ハイパフォーマンス
コンピューティング 77−12
(1999. 8. 3)

Cコンパイラにおけるループ最適化の検討

飯塚大介　小沢年弘 †　坂井修一　田中英彦

東京大学工学系研究科

富士通研究所 †

iizuka@mtl.t.u-tokyo.ac.jp

概要

C言語は多くのアプリケーションを記述するのに使用されているため、コンパイラによってC言語で書かれたプログラムを最適化することは重要である。コンパイラの最適化の研究を行なうには、既存のコンパイラを改良する方法やSUIF等の中間言語を用いる手法等がある。本稿では、複数レベルの中間言語を使用するCコンパイラを紹介し、最適化による効果の大きいループアンローリングを本コンパイラの中間言語のレベルで実装し、性能評価を行なったことを報告する。

Examination of loop optimization on C compiler

Daisuke Iizuka　Toshihiro Ozawa †　Shuichi Sakai　Hidehiko Tanaka

Graduate School of Engineering, University of Tokyo

Fujitsu Laboratry †

Abstract

C language is used to implement various application, so it is important to optimize C language program by compiler. To research about compiler optimization, we modify facilities compiler, or use intermediate code such as SUIF. This paper introduces about C compiler which use multiple intermediate code, describes implementing and evaluation of loop unrolling on intermediate code of the compiler .

1 はじめに

近年のスーパースカラやVLIWプロセッサは複数の命令ユニットを持ち、1サイクルに複数命令を同時実行することによって実行速度の向上を計っている。特にMerced等をはじめとしたVLIWプロセッサではスーパースカラプロセッサのように実行時に動的に命令スケジューリングを行なわずに、プログラムのコンパイル時に静的に命令スケジューリングを行なっているため、コンパイラによるスケジューリングが性能向上に大きく関わってくる。そのためプログラムから自動的に並列度を抽出する並列化コンパイラの研究がすすめられている。既存の逐次型言語のうち良く使用されるものとしては、FortranとC言語があげられる。C言語は一般のアプリケーションの記述に広く使用されているため、Cコンパイラの最適化技術は特に重要である。

このようなコンパイラの最適化の研究を行なう際の手法としては、

- 既存のコンパイラを改良する手法
- SUIF等の中間言語を利用して最適化を行なう手法
- 通常のコンパイラを用いてアセンブラに変換し、アセンブラレベルで最適化を行なう手法

等が考えられる。

本稿では、複数のレベルの中間言語を使用することが、コンパイラの最適化の研究に有効であることを示す。実際には、複数のレベルの中間言語を使用するCコンパイラ（以下、newccと称する）を用い、中間言語レベルでの最適化による効果の大きいループアンローリングを実装し、評価を行なった。コンパイラは、富士通研究所で作成したコンパイラをベースに使っている。

2 newcc

2.1 newccの特徴

newccはC言語のコンパイラである。本体はUtiLisp[4]で記述されている。本コンパイラは図1のように、プロセッサに非依存の処理と最適化を行なうfrontendと、プロセッサに依存する処理と最適化を行なうbackendに分けられる。現在、本コンパイラのbackendはSPARC version7用が作成されている。本コンパイラを新しいプロセッサに対応させるには、プロセッサに依存したIICを定義し、プロセッサに依存したbackendを作成することで対応するようになっている。

本コンパイラの中間言語はLispのS式で記述されており、入力されたC言語のソースはPICからIICまでの7つの中間言語を経て、それぞれのレベルでの最適化を行なった後に最後にアセンブラに変換されるようになっている。

本コンパイラはコンパイルの途中で任意のレベルの中間言語をファイルに出力したり、途中から任意の中間言語のファイルを入力してコンパイルを続行することが可能となっている。そのため、外部に最適化ルーチンを作成して各中間言語を処理することで、任意の最適化処理を行なえるようになっている。

2.2 中間言語の構成

本コンパイラの各中間言語の構成は以下のようになっている。

PIC

本コンパイラの中間言語はLispのS式で記述されているため、C言語の構文構造をLispのS式に変換するために設定された中間言語である。構文解析、字句解析を行なった後にS式に変換する。

HIC

PICをコンパイラの内部で扱う中間言語で扱いやすいように変換するために設定された中間言語である。

SIC

構造化されたループの最適化を行なうために設定された中間コードである。この中間コードのレベルでは変数定義情報の収集とループの検出を行ない、中間言語は実行コードと、変数のスコープを記述した記号表、structやunion等で構成された新たな型を記述したタグ表に分割される。

MIC

プログラムを基本ブロックに分割し、条件文等で使用される一時変数の作成や定数の計算、参照メモリの一致等の解析を行なうために設定された中間言語である。ループ構造はここで基本ブロックと条件分岐に変換される。

LIC

プロセッサには依存しないが、言語に依存した最適化のために設定された中間言語である。C言語では、ポインタを型の大きさに合わせてアドレスに変換する。また、数式はこの中間言語の時点で全て3番値コードに変換される。

	go	m88ksim	compress	li	vortex
SC	9.24	6.95	8.60	6.91	7.30
newcc	7.32	4.98	7.12	5.02	5.88
egcs	3.78	2.24	4.80	2.94	3.70
gcc	3.65	2.41	4.60	2.88	3.50

表 1: SPEC95 ベンチマーク結果

RIC

プロセッサおよび言語に依存しない最適化のために設定された中間言語である。基本的な変数はすべて無限大の数の仮想的なレジスタに変換され、配列や struct、union はメモリアクセスに変換される。

IIC

プロセッサに依存した最適化を行なうために設定された中間言語であり、レジスタ割り付けや命令スケジューリング等の処理を行なう。IIC はアセンブラとほぼ 1 対 1 に対応している。

2.3 newcc の性能評価

現在の本コンパイラの性能を評価するため、SPARC システム上で、本コンパイラと、Sun の SC4.2、egcs-release1.1、gcc-2.7.2.3 のそれぞれのコンパイラで SPECint95 ベンチマークプログラムをコンパイルして実行し、性能の比較を行なった。結果を表 1に示す。評価は Sun UltraEnterprise450(CPU UltraSparcII 250Mhz、メモリ 512MB) 上で行なった。なお、Sun UltraEnterprise450 は 4CPU 構成であるが、今回使用したベンチマークプログラムでは 1CPU のみ使用している。

表 1から、本コンパイラは、SC には及ばないものの、gcc や egcs よりも良い性能を出していることがわかる。

3 C 言語におけるループ最適化

ループは並列度を抽出する上で重要な要素であり、ループボディは非常に多く実行されるために並列化による効果が大きい。とくにループアンローリングを行なうことで多くの並列度を抽出することができる。しかし、newcc 上ではループアンローリングを実装していなかったため、ループアンローリングの実装を行なった。

C 言語のループ構造は中間言語の SIC の時点ではそのまま保たれているが、SIC から MIC に変換時点で基本ブロックと条件分岐に分割されてしまう。その

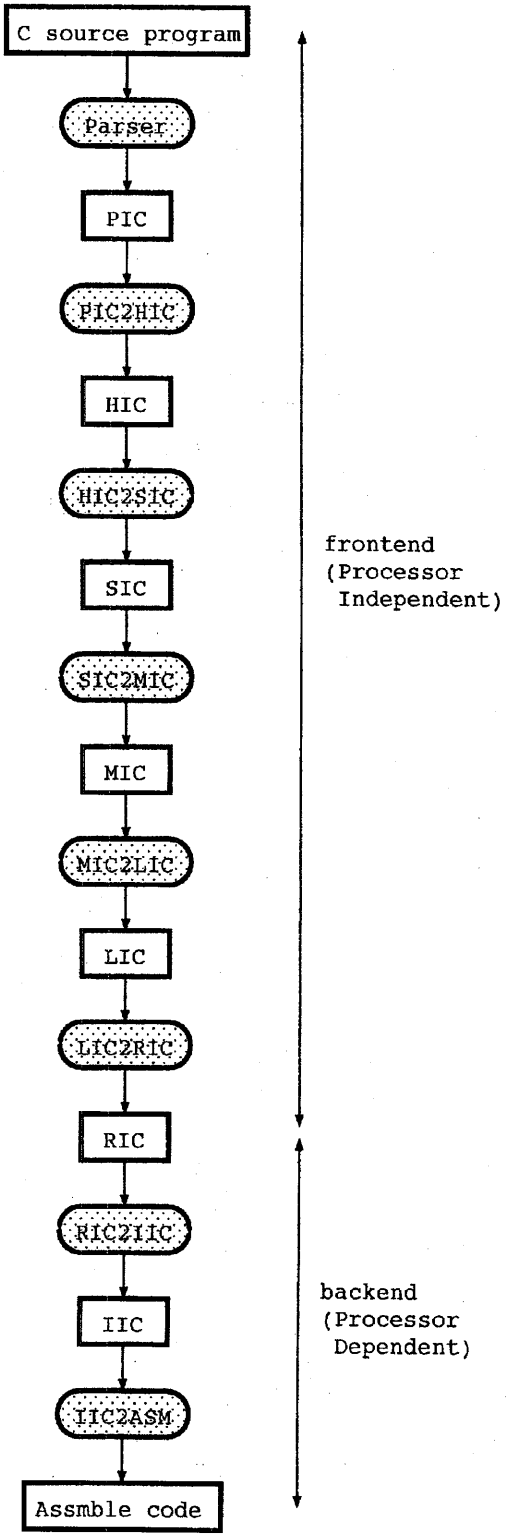

図 1: 中間言語の構成

```
int a[100],b[100],c[100];
void calc(void)
{
    int i;
    for (i = M; i < N; i++)
        a[i] = b[i] + c[i];
}
```

図 2: sample source

ためMIC以下のレベルでループアンローリングを実装するには、まずループ構造を探し、誘導変数とループ実行の継続条件を検索しなければならない。よってSICレベルでループアンローリングの実装を行なった。

図2のようなCで書かれたループをnewccを用いてSICに変換したものを図3に示す。

図3の上半分は記号表であり、変数の型と有効範囲のブロックを示している。下半分が実際のコードとなっている。この図からわかるように、SICではC言語のfor文がほぼそのままの形で残っており、また変数の有効範囲がすぐにわかるため、誘導変数やループの継続条件の解析が容易になっており、ループアンローリングに適した形式になっている。

ループアンローリングは以下のような手法で行なった。

3.1 最内周ループの検索

まずは最内周ループを検索し、誘導変数を見つける。前述のようにSICレベルで行なうことで、これらの検索を容易に行なうことができる。ループの継続条件が単純で、ループ実行中に変化せず、かつループボディー内で関数呼出を行なっていないのであればアンローリングできる可能性がある。

3.2 ループ継続条件の確認

もとのループの継続条件をそのままアンローリングしたループに適用すると、ループを余分に回ってしまう可能性があるので、アンローリングしたループは継続条件を変更する必要がある。

たとえば、誘導変数iが1ずつ増えていくループで、もとのループの継続条件がi < Nで、4回アンローリングを行なった場合、アンローリングしたループの継続条件はi < N - 4としなければならない。このときに、アンダーフローが発生してN - 4がNよりも大きくなってしまうとループが正常に実行されなくなってしまうためにこれをチェックするようにした。

```
;;;;; block0000 ;;;;;
(setq block0000 '
(#(function-entry calc (new global) #'(type void)
 code0000 block0001 (nil)  (nil) block0000)
 #(variable-entry c (global static)
 (array (type int) 100) nil nil nil 101 nil
 nil nil nil block0000)
 #(variable-entry b (global static)
 (array (type int) 100) nil nil nil 101 nil
   nil nil nil block0000)
 #(variable-entry a (global static)
 (array (type int) 100) nil nil nil 101 nil
   nil nil nil block0000)
 nil)
)
;;;;; block0001 ;;;;;
(setq block0001 '
(#(block-entry block0002) block0000)
)
;;;;; block0002 ;;;;;
(setq block0002 '
(#(variable-entry i (local auto) (type int)
 nil nil nil 501 nil nil nil nil
   block0002)
 block0001)
)

;;;;; code0000 (calc) ;;;;;
(setplist 'code0000 '(identifier calc))
(setq code0000 '
(block (note block0002 block0001)
  (LOOP 0
    nil
    nil
    nil
    (for 701
       (= 701 #(var i block0002) (const 0))
       (< 701 #(var i block0002) (const 100))
       (post-inc 701 #(var i block0002))
       (= 801
         (array-ref
            801
            #(var a block0000)
            #(var i block0002))
         (/+ 801
           (array-ref
              801
              #(var b block0000)
              #(var i block0002))
           (array-ref
              801
              #(var c block0000)
              #(var i block0002)))))))
```

図 3: sample source をSIC化したもの

3.3 alias 解析

次に、ループボディー内でポインタを用いてデータにアクセスを行なっている場合、誘導変数や継続条件を変化させてしまったりする可能性があるため、そのポインタがどのメモリ領域を指しているのかを解析する必要がある。

具体的には、誘導変数や継続条件に含まれる変数の alias があるかどうかを調べ、ループ中で alias を使用してこれらの変数に代入を行なっていないかどうかを調べる。

そのため、誘導変数や継続条件、ループ中で使用しているポインタ変数がグローバル変数や関数の引数で渡されたものである場合には alias が存在する可能性があるのでアンローリングはできない。

3.4 アンローリング

以上から、アンローリングして安全であると判断された最内周ループについて、ループボディを展開する。展開する回数はループボディ内の命令数と、対応するプロセッサのユニット数やレジスタ数を考慮して変化させる。

3.5 load 命令のスケジューリング

現在 newcc に使用されている SPARC version7 用のバックエンドでは、メモリ解析を行なっていないために、スケジューリングを行なう際に load 命令が先行する store 命令を追い越せないようになっている。そのため今回は、SIC でループアンローリングを行った後に、コンパイラが出力したアセンブラコードに手を入れ、load 命令と store 命令間に依存関係がない場合には load 命令が store 命令を追い越すようにスケジューリングを行なった。

以上のようにしてループアンローリングを行なった結果出力された SIC を C 言語に修正すると図 4 になる。

4 性能評価

以上のように、SIC レベルでのループアンローリングを実装した newcc の性能評価を行なった。

評価は Sun UltraEnterprise450 (CPU UltraSparcII 250Mhz ×、メモリ 512MB) 上で行なった。Sun UltraEnterprise450 は 4CPU 構成であるが、今回使用したベンチマークプログラムでは 1CPU しか使用していない。

なお、UltraSparcII は、ALU が 2、FPU が 2、Load/Store ユニットが 1 つあるスーパースカラプロセッサである。

```
int a[100],b[100],c[100];
void calc(void)
{
    int i;
    i = M;
    if (N - 4 < N)
        for (; i < N - 4; i+=4) {
            a[i]   = b[i]   + c[i];
            a[i+1] = b[i+1] + c[i+1];
            a[i+2] = b[i+2] + c[i+2];
            a[i+3] = b[i+3] + c[i+3];
        }
    for (; i < N; i++)
        a[i] = b[i] + c[i];
```

図 4: 展開後の sample source

Livermore Loop	Speed up rate
1 – hydro fragment	1.09
3 – inner product	2.65
5 – tri-diagonal elimination, below diagonal	1.61
11 – first sum	2.51
12 – first difference	1.67

表 2: 評価結果 (速度向上率)

使用したベンチマークは Livermore Loop のうちループ構造が単純であるものを使用した。なお、Livermore Loop はもとは Fortran で書かれているため、C 言語に移植して評価を行なった。

アンローリングを行なわない場合と比較した速度向上率を表 2 に示す。

結果からわかるように、ループアンローリングによってプログラムの実行速度が向上した。

5 まとめ

本稿では、まず複数レベルの中間言語を使用する C コンパイラ newcc を紹介した。このコンパイラは

- 任意のレベルの中間言語をファイルで入出力することができるために、外部に最適化ルーチンを作成することで独自の最適化を施すことができる。
- backend を作成することで各プロセッサのアセンブラコードを出力できるようになる。

等の特徴をもっている。

また、本稿では実際にこのコンパイラの中間言語 SIC のレベルでループアンローリングの実装を行ない、

性能評価を行なった。

今回実装したループアンローリングは、バックエンドでのメモリ解析を自動ではなく手動で行なった。実際にコンパイラによってメモリ解析を行なう場合には、ループ中でポインタを使用せずに配列のみを使用している場合や、関数内のみをみてポインタの値がわかる場合には行なうことができる。しかし、ループ内で使用しているポインタが関数のパラメータで渡されてきた場合にはメモリ解析を行なうことができない。このような場合に、ポインタ解析を全ソースに対してグローバルにを行なうことによりポインタの値のとりうる範囲を推測し、安全な場合にはアンローリングを行なうようにすることもできると思われる。

今後は上記解析を行ない、バックエンドのメモリ解析や、ソフトウェアパイプライニング等必要な機能を追加しながら、このコンパイラを使用してCコンパイラに関する研究を行なっていく予定である。

謝辞

本研究をすすめるにあたり、富士通研究所の木村康則氏に定期的に御助言を頂きました。深く感謝いたします。

参考文献

[1] B.W.Kernighan and D.M.Ritchie. プログラミング言語C第2版. 共立出版, 1989.

[2] SPARC International, Inc. *UltraSPARC™ User's Manual*, July 1997.

[3] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *SIGPLAN '91*, pp. 30–44. ACM, 1991.

[4] 田中哲朗. SPARCの特徴を生かしたUtiLisp/Cの実現法. 情報処理学会論文誌, Vol. 32, No. 5, pp. 684–690, 1991.

[5] 笠原博徳. 並列処理技術. コロナ社, 1991.