある．関数の引数については仮引数と実引数の対応を取ればよいが，大域変数が関数内に含まれる時にはデータ依存の情報がかわるために，大域変数かどうかを調べなくてはならない．UIRは大域変数の情報を持っているので，このデータを参照すると効率よく調べることができる．
6. 呼び出し側のHTGノードをコピーしたHTGで置き換える．
7. 展開が終ったコールグラフのエッジを消去する．

4 結果の検討

3節で述べたような実装を行い，以下に示すサンプルプログラムに対してインライン展開を適用した．

```
program test
integer i,j,k,l
integer a(10001)
do l=1,10000
  call add(a(l+1)),a(l),k)
enddo
end

subroutine add(i,j,k)
if(i .GE. 1000) then
  k=i*j
  return
else
  k=i/j
endif
return
end
```

図3は，その結果のHTGの変化の様子を示している．CALLによって呼び出された関数のHTGの構造がそのまま付け加えられていることが分かる。このような構造に変えることにより，制御フローや変数の振るまいが明確になり，更なるコード変換や依存解析が可能になる．ただし，今回の実装ではHTGの構造を変えることはできるが，データ依存などの情報を変更することができない．

従来の自動並列化コンパイラでは，ASTの段階でインライン展開を行っているために，他のプログラム情報を考慮する必要がなかった．UIR利用したインライン展開を実装する場合には，構造を変えた時に他の情報も一貫性を保つように操作をしなくてはならない．展開後の処理の一つとしてデータ依存解

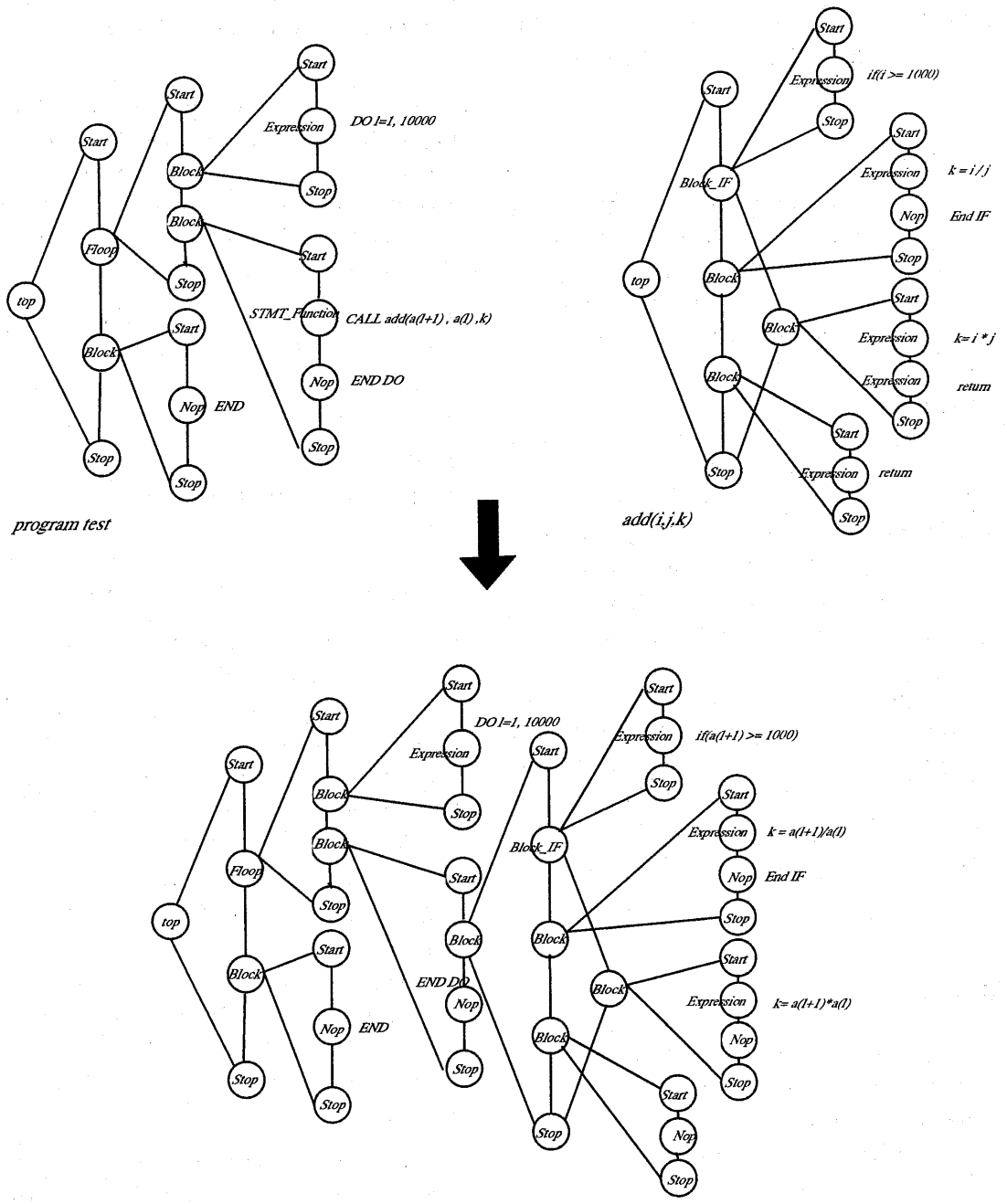

図 3: インライン展開による HTG の変化

析が必要であるが，一度の展開ごとにデータ依存解析を行うことはオーバーヘッドが大きい．従って部分的なデータ依存解析を行うことが望ましい．

5 まとめ

本稿では自動並列化コンパイラの統一的中間表現インターフェースを用いて，強力なコード変換手法の1つであるインライン展開を実装した．従来のコンパイラにおいて，インライン展開はコンパイルの初期段階でのみ適用されてきたが，UIRを核とした構造をコンパイラに持たせることで任意の順序で適用可能となった．また実装にインターフェースライブラリを使用したことで移植性も得ることができた．今回はDPGの実装が不十分であるためDPGに対する実装を行うことができなかったが，インターフェースの仕様を決めているため，関係なく実装を行うことができた．また再帰呼び出しを扱わないインライン展開はデータ依存や制御フローの構造がそれほど複雑ではないため少ないインターフェイスで実装することができたが，より構造の複雑な再帰呼び出しの展開などの実装を行う時には，より多くのインターフェースライブラリが必要となると考えられる．よって今後の課題としてインターフェースライブラリの整備を進めていくことが挙げられる．また，インライン展開後の部分的データ依存解析・データフロー解析法を開発することも今後の最重要課題の一つである．

参考文献

[1] M. Girkar and C.D. Polychronopoulos. "The Hierarchical Task Graph as a Universal Intermediate Representation", *Internal Journal of Parallel Programing*, 199522(5):519-551, October1994.

[2] C.J. Brownhill, A. Nicolau, S. Novack and C.D. Polychronopoulos, "Achieving Multi-level Parallelization",*Lecture Notes in Compute Science,Programmability*,1336 Springer:183-194, November 1997.

[3] C.D. Polychronopoulos, M.B. Girkar, M.R. Haghihat, C.L. Lee, B.P. Lee, B.P. Leung and D.A. Schouten, "The Structure of Parafrase-2:An advanced parallelizing compier for C and Fortran", *In Proc, Internat.Conf. on Parallel Processing*, 472-492, 1989.

[4] Robert Wilson et al, "An infrastructure for reserch on parallelizing and optimizing compilers", *Technical Report, Computer System Laboratory, Stanford University.*

[5] Keith A. Faigin, Jay P. Hoeflinger, David A. Padua, Paul M. Petersen and Stephan A. Weatherford, "The Polaris Internal Representation", *International Journal of Parallel Programming*, 22(5):553-286, October1994.

[6] T. Nakanishi, K. Joe, H. Saito, C.D. Polychronopoulos, A. Fukuda and K. Araki, "The Data Partitioning Graph: Extending Data and Control Dependencies for Data Partitioning", *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pp.170-185, August 1994.

[7] K. Kambe, T. Nakanishi, K. Joe, Y. Kunieda, F. Kako. "Implementaion of Loop Transformations with a Universal Intermediate Representation Interface Library", *Proceedings of The International Conference on Parallel and Distributed Proceeding Techniques and And Applications, Vol.IV*, pp.1905-1911, June1999.

[8] H. Saito, N. Stavrakos, S. Caroll, C. Polychronopoulos and A. Nicolau. "The Design of the PROMIS Compiler", *TechnicalReport, UIUCCSRD 1539(revised)*, March 1999.