ハイパフォーマンス
コンピューティング 52−13
（1994. 7. 23）

クリティカルパス情報とSSA形式を利用したプログラム最適化

古関 聰 † 小松 秀昭 ‡ 深澤 良彰 †
† 早稲田大学理工学部 ‡ 日本IBM(株) 東京基礎研究所

コードの最適化手法には、制御の流れに注目したコード変形と、データの流れに注目したコード変形がある。前者は、コードスケジューリング内で行なわれる命令の移動であり、後者は、データフロー解析をして行なわれる共通部分の削除やデータのコピーの削除である。これらの方法は、双対の関係にありながらも個別に適用され、マシーンリソースなどで互いに制約されている部分は考慮されていなかった。本論文では、新たな最適化アルゴリズムを提案し、この技法の上で両者を統合する。本アルゴリズムによって両者が有用な情報を共有でき、より品質のよいコードを生成することができる。

Global Optimization using SSA form and Critical Path of a Graph

Akira Koseki† Hideaki Komatsu‡ Yoshiaki Fukazawa†
† School of Science & Engineering, Waseda University
‡ Tokyo Research Laboratory, IBM Japan, Ltd.

Optimizing compiler includes code transformation based on control flow analysis and one based on data flow analysis. The former is performed in some code scheduling methods as movements of operations. The latter consists of eliminating common subexpressions and data copies. Though both techniques mutually have constraints on machine resources , there have been applied respectively. In this paper, we describe a new method to synthesize these techniques and it can utilize more information to get better code.

1 はじめに

コンピュータの高速化を実現するために、スーパースカラー、VLIW など、命令レベルでの並列処理に関する研究が盛んに行なわれている。命令レベルの並列処理では、コンパイラがマシンの特性に合わせた最適化コードを生成することが重要である。

コードの最適化手法には、制御の流れに注目したコード変形と、データの流れに注目したコード変形がある。前者は、データフローは変形せずに命令列の制御構造を変化させるものである。具体的にはコードスケジューリング内で行なわれる命令の移動がこれにあたる。具体的には、リストスケジューリング [1] における基本ブロック内の移動、トレーススケジューリング [2]、バーコレーションスケジューリング [3] における基本ブロックを越えた移動を挙げることができる。後者は、制御フローは変形せずに、命令列のデータ生成の方法を変化させるものである。具体的には、データフロー解析をして行なわれる共通部分の削除やデータのコピーの削除であり、Value Numbering アルゴリズム [4] や SSA 形式 (Static Single Assignment form)[5] を用いた Rosen らの方法で行なわれている方法である。

これらの方法は、双対の関係にありながらも個別に適用され、マシーンリソースなどで互いに制約されている部分は考慮されていなかった。本論文では、グラフのクリティカルパスと、SSA 形式を利用した新たな最適化アルゴリズムを提案し、この技法の上で両者を統合する。本アルゴリズムによって両者が有用な情報を共有でき、より品質のよいコードを生成することができる。

2 本研究の背景

本研究は、制御フローに基づいたコード変形とデータフローに基づいたコード変形を統合して行なう。ここでいう統合とは、共通のデータ構造に対して、同時に二つの最適化を行なうことを意味している。これにより、両者がこれまで参照しえなかった情報を利用することが可能となる。これらを同時に行なうことによって、次のような利点が得られる。

1) それぞれの最適化は、命令を増加させるような操作を含んでいる。これらを別々に行なった場合、他方の命令増加を考慮にいれないため、一方がマシンのリソースを使い果たしてしまう恐れがある。双方を同時に考慮することによって、このような可能性を排除することができる。

2) それぞれの最適化の操作は、もう一方の操作でより効率よく肩代りできる可能性がある。どちらを適用するかを、同一のフレームワーク内で判定することにより選択できる。

ただし、二つを統合したときにそれぞれの弱い点が影響し、統合によって得た利点を失ってしまわないようにすることが肝要である。また、個々の方法が本質的に持つ弱点を解決するような統合ができればさらに理想的である。

我々は、[1] ～ [5] やその他の手法を吟味することにより、コード変形については以下の点を考慮する必要があると考えた。

1) 高い並列度を引き出すため、大域的なアルゴリズムでなければならない。

2) 本当にプログラムの処理時間が短くなるように、コード変形を行なわなければならない。

3) 変形に際し、逆依存が発生して他の移動ができなくなることがないようにしなければならない。

4) 命令数を増やすような変形に際し、命令を無駄に増やすことのないようにしなければならない。

5) 数値計算のような分岐の方向が偏ったプログラムにうまく対応しなければならない。

6) 非数値計算のような分岐の方向が偏ってないプログラムにうまく対応しなければならない。

以上を実現するアプローチとして、我々は以下のことを考えている。

A) 最適化をインクリメンタルなコード変形アルゴズムととらえ、このアルゴリズムの中に二つのコード変形を取り入れる。

B) アルゴリズムに大域的な尺度を導入して、どの命令が全体の実行にクリティカルなのかを判定できるようにする。

C) 逆依存を排除する形式と方法を導入する。

D) プログラムの実行確率情報を含めたデータ構造を採用し、分岐方向の偏向性に柔軟に対応する。

この具体的な方法として、我々は、プログラム依存グラフ (Program Dependence Graph: 以下 PDG) [6] を改良したガード付き PDG(以下 GPDG) と、SSA 形式を利用した最適化アルゴリズムを開発した。

本手法では、プログラムの最内ループを GPDG で表現し、SSA 形式に変換する。この GPDG を対象として、いくつかのグラフ変換を繰り返し適用することによって、プログラムを並列化する。このグラフ変換は、これまで行なわれてきたコード変形に対応している。以降、本論文では、GPDG の定義、GPDG の変換法などを説明し、我々の手法を評価する。

3 GPDG の定義

3.1 依存関係と PDG

命令の間には、ある命令はある命令よりも後に実行されなければいけないという意味での依存関係が存在する。この依存関係をグラフで表現し、プログラム最適化に利用する研究は数多く行なわれており、Ferrante らの PDG を用いた方法 [6] がその代表例である。我々は、アルゴリズムが用いるデータ構造として、PDG を拡張した GPDG を用いる。GPDG の特

徴は、これまでの方法とは異なり、制御依存を表現するために、ある命令が実行されるかどうかを表すガードを使用することである。ガードを用いることによって、制御依存関係をデータ依存と同様に、ガードの生産者と消費者の関係ととらえることができる。これによって、データ依存と制御依存を一つのグラフの中で統一的に扱うことが可能となる。

また、最適化のために、プログラムを依存関係を用いたグラフで表現する手法は、PDG の他に、Polychronopoulos の方法 [7] がある。本研究で持ちいている GPDG は、制御依存関係を、ガードの生産者と消費者の関係ととらえることにより、データ依存と制御依存を全く区別する必要がない。このことにより、後に述べるアルゴリズムの実装を簡潔に行なうことができる。

3.2 GPDG の構成

3.2.1 依存関係

GPDG は、プログラムの各命令を表すノードと命令間の依存関係を表すエッジから構成される。また、各ノードには付加情報として、ノードが表す命令が生産するデータのディスティネーション、ソース、及び、実行条件となるガードが含まれる。さらに、グラフのトップとボトムを表す START ノードと END ノードを導入する。ここでは、プログラム中に存在する依存関係を次のように分類した。

(1) データ依存

命令 A が生産するデータを他の命令 B で使用（消費）する場合、命令 B は命令 A によるデータの生産の完了を待たなくてはならない。この関係をデータ依存という。また、定義されない変数 (レジスタ) を参照する命令は START ノードとデータ依存関係を持ち、GPDG の中で使用されない変数 (レジスタ) を生産する命令は END ノードとデータ依存関係を持つものとする。

(2) 制御依存

ノイマン型計算機では、プログラムの制御を内部状態の変化によって行なう。この内部状態の変化が決定するまではそれ以降の命令を実行できない。我々は、各命令にガードを付加し、制御依存を実行条件の生産者と消費者の関係で表している。図 1において、括弧のついた部分がガードである。(cc) が付加された命令は、条件レジスタ cc が真ならば、そのときに限り実行される。また、(!cc) が付加された命令は、cc が偽ならば、そのときに限り実行される。

(3) リソース依存

物理的なリソースの制約によって生じる依存関係のことである。逆依存、出力依存はレジスタリソースによって生じる依存関係と考える。

(4) その他の依存

上記の依存関係には当てはまらないが、命令間の実行順序を守らなければならない場合がある。例えば、ジャンプ命令は、その命令より前の命令と順番を入れ変えて実行してはならない。このような場合は、実行順序を守るために補助のエッジを用いる。

これらの依存関係でプログラムを表した例を図 1-a に示す。

3.2.2 SSA 形式の導入

本アルゴリズムでは、逆依存などを取り除くため、SSA 形式を導入する。ここで、SSA 形式への変形を意味する SSA 変換は、以下のことを示すものである。

1) 全ての (プログラム上での) 代入を 1 度しか起こらないようにする。元プログラムで同一変数 (例えば a) に複数の代入が行なわれている場合は、変数に番号をふって (例えば a1,a2,...) 代入先を変える。

2) 上記の変形によってもプログラムの意味が変わらないように、制御の合流点でドミナンスフロンティアと呼ばれる部分に Φ ファンクションを挿入する。

一般に SSA 変換は、コントロールフローグラフに対して行なわれるが、ここでは GPDG に対し変換をかける。Φ ファンクションは、Φ ファンクションによって値を調整される変数からのエッジと、合流が起こるべき分岐に対応したガードを生成した命令からのエッジが張られる。

図 1-a に SSA 変換を施したものを図 1-b に示す。このグラフにおいて、変数、条件変数に関するレジスタのリソース依存 (逆依存) が全くなくなっている。さらに、Φ ファンクションが GPDG 上の必要な位置に挿入されている。

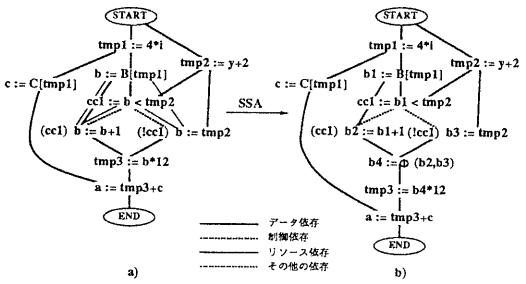

図 1: GPDG の例

4 GPDG の変換によるコード最適化

4.1 大域的な尺度としての GPDG の最長パス

リストスケジューリング (クリティカルパス法)[1] では、基本ブロックの中でデータフローグラフを作成し、その最長パスの情報を利用してコード変形を行なう。我々も、このような尺度を導入することで効率が良く、欠点の少ないアルゴリズムを構築するのが目的である。

我々は、制御依存を、条件レジスタへの代入と参照

として表現することで、データ依存と同様の取り扱いを可能にしている。これによって、ループのないプログラムを、一つの大域的データフローグラフとして取り扱うことができる。我々のGPDGは、分岐も含めた広いコード領域を扱っているので、大域的な指標(GPDG上で定義される最長パス)を持っているといえる。

ただし、GPDGの中には、分岐の概念が取り入れられているので、単なるグラフの最長パスをそのままアルゴリズムを制御する尺度として取り入れることはできない。プログラムに分岐が存在する場合、GPDGは、その分岐に対応するガードが真の部分と偽の部分に分割することができる。分岐が複数あるときは各分岐の組合せによって、やはりGPDGが分割できる。このようにして、全てのプログラムの分岐の組合せに対応したサブグラフを考え、その最長パスを利用する。また、これらの最長パスは分岐確率を持っているので、この分岐確率を基にして整列され、最適化の優先順位が決定される。

我々は、具体的な大域的尺度として、プログラムの実行に必要なマシンサイクル数を指標として用いることにする。マシンサイクル数を見積もるためには、上記の、分岐の組合せで分割されたグラフに対応した最長パスが利用される。

実行サイクルの見積りは、プログラムの条件の組合せについて、その組合せに対応したグラフの最長パスの予測実行サイクル数とその確率の積の総和をとったものを用いている。例として、図1-bの場合を考える。ここでb1<tmp2になる場合をc_1とし、その否定を!で表すとする。ある条件の組合せ$C \in (c_1,c_2,..c_i,..)$に対し、それが起こる確率をP(C)、そのときの最長パス長をL(C)、その予測実行サイクルをS(L(C))とすると、図1-bが表すグラフGの実行速度の見積りEst(G)は、

$P(c_1) \times S(L(c_1))+P(!c_1) \times S(L(!c_1))$

となる。

Pの値は、プログラムのプロファイルが存在すれば、その値を使用する。プロファイルの情報が得られない場合でも、ループからの脱出など、プログラムの性質から分岐の偏りが判別できる場合は、偏りのあるほうの分岐確率を1とする。それ以外は、分岐確率を50%とする。また、Lはグラフをたどることで求めることができる。

L(C)は命令列の実行順序関係を示しているので、ある命令の何サイクル後に次の命令が実行可能であるかの情報がわかれば、S(L)が求められる。ここでは簡単のため、S(L)をLの長さとする。

4.2 グラフの変換に基づいた最適化

変換は、

1 投機的命令移動による変換
2 ノード分割による変換
3 共通部分式の削除による変換
4 単一代入式の削除による変換

からなっており、これらを適用していくことによりプログラムを最適化する。1は制御フローに基づいたコード変形であり、3,4はデータフローに基づいたコード変形である。2は両方を含んだ変形となっている。また、この最適化では、グラフのパスの長さの変化と実行確率を基準として変換が適時駆動される。変換適用のポリシーとしては、

①プログラムの実行時間が短くなるように変換
②プログラム中の命令数が少なくなるように変換

が考えられるが、ここではプログラムの実行時間が短くなることを優先にし、そのなかでも命令数が少なくなるような実装を選んである。

4.2.1 投機的命令移動

命令の中のいくつかは、制御依存を無視して投機的に実行可能である。これを利用して最長パスの短縮を図る。

この操作は、GPDG上では、制御依存のエッジを外すことによって行なわれる。図2は、命令2の制御依存を外す変換の例である。いま、(r7 == 2)が真であった場合の最長パスが、変換前に、1-2-4-5の命令列を含んでいたとすると、変換後の最長パスは、1-4-5を含んだものに変化する。結果として、プログラムは、((r7 == 2)である確率) × (最長パスの減分)に相当する分だけ、実行時間が短縮される。

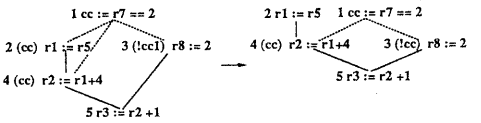

図2: 投機的命令移動

4.2.2 ノード分割

ノードの分割を行なうことによって、新たに投機的な移動を行なう機会を増やすことができる。また、場合によっては実行時間が短縮されることもある。

この操作は、GPDG上では、SSAのΦファンクションを越えて、命令を分割することによって行なわれる。図3は、レジスタ5に(レジスタ4 - 5)を代入する命令を、Φファンクションを越えて移動する例である。ここでは、他と重複しないレジスタとして、レジスタ6とレジスタ7を新たに用意し、これらのレジスタにディスティネーションを変更して演算が行なわれるようにする。この結果は、Φファンクションにより値が調整され、最終的にはレジスタ5に値が格納される。

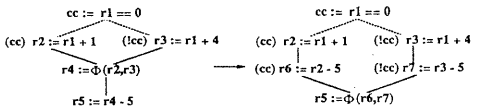

図 3: ノード分割

4.3 共通部分式の削除および単一代入式の削除

事前に計算された値を用いることによって、再計算を防ぎ、最適化を行なうことができる。

この操作は、GPDG 上では、ノードの内容の書換え、エッジの付け換えで行なわれる。例えば、図 4-a) では単一代入式の削除を行なうことによって、パスが短縮される。また、同図-b) では、(4*b1) の共通部分が削除されており、(t1:=4*b1) を含むパスがクリティカルパスでなければ、もとのパスが短縮される。

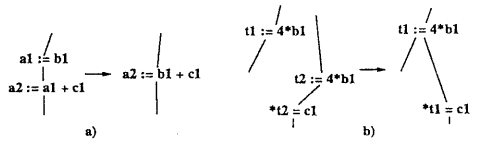

図 4: 事前の計算結果の利用

4.4 グラフ変換の適用法

グラフ変換の適用アルゴリズムを以下に示す。

```
schedule(gpdg)
GPDG gpdg; /* 最適化は gpdg への副作用によって行なう */
{
 GPDG のノード型 node;
 分岐の組合せ型 ccomb;

 while( True ) {
  グラフのパス型 target_path;
  パスリスト型 paths = グラフから全パスを抜き出す (gpdg);
  paths = 実行確率が最も高いもの (paths);

  if (null(paths)) exit;
  if (!unique(paths)) {
   /*  paths が 複数の要素からなっているとき */
   paths = 長さが最も長いもの (paths);
    if (!unique(paths)) {
     パスリスト型 tmp_paths = 投機的実行可な
                              ノードを含むもの (paths);
      if (null(tmp_paths)) {
       /* 任意の一つを取り出す */
       target_path = selectRandom(paths);
      }else{
       if (!unique(tmp_paths)) {
        target_path = selectRandom(tmp_paths);
      } else target_path = tmp_paths[0];
     }
    }else target_path = paths[0];
  }else target_path = paths[0];

  ccomb = 選ばれたパスの分岐の組合せを得る (target_path);

  while(True){ /* パスが短くなるまで */
   GPDG tmp_gpdg;

   if ((node = 投機的移動可ノードのうちの一つ (target_path))
       != NULL) {
    グラフのパス型 tmp_path;
    tmp_gpdg = 投機的移動適用 (gpdg,node);
    tmp_gpdg = 共通部分式削除等適用 (tmp_gpdg,node);
    tmp_path = 分岐の組合せに対応したパス (tmp_gpdg,ccomb);
    if (長さ (tmp_path) > 長さ (target_path)){
     /* コード変形後に長さが変化してない時は変形をとりけす */
     gpdg = tmp_gpdg;
     break;
    }
    else{
     node をこれ以上選ばれないようにする (target_path,node);
    }
   }else if ((node = ノード分割化
     ノードのうちの一つ (target_path)) != NULL) {

    tmp_gpdg = ノード分割適用 (gpdg,node);
    tmp_gpdg = 投機的移動適用 (tmp_gpdg,node);
    tmp_gpdg = 共通部分式削除等適用 (tmp_gpdg,node);
    tmp_path = 分岐の組合せに対応したパス (tmp_gpdg,ccomb);
    if (長さ (tmp_path) > 長さ (target_path)){
      /* コード変形後に長さが変化してない時は変形をとりけす */
      gpdg = tmp_gpdg;
      break;
    }
    else{
      node をこれ以上選ばれないようにする (target_path,node);
    }
   }else{
    /* target_path はこれ以上最適化できない */
    path をこれ以上選ばれないようにする (gpdg,target_path);
    break;
   }
  }
 }
}
```

4.5 考察

前節で紹介したアルゴリズムの特徴をまとめる。

1) GPDG を用いているため、大域的なアルゴリズムが実現されている。

2) 最長パスに基づき命令移動・分割を行なうため、影

響力の低い命令が変形対象となることがない。

3) 分岐に偏りがある場合は、プロファイルのデータにより実行確率の高いパスが優先的に最適化される。そうでない場合は、ターゲットパス群の実行確率は横並びになり、それぞれがパスの長さに合わせて最適化される。

4) 制御フローに基づいたコード変形と、データフローに基づいたコード変形を同時に行なう効果が得られる。

5) 命令の移動、分割をSSA形式の上で行なうため、変形の途中で逆依存等が発生しない。

5 評価

本章では、制御依存に基づいたコード変形と、データフローに基づいたコード変形を同時に行なったことによる効果を示す。

表1,2は、一般的なプログラムを対象としていろいろな最適化を行なった場合の結果である。表中の数値は、最適化をなにも行なわなかった場合の値を1としたときの相対値である。ターゲットアーキテクチャは、並列度4のVLIWとし、TORCH[9]やGIFT[8]のような、投機的実行のためのハードウェアサポートを持っているものとした。

表 1: 最適化結果 (相対実行時間)

プログラム名	fibonacci	prime	max
制御フローに基づいた変形のみ	5.00	2.43	1.96
データフローに基づいた変形のみ	1.54	1.00	1.20
両者を別々に適用	4.00	2.43	1.96
本手法	5.00	3.33	2.39

表 2: 最適化結果 (相対命令数)

プログラム名	fibonacci	prime	max
制御フローに基づいた変形のみ	1.00	1.00	1.00
データフローに基づいた変形のみ	0.65	1.00	0.85
両者を別々に適用	0.65	1.00	0.85
本手法	0.65	1.13	1.00

表1より、実行時間については、本手法が最も良い結果を出している。両者を別々に適用した場合よりも良い値がでているのは、GPDGのクリティカルパスの情報を利用することで、どの命令を移動するのが効率がよいかを得ることができるためである。また、両者を別々に行なった場合、命令数を増やしながらも全体の実行時間を短縮させることが困難であるが、本手法ではこれを効率よく行なうことができている。表2より、本手法では命令数の増加が若干あるが、これは実行時間の短縮を優先させた結果であり、実行時間の短縮の効果にくらべ十分妥当な値が得られていると結論できる。

6 おわりに

本稿では、スケジューリングにおける最適化手法と、共通部分式の削除などのコード変形を統合して行なうアルゴリズムとそのデータ構造を示した。また、本手法は、単に両者を統合しただけではなく、もともと各最適化が持っていた欠点をも同時に解決している。

コード最適化の手法としては、この他にレジスタアロケーションやループ最適化などが存在する。今回示したデータ構造を用いることで、これらの手法との統合を行なう見通しも既についており、現在、評価データの集計を行なっている。

参考文献

[1] E.G.Coffman and Jr.,'Computer and Job-Shop Scheduring Theory', John Willy & Sons (1976).

[2] J.R.Ellis, 'Bulldog: A Compiler for VLIW Architectures',The MIT Press (1985).

[3] A.Aiken and A.Nicolau, 'A Development Environment for Horizontal Microcode', IEEE Transactions on Software Engineering,vol.14, No.5,pp.584-594(1988).

[4] B.Alpern, M.N.Wegman and F.K.Zadeck. ,'Detecting equality of values in programs', Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,(1988).

[5] B.K.Rosen, M.N.Weg-man and F.K.Zadeck, 'Global value numbers and redundant computations', Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,(1989).

[6] J.Ferrante, K.J.Ottenstein and J.D.Warren, 'The Program Dependence Graph and Its Use in Optimization', ACM Transactions on Programming Languages and Systems,Vol.9,No.3, pp319-349,(1987).

[7] C.D.Polychronopoulos, 'Parallel Programing and Compilers', Kluwer Academic Publishers (1988).

[8] 小松, 古関, 鈴木, 深澤, ' 拡張VLIWプロセッサGIFTの命令レベル並列処理機構' , 情報処理学会論文誌, Vol.34, No.12, pp2599-2610,(1993).

[9] M.D.Smith, M.S.Lam, and M.A.Horowitz, 'Boosting Beyond Static Scheduring in a Superscalar Processor', Proceedings of the 17th Annual International Symposium on Computer Architecture,pp.344-354(1987).