目的コードの比較によるCOINS コンパイラ・バックエンドの改良

東京工業大学
理学部
情報科学科

米倉翔一
(0225740)

平成18年度卒業論文

指導教官 佐々 政孝 教授

２月６日

目 次

第1章　はじめに **5**
1.1　背景 5
1.2　概要 5

第2章　準備 **7**
2.1　制御フローグラフ 7
2.1.1　制御フローグラフ 7
2.1.2　ループ 7
2.2　並列化コンパイラ向け共通インフラストラクチャCOINS 8
2.2.1　背景 8
2.2.2　構成 9
2.2.3　COINS のバックエンドについて 10
2.3　レジスタプロモーション 11
2.3.1　レジスタプロモーションの概要 11
2.3.2　COINS のレジスタプロモーション 11

第3章　本研究の流れ **13**
3.1　SPEC ベンチマークの測定 13
3.2　プロファイル 14
3.3　比較するソースコード 15

第4章　レジスタプロモーションのバグの修正 **17**
4.1　コードの例 17
4.2　原因と改良 18

第5章　レジスタプロモーションでのループ脱出先ブロックへのストア命令の削減 **22**
5.1　コードの例 22
5.2　実装 23

第6章　整数定数による除算、剰余算命令の最適化 **24**
6.1　コードの例 24
6.2　変換方法 25
6.2.1　2のべき乗による符号付き除算 25
6.2.2　2のべき乗による除算の符号付き剰余 26
6.2.3　2のべき乗以外での符号付き除算と剰余 26
6.3　実装 29
6.3.1　はじめに 29

6.3.2 全体構造 29
6.3.3 各段階の説明 30

第7章 実行結果の評価と考察 31
7.1 実行環境とテストプログラム 31
7.2 実行結果 32
7.3 考察 32
7.4 今後の課題 33
7.4.1 除算命令の最適化の適用マシンの拡大 33
7.4.2 実行時間に影響の大きい部分の改善 33
7.4.3 その他の改善の方向 33
7.5 関連研究 34
7.5.1 整数除算、剰余算の変換の研究 34
7.5.2 最適化の組み合わせの研究 34
7.5.3 ポインタ解析 34

第8章 まとめ 35

図 目 次

2.1 制御フローグラフの例 8
2.2 並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図 9
2.3 LIR での制御フローグラフと基本ブロックの例 10

3.1 関数 generateMTFValues のソースコード 16

4.1 バグの出るループの例 20

7.1 SPEC ベンチマークの実行結果（A との相対値） 32

表 目 次

3.1 Sun Blade 1000 の主な仕様 13
3.2 COINS と gcc での SPEC ベンチマークの実行結果 (単位は秒) 14
3.3 256.bzip2 のプロファイル結果の一部 14

4.1 バグの出る部分のコードの例 17
4.2 レジスタプロモーションを行わなかった場合 18

6.1 除算、剰余算のコードの例 24
6.3 2 のべき乗以外での符号付き除算 27
6.4 除数が-2 以下の符号付き除算 28

7.1 Sun Blade 1000 の主な仕様 31
7.2 SPEC ベンチマークの実行結果 32

第1章　はじめに

1.1　背景

COINS [1] は、コンパイラ研究の基盤となる共通のコンパイラの作成を目的に開発された、並列化コンパイラ向け共通インフラストラクチャである。COINS にはSSA 最適化、レジスタプロモーションなどのさまざまな最適化が備えられており、出力する目的コードの実行時性能も向上している。しかし gcc [2] の出力する目的コードと比較すると、実行時間に差があるものが多い。

1.2　概要

本研究では、COINS と gcc で同じプログラムをコンパイルして出力された目的コードを比較し、その結果実行時間に差が出る原因と思われる部分を見つけ、より良いコードを生成するように COINS コンパイラの改良を行った。具体的には、

- レジスタプロモーションにあったバグの修正
- レジスタプロモーションのループ脱出先ブロックへの、メモリへのストア命令の削減
- 整数定数による整数の除算、剰余算の最適化

の 3 つの改良を行った。このコードの比較にはSPARC [8] [9] の目的コードを用いた。　レジスタプロモーション [4] とは、プログラムのコードのいくらかの部分で、普通はメモリに入っている値をレジスタに移して、コードの実行時間を改善することである。COINS ではグローバル変数でいくつかの条件を満たすもののうち、ループ内で使用されているものに対してレジスタプロモーションを行っている。このときの処理は、ループの入り口でレジスタプロモーションを行う変数のメモリに入っている値をレジスタに移し、ループ内ではレジスタを参照するようにする。その後ループからの脱出先となる部分に、値をレジスタからメモリに戻すストア命令を挿入する。

1 つ目の改良は、ループの脱出先となる部分に挿入されるストア命令が、ループ内にも挿入されてしまうバグがあったため、これを修正した。

2 つ目の改良は、この出口でのストア命令を挿入する必要がない変数（ループ内で値が変わらない変数）に対しては、ストア命令を挿入しないようにし、ストア命令を削減したものである。

多くのコンピュータでは、整数の除算は非常に時間がかかる。基本的な add 命令の時間の 20 倍以上であることもまれではなく、オペランドが小さいときでさえも同様に長い時間がかかるのが普通である。しかし、除数が整数定数のときは除算命令を避け、実行時間の短い他の命令の列に置き換えることができる。3 つ目の改良は、Henry S. Warren, Jr の著

書「Hacker’s Delight」[5] [6] で紹介されたアルゴリズムをもとに、整数定数による整数の除算、剰余算命令の置き換えを行った。1 つ目と 2 つ目の改良は目的機械に非依存、3 つ目の改良は実装だけが一部目的機械に依存している。

本論文の構成は以下のとおりである。

第 2 章では、本研究で使用した COINS コンパイラとレジスタプロモーションの説明と、以後の章で使用する用語の説明を行う。第 3 章では、本研究の流れを説明する。第 4 章から第 6 章では、今回行った改良それぞれについてコードの例を示し、その改良法について説明を行う。第 7 章では、それぞれの改良についてその結果を比較、評価する。

第2章　準備

2.1　制御フローグラフ

本論文の説明にあたって制御フローグラフの知識が必要となる。本節では制御フローグラフに関する説明を行う。

2.1.1　制御フローグラフ

プログラムの制御の流れをグラフで表したものを**制御フローグラフ**(control flow graph) と呼ぶ。制御フローグラフは基本ブロックをノードとしてそれらの間を分岐や合流を表す有向辺で結んだ有向グラフである。図 2.1 に制御フローグラフの例を示す。

基本ブロック(basic block) は、文 (statement) の列で、その間には分岐も合流もないものである。基本ブロックの先頭には一般に合流があり、最後からは分岐がある。基本ブロック中の文は先頭から最後まで一直線に実行される。

制御フローグラフにおいて、ノードXからノードYに向かって有向辺が引かれていることを、X → Y と表記する。ここでXはYの**先行ブロック**（predecessor block)、YはXの**後続ブロック**(successor block) という。Xの先行ブロックの集合を pred(X)、後続ブロックの集合を succ(X) と表す。図 2.1 において pred(5)={3,4},　succ(7)={8,9} である。

ノードの列 X_0 , X_1 ,..., X_i (i ≥ 0) に対して、e_1 : $X_0 \rightarrow X_1$, e_2 : $X_1 \rightarrow X_2$,..., e_i :$X_{i-1} \rightarrow X_i$ (i ≥ 0) なる有向辺が存在すると仮定する。このとき X_0 から X_i に**パス**(path) が存在するという。

フローグラフの基本ブロック中の連続した文の間の地点を**点**(point) という。基本ブロックの最初の文の直前と最後の文の直後も点という。

フローグラフの初期ノード（基本ブロック）からノード n_2 へいたるすべてのパスが必ずノード n_1 を通るとき、ノード n_1 はノード n_2 を**支配**するといい「$n_1 \quad dom \quad n_2$」と記す。この定義では、すべての頂点は自分自身を支配することになる。図 2.1 において 7 dom 9 である。

フローグラフ中の有向辺 $n_2 \rightarrow n_1$ は「$n_1 \quad dom \quad n_2$」が成り立つとき**帰辺**であるという。

2.1.2　ループ

帰辺 $b \rightarrow h$ があるとき、この辺に関する**自然ループ**(natural loop) とは、hと、hを通らずにbに到達できるようなすべての頂点（bを含む）をあわせたものである。自然ループは**入り口点**(entry point) となる頂点（基本ブロック）がただ１つである。これをヘッダという。ヘッダはループ内のすべての頂点を支配する。ループを繰り返すためのパス、つまりヘッダへと戻るパス（帰辺）が少なくとも１つある。

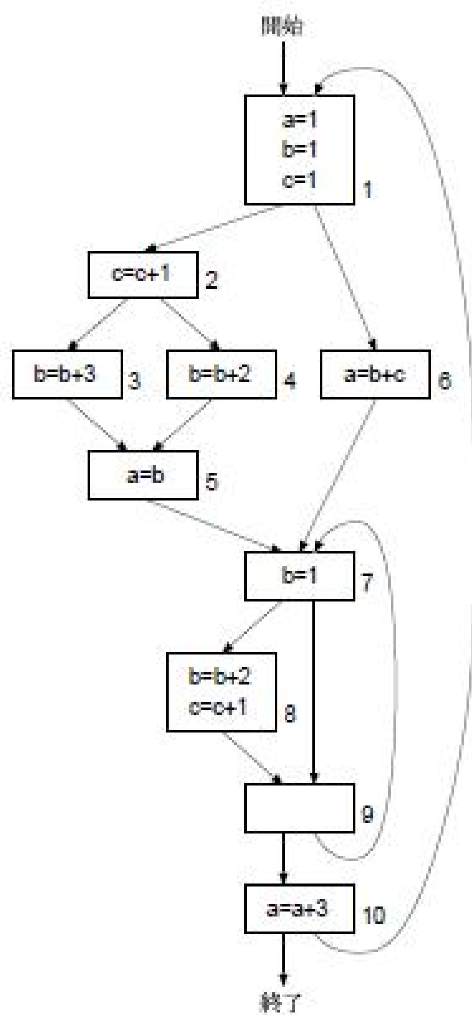

図 2.1: 制御フローグラフの例

２つの自然ループは、ヘッダが異なれば互いに共通部分がないか一方のループが他方のループに完全に含まれることが知られている。ループ内にない後続ブロックをもつ（ループ内の）基本ブロックを出口ブロックという。

ループＡとループＢがあり、ループＡの要素ブロックがすべてループＢの要素ブロックでもあるとき、つまりループＡがループＢに含まれるとき、ループＢをループＡのサラウンディングループ (surrounding loop) であるという。

2.2 並列化コンパイラ向け共通インフラストラクチャCOINS

2.2.1 背景

COINS はコンパイラ研究の基盤となる共通のコンパイラの作成を目標として研究が進められているコンパイラ・インフラストラクチャである。つまり、組み合わせ可能なコンパ

イラ部品で構成される共通インフラを作り、その上に各企業や研究者がそれぞれの目的に合う機能部品を加えることができるようにすることを目的としている [1]。

2.2.2 構成

一般にコンパイラは**フロントエンド**(front end) と**バックエンド**(back end) から構成される。フロントエンドは**原始プログラム**(source program) を**中間コード**(intermediate code) と呼ばれる内部形式に変換する。バックエンドは中間コードを計算機の機械コードに変換する。フロントエンドはさらに**字句解析器**(lexical analyzer)、**構文解析器**(syntax analyzer)、**意味解析器**(semantic analyzer) に分けられる。バックエンドは**最適化器**(optimizer) と**コード生成器**(code generator) に分けられる。これらの各部分はコンパイラのフェーズと呼ばれる。

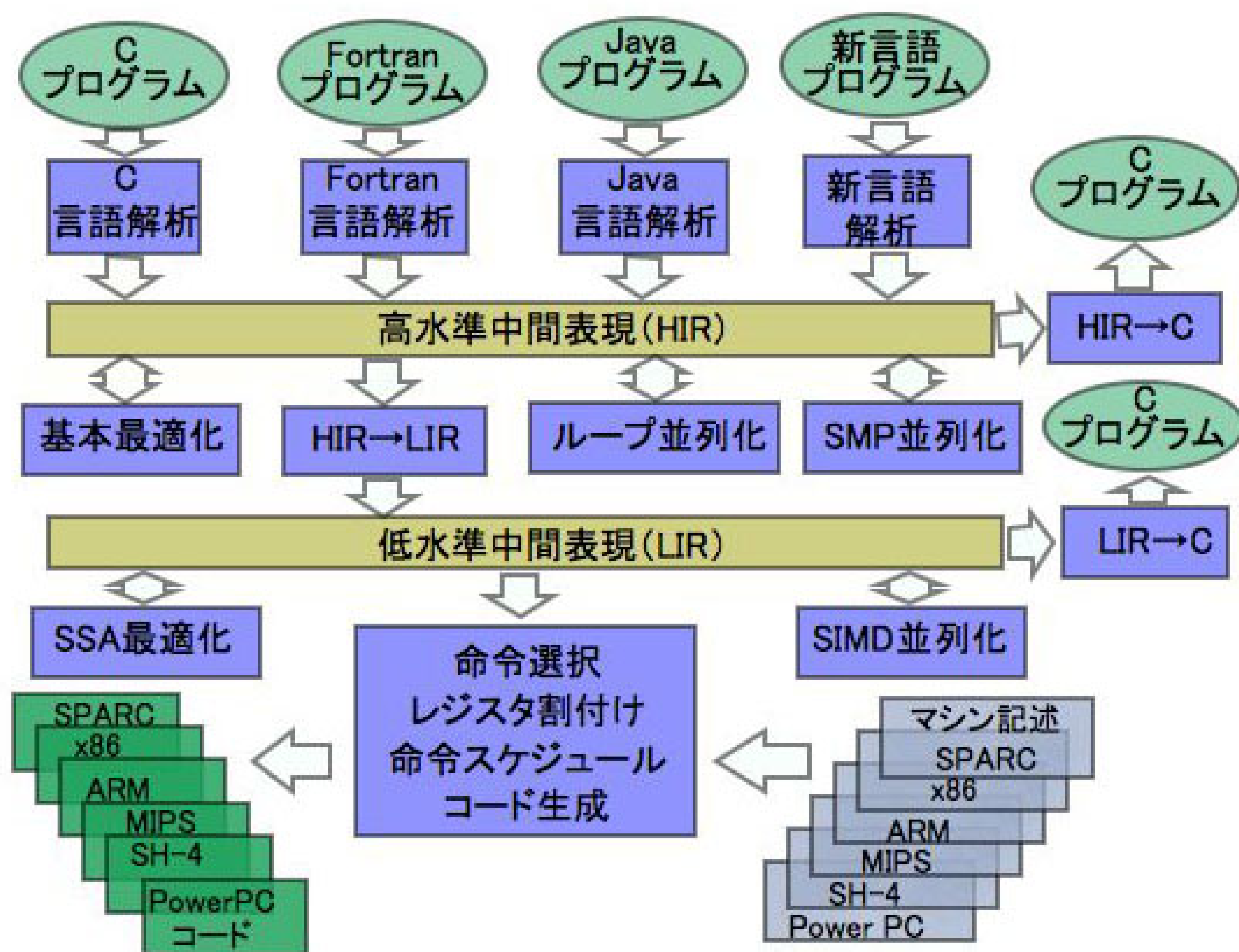

図 2.2: 並列化コンパイラ向け共通インフラストラクチャ(COINS) 概念図

本研究で用いるコンパイラ・インフラストラクチャCOINSの概念図を図2.2に示す。COINSでは、複数の入力言語、複数の目的機種に対応する 2 つの中間コードがある。入力言語の論

理構造に近いレベルの中間コードを**高水準中間表現**(high-level intermediate representation, HIR) と呼び、機械語に近いレベルの中間コードを**低水準中間表現**(low-level intermediate representation, LIR) と呼ぶ。COINS のソースはすべて Java 言語で書かれている。

2.2.3 COINS のバックエンドについて

COINS のバックエンドでは、中間コードの構成要素に関して、構成要素それぞれに対応するクラスが用意されており、構成要素はそのクラスのインスタンスとして表現される。例えば中間コード内の手続きは、これに対応する Function クラスのインスタンスとして扱われる。

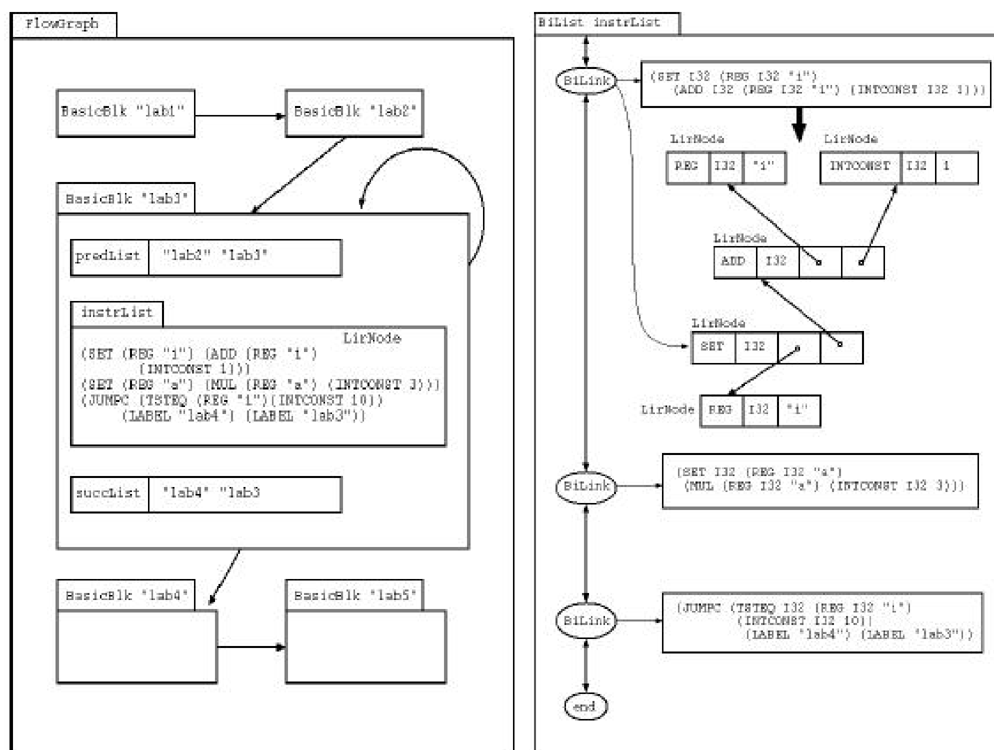

図 2.3: LIR での制御フローグラフと基本ブロックの例

LIR において手続き内部の制御フローグラフを現す FlowGraph クラスの内部の様子を図 2.3 に示した。左側が FlowGraph クラスの内部を示している。FlowGraph クラスのフィールドには、この制御フローグラフに属する基本ブロックを表す BasicBlk クラスが保存されている。BasicBlk クラス内には、そのブロックで実行される命令を保存する instrList というリスト、このブロックの後続ブロックを保存する succList などがある。

右側は基本ブロック内の命令リスト instrList の内部を示している。instrList の要素の一つ一つが、命令文となっている。例えば図の例でいえば、一番目の要素は

(SET I32 (REG I32 ”i”) (ADD I32 (REG I32 ”i”) (INTCONST I32 1)))

という命令である。これは変数 i の値がしまってあるレジスタの中身（つまり i の値）に 1 を足しこむというものである。(REG I32 ”i”) は変数 i の値をしまったレジスタの中身（つ

まり i の値)、(INTCONST I32 1) は定数 1 を表す。ADD は二つの引数を足した結果を表す。SET は第一引数に、第二引数の値を代入するという命令である。見て分かるように、ひとつの命令文は木構造になっている。先ほどの命令の木構造は、図の右側の中ほどに示されている。木のノード一つ一つは、命令文を構成する要素を表す LirNode というクラスのインスタンスである。

また、図では示されていないが、(STATIC I32 ”a”)、(FRAME I32 ”b”) はそれぞれ、グローバル変数 a のメモリアドレス、ローカル変数 b のメモリアドレスを表す LirNode である。このアドレスにしまわれている値を取り出すには、メモリから値を取り出す命令 MEM を使って (MEM I32 (STATIC I32 ”a”)) のようにする。例えば、グローバル変数 a にローカル変数 b の値を代入するという命令は MEM と SET を用いて

(SET I32 (MEM I32 (STATIC ”a”)) (MEM I32 (FRAME I32 ”b”)))

のようになる。

さて、グローバル変数のアドレス、ローカル変数のアドレス、レジスタそれぞれの LirNode には symbol(シンボル) と呼ばれる Symbol クラスのインスタンスが割り当てられていて、それぞれの symbol フィールドにしまわれている。簡単に言うと、symbol はタグを表すものである。例えば先ほど出てきた命令には (REG I32 ”i”) という LirNode が二つある。この二つは LirNode クラスの異なるインスタンスであるが、同じレジスタの値を表す。そのため、どちらの (REG I32 ”i”) の symbol も、同一の Symbol クラスのインスタンスになっている。同一の変数の値がしまわれているレジスタには同一の Symbol クラスのインスタンスを対応させるというわけである。(STATIC I32”a”)、(FRAME I32 ”b”) といった LirNode に関しても同様で、(STATIC I32 ”a”) が命令で複数回使われていても、同じグローバル変数 a のアドレスを表すなら、どの (STATIC I32 ”a”) の symbol も Symbol クラスの同じインスタンスである。

2.3 レジスタプロモーション

2.3.1 レジスタプロモーションの概要

レジスタプロモーション（レジスタ促進）とは、プログラムのコードのいくらかの部分で普通はメモリに入っている値をレジスタに移して、コードの実行時間を改善することである。 [4] 一般的にメモリへのアクセスよりもレジスタへのアクセスのほうが早いので、ループの中で頻繁にアクセスされる変数などに適用すると効果が大きい。

2.3.2 COINS のレジスタプロモーション

COINS のバックエンドでは、ローカル変数のうちアドレスが取られていない変数、すなわち、

$$p = \&a;$$

のような操作が行われていない変数 a についてはそれを擬似レジスタ名で置き換え、レジスタ割付けの結果によってレジスタに割り付けられるようにしている。

グローバル変数については、次の 2 つの条件

- ループ内に関数呼び出しがない。
- ループ内にポインタによるメモリへの間接参照がない

を満たすループについて、ループ内で参照されているグローバル変数に対してレジスタプロモーションを行う。具体的には、そのループに入る前にレジスタプロモーションを行うグローバル変数の値をレジスタにロードし、そのループを出たところでレジスタからそのグローバル変数に代入する。入れ子になっているループに対しては、上の条件を満足するループの中で一番外側のループを対象としている。このグローバル変数のレジスタプロモーションは、オプションで regpromote を指定すると、バックエンドの HIR から LIR への変換が行われた直後に実行される。 [4]

第3章　本研究の流れ

本研究では、COINS と gcc で同じプログラムをコンパイルして出力された目的コードを比較し、その結果実行時間に差が出る原因と思われる部分を見つけ、より良いコードを生成するように COINS コンパイラの改良を行う。本章ではその中の目的コードの比較の流れの部分を説明する。

3.1　SPEC ベンチマークの測定

まず、本研究の改良を行う前の COINS と gcc を用いての SPEC CPU2000 [7] ベンチマークの測定を行った。実験は、COINS version CVS(Date 2005/10/15 19:55)、gcc version 4.0.2 を用い、Sun Microsystems の Sun Blade 1000 (表 3.1) で行った。

アーキテクチャ	Superscalar SPARC Version 9
プロセッサ種別	750MHz UltraSPARC
プロセッサ数	2
1 次キャッシュ	64KB データ、32KB インストラクション
2 次キャッシュ	8MB 外部キャッシュ
メモリ容量	1GByte
オペレーティング環境	SunOS 5.8

表 3.1: Sun Blade 1000 の主な仕様

COINS で行った最適化は、SSA 最適化、命令スケジューリング、レジスタプロモーションである。目的コードの比較はこれらの最適化をかけたときに出力された目的コードで行った。

SSA 最適化のオプションは

prun/divex/cse/cstp/hli/osr/hli/cstp/cpyp/preqp/cstp/rpe/dce/srd3

prun:Pruned SSA 形式への変換、srd3:Sreedhar の方法 3 による逆変換
cse:共通部分式除去、cstp:条件分岐を考慮した定数畳み込みと定数伝播、
hli:ループ不変式移動、divex:式を 3 アドレス方式に変換（代入の右辺の演算子はたかだか１つ）
osr:ループの機能変数に関わる演算の強さの軽減と判定の置き換え、
cpyp:コピー伝播、preqp:質問伝播に基づく大域値番号付けと部分冗長除去、
rpe:冗長な Phi 関数の除去、dce:無用命令削除

とした。gcc は最適化オプション O2, O3 の 2 通りを行った (両方とも-mcpu=v8 オプションもつけている)。

	COINS	gcc-O2	gcc-O3
164.gzip	732	538	543
175.vpr	639	492	487
181.mcf	466	478	441
197.parser	769	665	599
255.vortex	704	558	551
256.bzip2	788	483	478
300.twolf	1052	818	815
171.swim	2093	1906	1904
172.mgrid	1677	1943	1943
177.mesa	752	655	644
179.art	506	416	438
183.equake	855	854	813

表 3.2: COINS と gcc での SPEC ベンチマークの実行結果 (単位は秒)

実行時間の測定結果を表 3.2 に示す。実行時間は 3 回の測定の中央値を示している。

結果を見ると、181.mcf,172.mgrid では COINS のほうが gcc の O2 より速く、172.mgrid では gcc の O3 よりも速い。しかし、そのほかのプログラムでは gcc のほうが速く、特に 256.bzip2 ではかなり実行時間の差が大きい。

3.2 プロファイル

SPEC ベンチマークの結果から、gcc とのベンチマークの実行時間の差が最も大きい 256.bzip2 について目的コードの比較を行うことにした。比較の範囲を絞るため、プログラムのプロファイルを用い、実行時間の長い関数について比較を行うことにする。しかし、COINS にはプロファイルをとる機能がない。そこで gcc でのプロファイルの結果を用いることにした。gcc の-O3 オプションをつけたときのプロファイルの結果の一部を表 3.3 に示す。

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
23.74	90.92	90.92	166	547.71	547.71	generateMTFValues
15.15	148.96	58.04	186078174	0.00	0.00	fullGrU
14.46	204.36	55.40	166	333.73	358.71	getAndMoveToFrontDecode
13.47	255.97	51.61				internal mcouns
7.21	283.57	27.60	166	166.27	190.57	sendMTFValues
6.45	308.26	24.69	176	140.28	540.57	sortIt
3.85	324.12	15.86	166	95.54	124.61	undoReversibleTransformation fast

表 3.3: 256.bzip2 のプロファイル結果の一部

表の値は左から

- 関数の実行の全体に占める割合
- (手続き呼び出しを含んだ) 関数の実行時間 (秒)
- 関数自身の実行時間
- 関数の呼び出し回数
- １回の平均実行時間 (ミリ秒)
- (手続き呼び出しを含んだ) １回の平均実行時間
- 関数の名前

をあらわしている。この結果より、実行時間の長い generateMTFValues, fullGtU の２つの関数を中心に目的コードの比較をおこなった。

3.3 比較するソースコード

以後の３つの章の説明に用いる bzip2.c の関数 generateMTFValues を以下の図 3.1 に示す。これは前節のプロファイル結果で最も実行時間のかかっていた関数であり、実際に目的コードの比較を行った部分のソースコードのひとつである。

```
   void generateMTFValues ( void )
{
   UChar  yy[256];
   Int32  i, j;
   UChar  tmp;
   UChar  tmp2;
   Int32  zPend;
   Int32  wr;
   Int32  EOB;

   makeMaps();
   EOB = nInUse+1;
   for (i = 0; i <= EOB; i++) mtfFreq[i] = 0;

   wr = 0;
   zPend = 0;
   for (i = 0; i < nInUse; i++) yy[i] = (UChar) i;

   for (i = 0; i <= last; i++) {                              —（A）

      UChar ll_i;
      j = 0;
```

```
        tmp = yy[j];
      while ( ll_i != tmp ) {                                    — (1)
         j++;
         tmp2 = tmp;
         tmp = yy[j];
         yy[j] = tmp2;
      };
      yy[0] = tmp;
      if (j == 0) {
         zPend++;
      } else {
         if (zPend > 0) {
            zPend--;
            while (True) {
               switch (zPend % 2) {
                  case 0: szptr[wr] = RUNA; wr++; mtfFreq[RUNA]++; break;
                  case 1: szptr[wr] = RUNB; wr++; mtfFreq[RUNB]++; break;
               };
               if (zPend < 2) break;
               zPend = (zPend - 2) / 2;
            };
            zPend = 0;
         }
         szptr[wr] = j+1; wr++; mtfFreq[j+1]++;
      }
   }                                                              — (2)
   if (zPend > 0) {                                               — (B)
      zPend--;
      while (True) {
         switch (zPend % 2) {                                     — (X)

            case 0:  szptr[wr] = RUNA; wr++; mtfFreq[RUNA]++; break;
            case 1:  szptr[wr] = RUNB; wr++; mtfFreq[RUNB]++; break;
         };
         if (zPend < 2) break;
         zPend = (zPend - 2) / 2;                                 — (Y)
      };
   }
   szptr[wr] = EOB; wr++; mtfFreq[EOB]++;
   nMTF = wr;
}
```

図 3.1: 関数 generateMTFValues のソースコード

第4章　レジスタプロモーションのバグの修正

本章から3章に渡って、本研究で行った改良について説明する。説明に当たってまず比較した目的コードのうち、改良する部分を示し、そのあとに改良方法を説明する。

4.1　コードの例

まず、本研究で発見したレジスタプロモーションのバグの例を4.1示す。これは前章の

ソースプログラム	COINS			gcc	
while(ll_i != tmp)	.L2113:		.LL293:		
{	sethi	%hi(last), %i2		mov	%i0, %g3
j++;	or	%i2, %lo(last), %i2		add	%i2, 1, %i2
tmp2 = tmp;	sethi	%hi(block), %l7		ldub	[%i1+%i2], %i0
tmp = yy[j];	or	%l7, %lo(block), %l7		and	%i0, 0xff, %g2
yy[j] = tmp2;	st	%l1, [%i2]		cmp	%i3, %g2
};	sethi	%hi(zptr), %i2		bne	.LL293
	or	%i2, %lo(zptr), %i2		stb	%g3, [%i1+%i2]
	st	%l0, [%l7]			
	sethi	%hi(szptr), %l7			
	or	%l7, %lo(szptr), %l7			
	st	%o0, [%i2]			
	add	%i4, 1, %i4			
	mov	%i1, %i2			
	st	%i0, [%l7]			
	ldsb	[%i4], %i1			
	stb	%i2, [%i4]			
	and	%i1, 255, %i2			
	cmp	%g2, %i2			
	bne	.L2113			
	add	%l6, 1, %l6			

表 4.1: バグの出る部分のコードの例

図3.1の関数generateMTFValuesのなかの（1）のwhileループ部分の目的コードである。左側がソースプログラム、真ん中がCOINSで3章のオプションでコンパイルしたときの目

的コード、右側が gcc の O3 オプションでコンパイルしたときの目的コードである。

これを見ると COINS は 20 命令、gcc は 7 命令と、命令数に大きく差があるころが分かる。coins のほうのコードの 1 11 行目と 14 行目は sethi,or 命令で変数のメモリアドレスを取得し、そのメモリアドレスにレジスタの値をストアしている。

レジスタプロモーションを行わずにコンパイルした場合の、同じ部分の目的コードは表 4.2 のようになっている。これを見るとレジスタプロモーションによって表 4.1 の sethi, or

```
.L1723:
        mov   %i0, %i1
        add   %i3, 1, %i3
        ldsb  [%i3], %i0
        stb   %i1, [%i3]
        and   %i0, 255, %i1
        cmp   %l3, %i1
        bne   .L1723
        add   %l1, 1, %l1
```

表 4.2: レジスタプロモーションを行わなかった場合

とストア命令が挿入されていることが分かる。この追加されているコードは、ループの脱出先ブロックに挿入されるはずのレジスタプロモーションを行った変数の値を、メモリーにストアする命令である。本来は入れ子になっているループの場合、最も外側のループに対してレジスタプロモーションを行うので、図 3.1 の場合(A) の for ループに対してプロモーションを行っている。(1)の部分はループの脱出先ではないのでここにレジスタをストアする命令は来ないはずであるが、余計な場所にストア命令が挿入されていることが分かる。このバグではプログラムの実行結果は変わらないが、実行時間にはかなり影響があると思われる。

4.2 原因と改良

レジスタプロモーションの実装では、ループの脱出先のブロックのリスト ExitList を用いて、ExitList に含まれるブロックにストア命令を挿入している。この ExitList の作成手順は、

1. ループの要素となっているそれぞれのブロックに対し、そのブロックの後続ブロックのリストの要素を ExitList に加える。
2. ExitList からそのループの要素ブロックを削除する

となっている。つまり ExitList はループの要素ブロックの後続ブロックのうち、ループの要素ブロックでないものである。このループの要素ブロックのリストの作り方は、

1. 関数の全ブロックに対し、ブロックがループの要素ならヘッダの一致するループの要素ブロックに加える。

2. compMember メソッドを用いて、サラウンディングループがあった場合、外側のループの要素ブロックリストに内側のループの要素ブロックを加える。

である。COINS にはループ解析を行うクラス LoopAnalysis がある。LoopAnalysis を用いると、基本ブロックがループの要素の場合、そのループのヘッダのブロックが分かるので、１ではそれを用いてヘッダのループの要素ブロックリストにその基本ブロックを加えている。しかしループが入れ子になっている場合、内側のループの中のブロックは外側のループの要素ブロックに入らない (内側のループのヘッダのみ外側のループの要素ブロックリストに入る)。そのため２で compMember メソッドを用いて内側のループの要素ブロックも加えている。

今回バグがあったのはこの compMember メソッドである。compMember メソッドでは、ループの要素ブロックの中に、あるループのヘッダになっているものがある場合、そのブロックがヘッダになっているループの要素ブロックリストを blist という変数に保存しておく。そして blist に入っているブロックが外側のループの要素ブロックリストに含まれていなかった場合、そのリストに追加している。しかしループの中に２つループがあった場合、２つ目のループの要素ブロックリストで blist が書き換えられ、１つ目のループの要素ブロックが外側のループの要素ブロックリストに加えられていなかった。

以下に例を示す。図 4.1 の中には L1，L2，L3 の３つのループがあり、それぞれのヘッダは B1，B2，B5 になっている。要素ブロックリストの作り方の１を行うとそれぞれの要素ブロックリストには

L1：B1，B2，B4，B5，B7 の５つ
L2：B2，B3 の２つ
L3：B5，B6 の２つ

が入る。B3，B6 のヘッダはそれぞれ B2，B4 となるため、L1 の要素ブロックリストには加えられない。次に要素ブロックリストの作り方の２を行う。L1 の要素ブロックリストの中でまず B2 がループ L3 のヘッダになっている（そのループ自身のヘッダは除くため、B1 は無視する）。そこで、リスト blist を L2 の要素ブロックリストにする。このとき、blist の中には B2，B3 の２つが入っている。しかし、そのあと B5 がループ L3 のヘッダになっていて、修正前の compMember では blist が L3 の要素ブロックリストに書き換えられてしまっていた。そのため、L3 の要素の B6 は L1 の要素ブロックリストに入るが B3 は入らず、L1 の要素ブロックリストの中身は

B1，B2，B4，B5，B6，B7

になる。その後ループの脱出先ブロックのリスト ExitList を作成する。手順１でループ１の要素ブロックの後続ブロックが ExitList に加えられる。このとき ExitList は

B2，B3，B4，B5，B6，B5，B7、B1，B8

となる。次に手順２で L1 の要素ブロックリストに含まれているものを除くと

B3，B8

となる。正しい ExitList は B8 のみなのであるが、誤った ExitList により B3 に不要なストア命令を挿入してしまっていた。

図 3.1 の関数 generateMTFValues に話を戻すと、そこでは (A) の for ループの中に (1),(2) の２つのループがある。そこで上の例と同様にして (1) のループの中にストア命令が挿入され、表 4.1 のような出力結果になっていた。

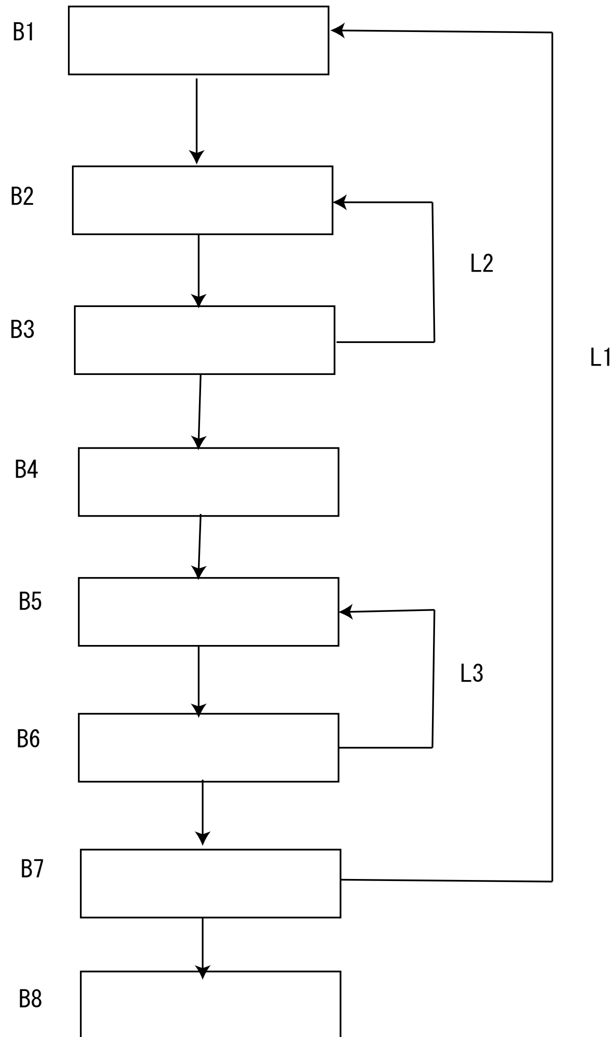

図 4.1: バグの出るループの例

本研究では、レジスタプロモーションを行う java ファイル RegPromote.java の compMember メソッドに修正を施した。バグの原因は blist の中身が書き換えられてしまっていたためだったので、ループのヘッダとなるブロックを発見し、blist にそのブロックをヘッダとするループの要素ブロックリストを追加したら、すぐに外側のループの要素ブロックリストに blist を加えるように修正した。修正後のレジスタプロモーションを行った時の図 3.1 の（1）の while ループの部分の目的コードは

```
.L1920:
        mov  %i0, %i1
        add  %i3, 1, %i3
        ldsb [%i3], %i0
        stb  %i1, [%i3]
        and  %i0, 255, %i1
        cmp  %l4, %i1
        bne  .L1920
        add  l3, 1, %l3
```

となり、バグの部分が除去された。

第5章　レジスタプロモーションでのループ脱出先ブロックへのストア命令の削減

本章では2つ目の改良のレジスタプロモーションでのループ脱出先ブロックへのストア命令の削減について説明を行う。

5.1　コードの例

まず、改良を行う前の目的コードを以下に示す。これは図3.1の(2)と(B)の部分をCOINSとgccで前述のオプションでコンパイルしたときの目的コードである。

coins

```
.L1595:
        sethi   %hi(last), %i2
        or      %i2, %lo(last), %i2
        sethi   %hi(block), %i1
        or      %i1, %lo(block), %i1
        st      %l1, [%i2]
        sethi   %hi(zptr), %i2
        or      %i2, %lo(zptr),%i2
        st      %l0, [%i1]
        sethi   %hi(szptr), %i1
        or      %i1, %lo(szptr),%i1
        st      %o0, [%i2]
        st      %i0, [%i1]
        cmp     %l5, 0
        ble     .L303
        sll     %o3, 1, %i4
```

gcc

```
        cmp     %g3, 0
.LL251:
        ble     .LL237
        sethi   %hi(mtfFreq), %o4
```

ここは(A)のループを出た直後の部分であり、COINSの14,15行目、gccの1,3行目のcmp,bleの2つの命令が(B)のif文の条件分岐の部分である。COINSの13行目までは、レジスタプロモーションによるループ(A)の出口のストア命令である。(A)のループの中ではlast,block,zptr,szptrの4つの変数に対してレジスタプロモーションを行っている。しかし、この(A)のループの中でこの4つの変数の値は変わっていないため、ループ出口での

ストア命令は不要である。実際、gcc でもレジスタプロモーションは行っているがストア命令を挿入していない。

そこで、レジスタプロモーションを行う変数のうち、ループの中で値の変わらない変数に対してはループ出口にストア命令を挿入しないようにした。

5.2 実装

レジスタプロモーションを行う java ファイル RegPromote.java の insertNewInst メソッドに改良を施した。insertNewInst メソッドは、ループの出口に新しいブロックを挿入しメモリ、レジスタ間のロードやストアの命令を挿入するメソッドである。このメソッドのストア命令を挿入する部分に改良を施す。改良した部分のアルゴリズムは以下のようになる。

1. ループの中に値が変更されているグローバル変数があるかどうかを調べる。
2. あった場合、その変数の symbol を addStInst というリストに入れる。
3. レジスタプロモーションを行うグローバル変数のうち、その変数の symbol が addStInst に含まれていた時、ストア命令を挿入する。含まれていないものに対してはストア命令を挿入しない。

バックエンドのはじめのフェーズでの LIR におけるグローバル変数の値の変更について述べる。値が変更されるときの LIR は、グローバル変数への代入の式になっている。LIR ではグローバル変数のアドレスは、(STATIC ”a”) のように表される。これはグローバル変数 a のアドレスを示すものであり、ローカル変数なら (FRAME ”a”) のようになる。メモリにしまわれる値を参照するときは、MEM という命令を使う。例えばグローバル変数 a の値は、

(MEM I32 (STATIC ”a”))

という命令で参照できる。I32 というのは参照する値の型である。レジスタやメモリへの値の代入には SET 命令を使う。グローバル変数への値の代入は

(SET I32 (MEM I32 (STATIC ”a”)) (INTCONST I32 5)))

のようになる。これはグローバル変数 a に 5 を代入している。以上のことから、(SET (MEM (STATIC ⋯))) のような形の式を見つけたときグローバル変数の値が変更されたと判断する。ステップ１ではループの中の式全体に対して、この形の式がないか調べている。

ステップ 3 では、まず boolean 型の変数 storeSt を用意しておき、ステップ１でループの中にグローバル変数の値の変更があったときのみ storeSt を true にするようにしておく。これは storeSt が false のときグローバル変数の値の変更はないので、無駄に変数の symbol が L_Lift(レジスタプロモーションを行った変数のシンボルのリスト) に含まれていないか調べるのを防ぐためである。また、レジスタプロモーションでは脱出先ブロックの前に新しいブロックを挿入し、そこにストア命令を挿入している。そのため無駄なブロック挿入を防ぐためでもある。storeSt が true のときは、レジスタプロモーションを行うグローバル変数の symbol が L_Lift に含まれていたときだけ、従来の insertNewInst メソッドのストア命令の挿入の部分の処理を行う。

第6章　整数定数による除算、剰余算命令の最適化

6.1　コードの例

まず、改良する前のコードの例を表6.1に示す。これは図3.1の(X),(Y)の整数定数による除算、剰余算の部分の目的コードである。以下で用いるマシンは32ビットマシンである。これは上が2での剰余算、下が2での除算のコードである。gccのコードは2で割る部分は

ソースプログラム	COINS		gcc	
zPend%2	sra	%l1, 31, %g1	srl	%i4, 31, %g2
	mov	%g1, %y	add	%i4, %g2, %g2
	nop		and	%g2, -2, %g2
	nop		subcc	%i4, %g2, %i0
	nop			
	sdiv	%l1, 2, %g1		
	smul	%g1, 2, %g1		
	sub	%l1, %g1, %i0		
zPend = (zPend - 2) / 2	sub	%l1, 2, %i0	add	%i4, -2, %g2
	sra	%i0, 31, %g1	srl	%g2, 31, %g3
	mov	%g1, %y	add	%g2, %g3, %g2
	nop		sra	%g2, 1, %i4
	nop			
	nop			
	sdiv	%i0, 2, %l1		

表6.1: 除算、剰余算のコードの例

算術右シフト命令sraを用いて計算している。COINSではsdivという除算命令を用いている。しかしgccではdivide命令を用いず、符号付右シフトsraなどを使って計算している。多くのコンピュータでは除算は非常に時間がかかるので、gccのようにdivide命令を避けるのは実行時間の向上に効果があると思われる。そこで、Henry S. Warren, Jr. のアルゴリズム[5]を用いて整数定数による除算、剰余算命令の最適化を実装した。

6.2　変換方法

6.2.1　2のべき乗による符号付き除算

2のべき乗による除算のときは、表6.1のgccのコードと同じように符号付右シフトsraを用いる。右に1ビット分シフトするのは2で割ることに一致するので、右シフトをkビット分行えば数値を2^kで割れる。しかしSPARCでは切り捨て除算を行っていて、被除数が負の数で、割り切れないときは必ずしも一致しない。例えば$-9/4$を考える。この結果は正しくは-2となる。-9を2進数で表すと

11111111111111111111111111110111

であり、$4=2^2$なので2ビット分右シフトすると

11111111111111111111111111111101

となり、これは-3である。この例のように、右シフトを行っただけでは値が1少なくなってしまうことがある。

そこで$q=n/2^k$を求めるのに$n<0$のとき、あらかじめnに2^k-1を加えるようにする。nが2^kで割り切れないときはこれにより1多くなるので正しい結果となる。nが2^kで割り切れるときは1少なくなってしまいそうだが、nに2^k-1を加えることにより2^kで割り切れなくなり、上の例のように右シフトで1少ない値になるのでその結果正しい値になる。例えば$-9/4$のときは$3=2^2-1$を加えると$-6/4$となり割り切れないが、これを右に2ビット符号付きシフトすると-2となり$-9/4$の正しい結果になる。$-8/4$のように割り切れるときは同様にして、$-5/4$となり割り切れなくなるが、同様に-2となり正しい結果になる。

以上により2のべき乗による符号付除算$q=n/2^k$は以下のようなコードを生成するようにする。

```
sra n, k-1, t
srl t, 32-k, t
add n, t, t
sra t, k, q
```

上の3行がnに2^k-1を加える部分である。1行目でk-1ビット符号付右シフトすることにより、$n<0$のときtの上位kビットは1となる。それを32-k論理右シフトすることにより、tは右からkビットのみ1になり、これは2^k-1であり3行目でそれをnに足している。$n>0$のときは符号ビットが0なので、上の2行でtの値は0になるため大丈夫である。

例えば$n=-9,k=3$のときは1行目で

11111111111111111111111111110111

をまず右に2ビット符号付シフトして

11111111111111111111111111111101

となり、2行目でこれを右に29ビット論理シフトすると

00000000000000000000000000000111

となり、これは $7 = 2^3 - 1$ となる。
$n = 8, k = 3$ のときは 1 行目で

00000000000000000000000000001000

を 2 ビット符号付右シフトして

00000000000000000000000000000010

となり、これを 2 行目で 29 ビット論理右シフトして

00000000000000000000000000000000

となり $0 = 2^3 - 8$ が生成される。-2^k で割るときは符号が逆なので、上の 4 命令に

```
sub 0，q，q
```

を追加する。

6.2.2　2 のべき乗による除算の符号付き剰余

$n \div 2^k$ の剰余は $r = n - q * 2^k$ で剰余 r を求める。これは前節の商 q の計算の 4 命令の後に

sll q, k, r r=q*2^k sub n, r, r

を加えることで求められる。このとき商 q の計算の 4 命令目の sra と剰余の計算の 1 命令目の srl では、まず右に k ビットシフトした後、それを左に k ビットシフトしている。これは右から k ビットが 0 で、残りが 1 である -2^k と AND 演算を行うことと一緒である。そこで 2 のべき乗による除算の符号付剰余は以下の 5 命令で計算できる。

sra　n, k-1, t
srl　t, 32-k, t
add　n, t, t
and　t, -2^k, t
sub　n, t, r

6.2.3　2 のべき乗以外での符号付き除算と剰余

ここからはターゲットマシンを 32 ビットマシンとする。

除数が２以上のとき

ここで行うのは n/d を計算することである。基本となる方法は、およそ $2^{32}/d$ であるような、除数 d の逆数の一種を n に掛けて、積の左 32 ビットを取り出すことである。しかし誤差の問題があるため、計算は $2 \leq d \leq 2^{31}$ であるような除数 d が与えられたとき、

$$\lfloor\frac{mn}{2^p}\rfloor = \lfloor\frac{n}{d}\rfloor \qquad (0 \leq n \leq 2^{31} \text{のとき}) \tag{6.1}$$

かつ、

$$\lfloor\frac{mn}{2^p}\rfloor + 1 = \lceil\frac{n}{d}\rceil \qquad (-2^{31} \leq n \leq -1 \text{ のとき}) \tag{6.2}$$

となるような最小の整数mとpを見つけ（ただし、$0 \leq m \leq 2^{32}$ かつ $p \geq 32$)、それを用いて商qを求める。mとpは

$$2^p > n_c(d - rem(2^p, d)) \tag{6.3}$$

$$m = \frac{2^p + d - rem(2^p, d)}{d} \tag{6.4}$$

で求められる。ここで n_c は $rem(n, d) = d - 1$ が成り立つ最大の（正の）nの値である ($n_c \leq 231 - 1$)。

このmとpを用いて除算命令の変換の命令列を生成するが、$2^{31} \geq m < 2^{32}$ のとき 32 ビット整数で表現できないため

$$M = \begin{cases} m & 0 \leq m < 2^{31} \text{ のとき} \\ m - 2^{32} & 2^{31} \leq m < 2^{32} \text{ のとき} \end{cases} \tag{6.5}$$

を用いる。この M をマジックナンバーと呼ぶ。また、$s = p - 32$ とする。この M と s を用いてコードを生成する。ここではレジスタを次のように表記する。

n	:	入力の整数（分子）
a	:	「マジックナンバー」が入る。
t	:	一時レジスタ
q	:	商が入る
r	:	剰余が入る

出力するコードは表 6.3 のようになる。32 ビット整数 2 つの積は常に 64 ビットで表現で

mov	M, a	マジックナンバー M を a にロード
mulhs	a, n, q	$q = M * n/2^{32}$
(add	q, n, q	$M < 0$ のときこの命令を挿入する）
(sra	q, s, q	$s > 0$ のときこの命令を挿入する）
srl	n, 31, t	nが負のとき
add	q, t, q	q に 1 を加算

表 6.3: 2 のべき乗以外での符号付き除算

き、２行目の mulhs 命令は符号付乗算の上位、積の結果 64 ビットの上位 32 ビットをとる命令である。この命令の結果は $mn/2^{32}$ に相当する。これは SPARC の命令にはないが、実

装については6.3節で説明する。3行目のadd命令は$M < 0$のときに挿入する。Mが負になるのは$2^{31} \leq m < 2^{32}$のときで、このとき$M = m - 2^{32}$である。nを足しているのは

$$\frac{(m-2^{32})*n}{2^{32}} + n = \frac{(m-2^{32})*n + n*2^{32}}{2^{32}} = \frac{mn}{2^{32}}$$

とするためである。4行目のsra命令は$s > 0$のときに挿入する。このときは$p = 32 + s$である。式(6.1)のように2^pで割るには、まだ2^{32}でしか割っていないので、さらに2^sで割らないといけないために挿入する。最後の2命令はnが負のときに1を加えている(式(6.2))。

剰余算は$r = n - qd$で求める。これは表6.2.3の命令列に

```
smul q, d, t
sub  n, t, r
```

の2命令を加えればいい。

除数が-2以下の符号付き除算、剰余算

除数が2以上のときと同じように、$-2^{31} \leq d \leq -2$である除数dが与えられたとき

$$\lfloor \frac{mn}{2^p} \rfloor = \lfloor \frac{n}{d} \rfloor \qquad (-2^{31} \leq n \leq 0 \text{ のとき}) \tag{6.6}$$

かつ、

$$\lfloor \frac{mn}{2^p} \rfloor + 1 = \lceil \frac{n}{d} \rceil \qquad (1 \leq n \leq 2^{31} \text{のとき}) \tag{6.7}$$

であるような（絶対値で）最小の整数mとpを使って商を求める(ただし、$-2^{32} \leq m \leq 0$かつ$p \geq 32$)。mとpは

$$2^p > n_c(d + rem(2^p, d)) \tag{6.8}$$

$$m = \frac{2^p - d - rem(2^p, d)}{d} \tag{6.9}$$

で求められる。$-2^{32} \leq m \leq -2^{31}$の時、32ビット整数では表現できないため、マジックナンバーMを

$$M = \begin{cases} m & -2^{31} < m \leq 0 \text{ のとき} \\ m + 2^{32} & -2^{32} \leq m \leq -2^{31} \text{ のとき} \end{cases} \tag{6.10}$$

とし、$s = p - 32$とする。出力するコードは表6.4のようになる。1,2行目は除数が2以上

mov	M, a	マジックナンバーMをaにロード
mulhs	a, n, q	$q = M * n/2^{32}$
(sub	q, n, q	$M > 0$のときこの命令を挿入する）
(sra	q, s, q	$s > 0$のときこの命令を挿入する）
srl	q, 31, t	qが負のとき(nが正のとき)
add	q, t, q	qに1を加算

表6.4: 除数が-2以下の符号付き除算

のときと同じである。3行目のsub命令は$M > 0$すなわち$-2^{32} \leq m \leq -2^{31}$のときに挿入される。これは

$$\frac{(m+2^{32})*n}{2^{32}} - n = \frac{(m+2^{32})*n - n*2^{32}}{2^{32}} = \frac{mn}{2^{32}}$$

とするためである。4 行目の sra 命令は除数が 2 以上のときと同様の理由で $s > 0$ のときに挿入される。5，6 行目は $n > 0$ のときに 1 を足している（式 (6.7))。

剰余算は除数が 2 以上のときと同様で、表 6.4 の命令列の後に

```
smul q, d, t
sub  n, t, r
```

を付け加えればいい。

6.3 実装

6.3.1 はじめに

除算命令の最適化は、COINS にその機能を追加する形で実装する。COINS のバックエンドに追加機能を与えるのフォルダ backend/contrib に、新しく DivOpt.java を追加する。

6.3.2 全体構造

DivOpt.java 内のクラスは、実際に除算の最適化を行うクラス DivOpt と、途中のメソッド間の値の受け渡しに使う Magic, Power の 2 つのクラスを用意した。これらのクラスは以下の要素で構成される。

public class DivOpt	
フィールド	
public Function function	解析する手続き（関数）
public LirFactory newLir	新しい LIR の命令を生成するためのフィールド
メソッド	
public void doIt	除算命令の最適化を実行するメソッド
public void searchDiv	LIR の中から除算、剰余算を探すメソッド
public void changeDiv	除算、剰余算命令の変換を行うメソッド
public Power isPopwerOf2	引数の値が 2 のべき乗か調べ、その結果を返すメソッド
public Magic calcMagic	マジックナンバー M とシフト量 s を求めるメソッド
class Power	
フィールド	
boolean p	2 のべき乗なら true, そうでないなら false となる
int num	2 の何乗かをあらわす
class Magic	
フィールド	
int m	マジックナンバーを表す
int s	シフト量を表す

6.3.3 各段階の説明

除算命令の最適化は、クラス DivOpt の doIt メソッドで実行される。doIt メソッドの内部を順を追って説明する。

1. 除算、剰余算を探す

doIt メソッドでは関数の LIR 1 命令ごとに searchDiv メソッドを呼ぶ。この searchDiv メソッドではその LIR 命令の中に符号付除算、剰余算があるかを調べる。LIR では符号付除算は DIVS, 符号付剰余算は MODS となっている。この DIVS,MODS を発見した場合は実際に除算命令の変換を行う changeDiv メソッドを呼ぶ。今回は符号なし除算については扱わない。

2. 適用条件を調べる

changeDiv メソッドでは、まず除数が整数定数であるかと、演算が整数除算かを調べる。そうでない場合は何もせずに searchDiv メソッドに戻り、再び除算の探索を行う。条件に合った場合は isPowerOf2 メソッドを呼び、除数が 2 のべき乗かどうかを調べる。2 のべき乗でないときは calcMagic メソッドを呼び、マジックナンバー M とシフト量 s を計算する。

3. 除算命令を変換する

2 で調べた場合ごとに LIR の除算命令を、変換する目的コードに対応する LIR 命令に変換する。LIR の式は木構造で表現でき、それぞれのノードは LirNode クラスで表される。COINS のバックエンドの LirFactory クラスには新しい LirNode を作るメソッドが用意されている。このメソッドを用いて変換する目的コードに対応する LIR を生成し、LIR の木の変換する除算、剰余算に対応する部分を新しい LIR に変更する。

前節で説明した生成する目的コードのうち、mulhs 命令は 32 ビットの符号付乗算の結果の（64 ビットになる）上位 32 ビットをとる命令だが、これに対応する SPARC の命令も LIR の命令もない。SPARC の乗算命令 smul の乗算結果の上位 32 ビットは%y という特別なレジスタに入る。そこで LIR に新しく MULH という命令を作り、ターゲットマシンディスクリプション TMD を書き換え、この命令では

```
smul %op1, %op2, %d
mov  %y, %d
```

というコードを生成しするようにした。%op1,%op2 はオペランドのレジスタ、%d は結果を入れるレジスタである。

第7章　実行結果の評価と考察

7.1　実行環境とテストプログラム

実験には改良前と同じ COINS version CVS(Date 2005/10/15 19:55),gcc version4.0.2 を用い、Sun Microsystems の Sun Blade 1000(表 7.1) で行った。

アーキテクチャ	Superscalar SPARC Version 9
プロセッサ種別	750MHz UltraSPARC
プロセッサ数	2
1次キャッシュ	64KB データ、32KB インストラクション
2次キャッシュ	8MB 外部キャッシュ
メモリ容量	1GByte
オペレーティング環境	SunOS 5.8

表 7.1: Sun Blade 1000 の主な仕様

実験では、次の6種類でコンパイルを行った。

no-change	:	COINS SSA 最適化 + 命令スケジューリング + レジスタプロモーション（改良前）
no-bug	:	COINS で上の no-change にレジスタプロモーションのバグの除去を追加
cut-store	:	COINS で上の no-bug にレジスタプロモーションのストア命令の削減を追加
div-opt	:	COINS で C 上の cut-store に除算命令の最適化を追加
gcc-O2	:	gcc O2 オプション
gcc-O3	:	gcc O3 オプション

SSA 最適化のオプションは

prun/divex/cse/cstp/hli/osr/hli/cstp/cpyp/preqp/cstp/rpe/dce/srd3
prun:Pruned SSA 形式への変換、srd3:Sreedhar の方法３による逆変換
cse:共通部分式除去、cstp:条件分岐を考慮した定数畳み込みと定数伝播、
hli:ループ不変式移動、divex:式を３アドレス方式に変換（代入の右辺の演算子はたかだか１つ）
osr:ループの機能変数に関わる演算の強さの軽減と判定の置き換え、
cpyp:コピー伝播、preqp:質問伝播に基づく大域値番号付けと部分冗長除去、
rpe:冗長な Phi 関数の除去、dce:無用命令削除

とし、実験には SPEC CPU 2000 に収録されているベンチマークを用いた。

7.2 実行結果

SPEC ベンチマークの実行結果を表 7.2, 図 7.1 に示す。表 7.2 での SPEC ベンチマークの入力は ref で、実行時間は 3 回の測定の中央値を示している。図 7.1 は A の実行時間を 1 としたときの相対値のグラフである。

	A	B	C	D	E	F
164.gzip	741	741	734	734	540	541
175.vpr	634	632	630	600	490	484
181.mcf	499	490	497	474	460	462
197.parser	767	767	776	764	665	600
255.vortex	705	693	695	684	550	541
256.bzip2	788	765	763	760	487	476
300.twolf	1049	893	869	852	802	800
171.swim	2097	2100	2094	2096	1910	1913
172.mgrid	1738	1675	1680	1683	1940	1936
173.applu	1819	1804	1806	1803	1489	1481
177.mesa	760	760	760	759	654	645
179.art	509	573	580	567	414	420
183.equake	870	868	864	857	856	810
188.ammp	1271	1270	1236	1234	996	947

表 7.2: SPEC ベンチマークの実行結果

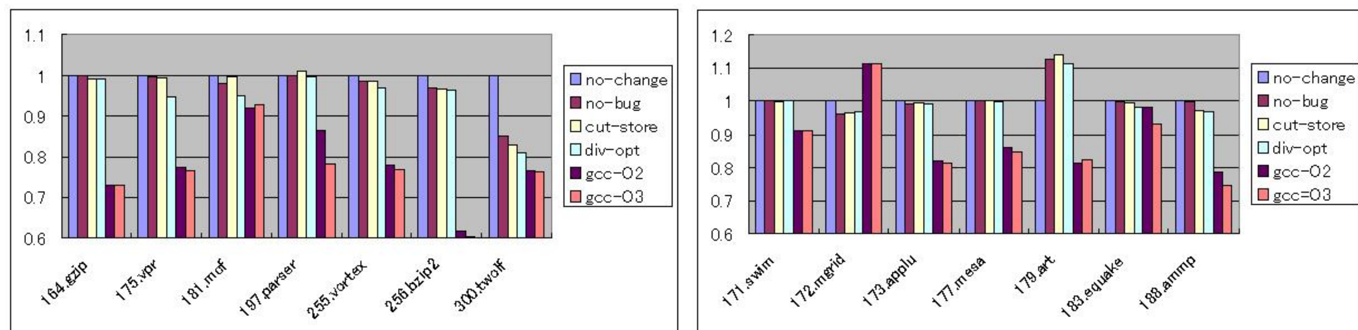

図 7.1: SPEC ベンチマークの実行結果（A との相対値）

7.3 考察

図 7.1 を見るとバグの修正については 256.bzip2,300.twolf では実行時間が速くなっている。特に 300.twolf では 15%近く速くなっている。これは、これらのプログラムではバグの出るような構造が多く、実行時間に影響が出ていたためと思われる。

ストア命令の削減については300.twolf以外ほとんど実行時間に効果がなかった。ストア命令の削減が適用できるような例が少ないか、あまり実行時間に影響の出る部分ではなかったためだと考えられる。

除算命令の最適化は175.vpr,181.mcfではある程度の効果があった。181.mcfではgccのO2オプションよりも良い結果となった。このように、除算、剰余算命令の多いプログラムでは最適化の効果が出る。除算、剰余算を頻繁に使用するプログラムではさらに効果が出ると思われる。

179.artではバグの修正で10%以上実行時間が増えてしまった。これはバグの修正で外側のループに追加されたブロックの中で、グローバル変数が多く使用されており、レジスタプロモーションを行う変数が増えて、レジスタの圧力が上がりすぎ、スピルが起こったため実行時間が遅くなったという可能性が考えられる。

7.4 今後の課題

7.4.1 除算命令の最適化の適用マシンの拡大

除算命令の最適化では、2のべき乗以外での除算のとき、LIRに新しい命令を追加したが、これはまだSPARC-V8以外には対応していない。そのため、他のマシンにも対応するようにしたい。またSPARCのV8より前のマシンにはsmul,sdiv命令はなく、関数呼び出しで乗算、除算を行っている。このときはさらに実行時間改善に効果があると思われるため、関数呼び出しの場合にも対応できるようにしたい。

7.4.2 実行時間に影響の大きい部分の改善

今回の改良では一部のプログラムを除いて余り効果が出ていなかった。これは改良の適用条件が限られていたためと思われる。更なる改良のためには、今回検討した以外のプログラムについて、COINSとgccの目的コードを比較するプログラムを増やすことや、プログラムのどこで実行に時間がかかっているかを、もっと詳しく解析する必要がある。

7.4.3 その他の改善の方向

上の検討に基づきCOINSの最適化の改良には次のような可能性がある。

- HIRレベルでの最適化の改良（関数のインライン展開など）
- SSA形式での最適化の改良
- レジスタプロモーションのアルゴリズムの改良
- 命令スケジューリングの改良
- ソフトウェアパイプライニング
- 命令集合を最大限利用したコード生成

- 除き穴最適化 [11]

これらについて、今後検討していく。

7.5 関連研究

7.5.1 整数除算、剰余算の変換の研究

本研究以外の整数除算、剰余算命令の変換の研究として、符号なし除算に対する最良の命令列生成法として藤波の論文 [藤波 2004] がある [10]。

7.5.2 最適化の組み合わせの研究

最適化の組み合わせを変えて、効率のよい目的コードを出す最適化の列を探す研究として L.almagor らの論文がある [12]。
また、それに関連して、少数レジスタマシンでの最適化の効果をあげる研究も行われている [13]。

7.5.3 ポインタ解析

ポインタ解析を行うことで、レジスタに載せられる変数を増やして、最適化の効果をあげる研究として Steensgaard の論文や、千代の論文がある [14] [15]。

第8章　まとめ

本研究では、COINS と gcc の生成する SPARC の目的コードを比較し、その結果見つかった問題点に対して、

1. レジスタプロモーションのバグの修正
2. レジスタプロモーションでのループ出口のストア命令の削減
3. 整数定数による除算、剰余算命令の最適化の

の 3 つの改良を行った。本研究の改良の結果、一部のプログラムではかなりの実行時間の改善が見られた。しかし効果が出ているものでも少ししか改善されないものもある。

今後の課題としては、今回実装した除算命令の最適化は、SPARC の V8 以降のものにしか対応してない。そのため、そのほかのマシンに対しても対応できるようにしていきたい。また、もっと効果の出る改良を行える部分を見つけられるよう、プログラムのどこで実行時間が多くかかっているかをさらに詳しく解析できるようにしていきたい。

謝辞

本研究を進めるにあたり多大なる御指導ご鞭撻を頂いた、東京工業大学 数理・計算科学専攻教授の佐々政孝先生に深く感謝の意を表します。

また、佐々研究室の皆様にはさまざまな面で助力を頂きました。あらためまして、ここに深くお礼申し上げます。

参考文献

[1] COINS - Project. Coins - project home page. http://www.coins-project.org/.

[2] GCC. Gnu compiler collection home page. http://gcc.gnu.org/.

[3] 狩野祐介. グローバル変数のレジスタプロモーションの実装. 東京工業大学 情報科学科 卒業論文,2005.

[4] Lu, John and Keith D. Cooper 1997. "Register promotion in C programs." In *PLDI '97:Proceedings of the ACM SIGPLAN 1997 conference on Programing Language design and implementation.* ACM Press.

[5] Henry S. Warren, jr. *Hacker's Delight*. Addison-Wesley, 2003

[6] ヘンリー・S・ウォーレン、ジュニア. ハッカーのたのしみ.　滝沢 徹, 鈴木 貢, 赤池 英夫, 葛 毅, 藤波 順久, 玉井 浩 訳, 株式会社エスアイビー・アクセス, 2004.

[7] SPEC. Standard performance evaluation corporation home page. http://www.spec.org/.

[8] SPARC International. Inc. 相越 克久, 田中長光 訳. SPARC アーキテクチャ・マニュアルバージョン 8. 株式会社トップジャパン,1992.

[9] Sun Microsystems. Inc. *Ultra SPARC Cu User's Manual*, 2004. http://www.sun.com/ processors/manuals/US v2.pdf.

[10] 藤波 順久. 整数定数による除算のための最良の命令列生成法,2004.

[11] 佐原聡一郎. 移植可能な覗き穴最適化器の設計とプロトタイプ実装. 東京工業大学 情報科学科 卒業論文, 2005.

[12] L.Almagor , Keith D.Cooper , Alexander Grosul , Timothy J. Harvey , Steven W.Reeves , Devika Subramanian , Linda Torezon and Todd Waterman. Finding Effective compilation Sequences. *Procedings of the 2004 ACM SIGPLAN Conference on Languages, and Tools for Embedded Systems* , pp.231-239,2004.

[13] 今橋孝典. 少数レジスタマシンにおけるスピルを考慮した最適化の組み合わせの研究. 東京工業大学　情報科学科　卒業論文, 2006.

[14] Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. *POPL'96*, pp.32-41, 1996.

[15] 千代英一郎. Deductive System による C プログラムのポインタ解析.　情報処理学会論文誌: プログラミング,Vol.47,No.SIG 2(Pro 28),pp.1-17, 2006.