アドレッシングモードの有効利用を重視したコード生成系について

小松 秀昭　　門倉 敏夫　　深沢 良彰
（早稲田大学）　　（相模工業大学）

１．はじめに

半導体技術の進歩によって、より高度なアーキテクチャをもったチップレベルのプロセッサが出現している。これらの新しいプロセッサ・アーキテクチャに共通して見られる特徴の１つは、高級言語を指向した機械語をもつことであろう。オペランドに対するアドレッシングモードの強化もその一部であり、命令コードとオペランドに許されるアドレッシングモードの直交性も重視されてきている。さらに、この傾向は今後いっそう強まっていくものと考えられる。

この様な、アーキテクチャの多様化、高度化はRetargetable Code Generation〔1〕に対して基本的に２つの影響を持つと考えられる。１つは、命令コードの高機能化によって、高級言語の演算子とのマッピングが容易になりうること、もう１つは、アドレッシングモードや記憶域管理などのデータへのアクセス法の高機能化によって、コード生成をより複雑なものとしてしまうことである。よって、アドレッシングモードの有効利用は、最適なコードを生成するコンパイラにとって必要条件と考えられる。

代表的なRetargetable Code GenerationシステムであるGraham〔2〕のものでは、レジスタマネージメントの効率的な処理や、アドレッシングモードの有効利用を簡単に実現するのは困難である。また、Cattell のPQCC〔3〕では、コード生成段階の前処理として暫定的なレジスタ割り付けを行っており、その割り付けで失敗した場合には、コード生成部でsub-targettingアルゴリズムを用いて回避している。この方法は、アドレッシングモードの有効利用という点から見れば、積極的な解決法とは言えない。

本システムのコード生成部では、命令コードとアドレッシングモードを分離して、２レベルのパターンマッチングを行う。これによって、アドレッシングモードをより有効に利用し、このような新しいアーキテクチャの特徴を吸収することを目的としている。また、アドレッシングモードを有効に利用するために必要なレジスタマネージメントの処理を、コード生成前とコード生成中の２段階にわたって行うことによって、より効率的なものとしている。

２．本システムの概要

本システムは、パーサ生成部、最適化部、コード生成部、命令テーブル生成部より構成される。　（図１）

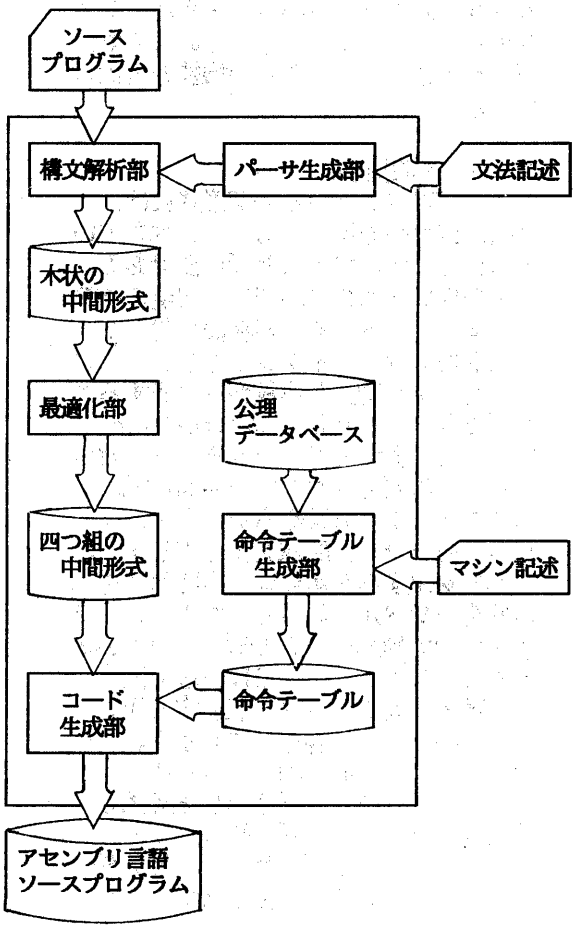

図１：　本システムの構成図

2－1．パーサ生成部

パーサ生成部は、ソースプログラムに対する文法及びアクション、各ターミナルシンボルに対するエラー回復コストを入力として、LALR パーサ(4) である構文解析部を自動生成する。

本システムのパーサ生成部の特徴として、生成されたパーサのエラー回復能力の向上が挙げられる。エラー回復のアルゴリズムは、基本的にはlocally least-cost error correction algorithm (5) を用いたものである。

また、本システムのパーサ生成部ではLL(1)文法の文法記述に対してconflictを起こさないようにするために、識別子の動的なクラスづけを可能にしている。この問題は、レキシカル・アナライザのレベルで解決することも可能であるが、本システムでは効率の点から専用のレキシカル・アナライザを用いているので、パーサ生成部にこのための機能を付加している。

2－2．中間形式について

コンパイラの用いる中間形式には、三つ組、四つ組を用いるものと、木を用いるものとに大別することが可能である。

木を用いるものの利点は、表現の柔軟性であり、目的とする機械語に対して、極大な部分木をマッチングさせることによって、より強力な機械語を生成することができる。しかし、Retargetable Compilerにおいては、マッチングの対象となる機械語のレベルまで中間形式のレベルを下げなければならないため、Retargetability に問題を生じてしまう。これを解決するためには、中間形式を比較的高いレベルに設定しておき、コード生成段階で動的に木を変形していく必要がある。しかし、これでは一般の演算子と変数の実効アドレスを記述する演算子が一つの木の中に混在してしまう。すなわち、ノードを統一的に扱うため、命令コードとオペランドを分離した形式での効果的なマッチングが、難しくなってしまう。このことは、命令コードとアドレッシングモードの直交性を利用しにくくしてしまう。

そこで我々のシステムでは、命令コードとオペランドを分離した形式でのマッチングを行う。そのためにコード生成部の入力の中間形式として、アドレス計算を行う演算子に関して、若干の拡張を加えた四つ組を用いることにした。この拡張部分は、変数のアクセスを記述するものであり、中間形式のレベルの下げずにアドレスの概念を導入している。

四つ組は、木に比べてグローバルなデータフロー解析を行いやすい反面、表現の自由度は劣るため、四つ組のレベルの設定を注意して行わないと、マシン依存性が強くなりやすい。このため本システムの中間形式としてもちいる四つ組は、マシン依存性がないレベルまで上げられている。

しかし、中間形式としていきなり四つ組を生成するパーサでは、四つ組の生成効率に関して問題がある。これは、四つ組が、表現の柔軟性に欠けるため、最適化効率に問題があるからである。また、パーサも木を生成する方が一般に容易であると考えられる。

そこで、本システムでは、各種最適化に対して、効率の良いデータ構造をとるために、最適化部への入力として、木状の中間形式をとることに決定した。

図２に本システムが用いている四つ組の例を示す。図２において、四つ組のオペランドに現れるINDEX1からINDEX3までの変数は、アドレスデータの一時変数である。

```
DCL   1   DATA( 100 ),
          2   NAME          CHAR(8),
          2   CODE( 3 )     BIN FIXED( 15,0

A = DATA( I ).CODE( J ) + B;

/*PL/Iで記述した構造体要素へのアクセス*/


@ADRS_ARY     DATA      I        INDEX1
@ADRS_ARY     CODE      J        INDEX2
@ADRS_STR     INDEX1    INDEX2   INDEX3
@IADD3        INDEX3    B          A

/*  対応する四つ組の列  */
```

図２：高級言語と四つ組の比較

2－3．最適化部

まず入力である木状の中間形式に対して、ループ展開や融合などの最適化を行う。次に基本ブロック分割を行い、フローグラフを生成する。さらに、基本ブロック内の木状の中間形式を四つ組へ変換する。この四つ組よりDAG を構成して局所的な共通部分式の削除や式の評価順序の決定などを行う。

また、四つ組の中間形式に対して、各種のデータフロー解析の結果により、ループに関した最適化や大域的な共通部分式の削除、複写の伝搬、コードの巻き上げなどの最適化を行う。さらに、各種最適化の後で、基本ブロックを再構成する。

データフロー解析の結果として得られた、変数のライフタイムの情報や、基本ブロックのループのネスト数などを、コード生成部に渡す。これは、コード生成の際に行われるレジスタマネージメントに用いる。

```
CPU = MC68000;
CLASS;
  MEMORY( 0:16777215 )( 7..0 );
  REGISTER = R( 0:16 )( 31..0 ) , SR( 15..0 );
  NAME  R( 1 ) = 'D0' ,
        R( 2 ) = 'D1' ,
                 :
DATATYPE;
  INTEGER  ( 7..0 );
  INTEGER  ( 15..0 ) : BOUNDARY 2;
                 :
ADDRESSING;
  1  DR_DIRECT       : $1 = ( 0:7 );
                       EA = R( $1 );
                       FORM = 'R( $1 )';
                       COST = ( 0,6 );

  11 AR_IND_INDEX_W : $1 = ( 8:15 );
                       $2 = ( 0:15 );
                       $3 = DISP( -128:127 );
                       EA = @( R( $1 ) + %R( $2 )( 15..0 ) + $3 );
                       FROM = '$3(R($1),R($2)';
                 :
INSTRUCTION;
  ADD_W :  'ADD.W  $1  $2';
           TYPE ( $1,$2 ) INTEGER( 15..0 );
           {
              $2 = $2 + $1;
           }
           ADDRESSING $1 ( 1,2,3,5,8,10,11,13,14,15 );
           ADDRESSING $2 ( 1 );
           CC ( C,V,Z,N,X=C );
           COST = ( 4,4 );
              :
              :
END;
```

図３：MC68000 のマシン記述の一部

2－4．命令テーブル生成部

命令テーブル生成部は、ターゲットマシンのマシン記述を入力として命令テーブルを生成する。マシン記述には以下の情報を含んでいる。

・メモリ、レジスタなどの記憶域の記述
・自動記憶域に対する記述
・コンディションコードに対する記述
・アドレッシングモードの記述
・命令コードの記述
・ユーザマクロの記述

記述は，手続的でなく宣言的なものとなっている。例としてモトローラMC68000 に対するマシン記述は、約１５００行程度となっている。その一部を図３に示す。

記憶域の記述には、マシンが扱うことのできるデータタイプや、それらが記憶域の中でどの様に表現されるかも含んでいる。

自動記憶域に対する記述には、スタックポインタとして使用するレジスタの記述や、スタックフレームの取り扱いなどが含まれる。

また、アドレッシングモードと命令コードの記述の中には、時間のコスト、空間のコスト、及びアセンブリ言語での表記なども含んでいる。この時間と空間のコストは、コード生成の際の評価関数のパラメータとなる。

コード生成部の入力となる四つ組にマッチングがとれない場合、等価変換公理によってマッチングテンプレートを生成する。

本システムが用いている公理データベースは、現段階では四つ組と木の等価変換を行うだけの機能しかもっていない。そのため、本システムの能力ではコード生成できない四つ組に対応するマッチン

グテンプレートに対しては、ユーザマクロの入力を促す。

2－5．コード生成部

コード生成部は、入力となる四つ組と命令テーブルに含まれたマシン情報をパターンマッチングして、そのマッチングの結果をアセンブリ言語のソースプログラムの形式で出力する。

本システムのコード生成アルゴリズムは、入力となる四つ組に対して、2レベルのパターンマッチングを行うものである。それは、アドレッシングモードに対するパターンマッチング、及び命令コードに対するパターンマッチングである。

それぞれのレベルで、複数のマッチングに対して、最適なコードを生成するために評価関数によるコスト計算を行う。

このアルゴリズムの概略を以下に示す

入力となるのは、基本ブロックに分割された四つ組である。まず、アドレス演算子をもった四つ組及び、各四つ組のオペランドを、それらに対応した木の表現であるアドレッシング木に変換する。このアドレッシング木に対してターゲットマシンのアドレッシングモードとのパターンマッチングを行って、コストの最も低いアドレッシングモードを採用する。このアドレッシング木は、複数の四つ組から構成することも可能である。

もし、アドレッシングモードで表現可能でないものがあったら、データへのアクセスのコストが最も低くなるようなアドレス計算を行うコード列を生成して、最終的に使用するアドレッシングモードを登録する。この結果には、アドレッシングモードの実現に必要となるレジスタも含まれる。この段階では、十分なレジスタマネージメントを行うことは不可能であるから、レジスタをクラスに分割し管理しておき、最終的にコードが生成される段階でこのクラスのうちから、実際に使用されるレジスタが選ばれる。

次の段階で、これ以外のすべての四つ組に対して、命令コードとのパターンマッチングを行う。命令コードは、演算の種類ごとにコストの低い順にソートされて命令テーブルに入っており、この順序に従ってマッチングが行われる。命令コードとすべてのオペランドの情報がマッチしたとき、目的コードが出力される。

マッチングに失敗した場合には、オペランドの中でアクセスコストの高いものを、アドレッシングモードを使用してアクセスできるように、前述の順序に従ってマッチングを行う。さらにマッチングに失敗した場合には、対象となるオペランドとして、よりコストの低いものを選び、マッチングをくり返す。また、この段階でマッチングが複数個成功した場合には、その中で最もコストの低いものを採用する。

3．コード生成部の構成

コード生成部は、アドレッシングモードパターンマッチャ、インストラクションパターンマッチャ、マクロプロセッサレジスタマネージャの4つの部分により構成される。

3－1．アドレッシングモードパターンマッチャ

アドレッシングモードパターンマッチャは、n個の四つ組とアドレッシングモードのパターンマッチングを行うものであり、次の処理から構成される。

- アドレス演算子及び、オペランドからアドレッシング木への変換
- アドレッシング木とアドレッシングモードとのパターンマッチング
- マッチングに失敗した場合は、アクセスコストが最少であるアドレス計算を行うコード列の生成

アドレス演算子及び、オペランドからアドレッシング木への変換は、一定の変換規則に従って行われる。

図2のように、複数の四つ組で変数のアクセスを記述するような場合は、前に変換したアドレッシング木をむだにしないように、新しいアドレッシング木を構成する。（図4）

さらに、前のマッチング結果をむだにしないように、アドレッシングモードのテンプレートに部分木の情報を加えてある。

マッチングに失敗した場合に生成され

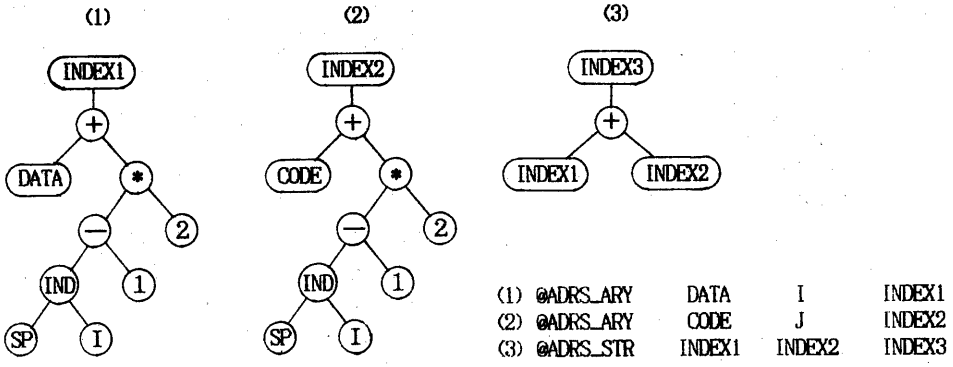

図4：アドレッシング木の例

るアドレス計算を行うコード列は、確定したものではなく、実際に確定するのはレジスタマネージャがレジスタ割り付けを終了した後である。

3—2．インストラクションパターンマッチャ

インストラルションパターンマッチャは、n個の四つ組とn個の命令コードとのパターンマッチングを行う。ただし、n個の命令コードとは、命令テーブル生成部が、公理データベースより生成したマッチングテンプレートを意味する。

ここでは、次の処理を行う。

- 一般の演算子に対しての命令コードとのマッチング
- マッチングに失敗した場合には、コストを評価し、高いものから順にアドレッシングモードとして吸収できるまでのマッチングのくり返し

このコストとは、演算全体のコストであり、変数のアクセスコストも含めたものである。

アドレッシングモードと命令コードに直交性がないためにマッチングに失敗した場合には、別の使用可能なアドレッシングモードを指定してアドレッシングモードパターンマッチャを呼び出す。この結果、新しいアドレッシングモードを使用した変数のアクセスのために、レジスタ割り付けで確定した以外のレジスタが必要になる可能性がある。そのために、レジスタのロードとストアのために必要となるコストも評価コストに加える。

3—3．マクロプロセッサ

マクロプロセッサは、n個の四つ組とn個の命令コードのパターンマッチングを行う。n個の命令コードとは、ユーザマクロを意味する。

本システムのマクロプロセッサには、アドレッシングモードパターンマッチャを自動的に呼び出す機能がある。この機能を利用することによって、ユーザは、マクロ記述に変数のアクセス方法や、レジスタマネージメントに関して、何の考慮もする必要がなくなる。また、これによって、システムにとって、レジスタの選択に対して自由度を増すため、より効率の良いコードを期待することが可能となる。

さらにこのことは、マクロのマシン依存性を、アドレッシングモードやレジスタなどの下位レベルの部分に関して分離できることを意味している。よって、公理データベースでは、純粋に演算子だけに注目すればよいことになり、変換公理を簡略化することができる。

図5は、ある仮想マシンに対する、ユーザマクロの記述例である。まず、マクロ名の宣言の後に、対応させる四つ組の並びが続く。次に、マクロパラメータの型宣言とマクロ本体の記述を行う。

この例は、この仮想のマシンに対して整数の剰余を求める命令を記述したものある。このマシンでは、剰余命令の結果は、レジスタの1番に格納されると仮定している。もし、このマクロが呼び出されたときに、四つ組MOD の3番めのオペランドが、レジスタ1番に割り当てられているなら剰余命令を生成し、そうでな

```
MACRO
  MOD : IDIV @1 @2 @3
        IMUL @1 @2 @3
        ISUB @1 @2 @3
  /* PARAMETER */
        TYPE @1    INTEGER( 15..0 );
        TYPE @2    INTEGER( 15..0 );
        TYPE @3    INTEGER( 15..0 );
  /* MACRO BODY */
    BEGIN
      IF @3 = R(1)
      THEN GEN( 'MOD', @1, @2);
      ELSE BEGIN
          REGISTER( 1 );
          GEN( 'MOD', @1, @2 );
          GEN( 'MOVE', @3, R(1) );
      END;
      RESULT( @3 );
    END;
END
```

図５：ユーザマクロの記述

ければ、レジスタ１番を利用可能な状態にして、剰余命令を生成し、レジスタ１番に残っている結果を、四つ組MOD の３番オペランドに格納することを意味している。また、最後のＲＥＳＵＬＴ文は、このマクロで定義された演算が実行された時の結果の格納場所を示している。

本システムが用いているマクロ処理には、次の様な有効性がある。

- ターゲットマシンが、扱うことのできないデータ型に対する演算や、それらを用いた制御命令に対して、本システムのコード生成アルゴリズムでは、コード生成を行うことはできない。これを、マクロ処理によって補うことができる。
- 入出力処理のような、環境に依存する命令に対してコード生成を行うことができる。これによって、ＯＳとのインターフェースをとることもできる。
- マシン固有の特殊な命令を用いたコード生成を行うことができる。

本システムのマシン記述は、各アーキテクチャに対して記述をおこなえるように、汎用性を考慮しているが、これからどのような命令をもったマシンが出現するかは不明である。そのような命令に対しても、このマクロ処理によってコード生成を行うことが可能となる。

３－４．レジスタマネージャ

本システムのレジスタマネージメントは、基本ブロック中での変数のライフタイム情報及び、基本ブロック間での変数のライフタイム情報によるグラフのカラーリングアルゴリズム(6) を用いたものである。しかし、実際のアーキテクチャでは、アドレッシングモードとレジスタの直交性は決して完全ではなく、さらに実際にコードを生成してみるまでは、変数のアクセスにどれだけのレジスタを必要とするかはわからない。よって、かなりの部分をヒューリスティックに解決しなければならない。

そこで、本システムのレジスタマネージメントでは、まずコード生成の前処理として基本ブロック間での変数のライフタイム情報及び、各基本ブロックのループのネスト数によって、できるだけ少ないレジスタを割り付ける。次に、基本ブロックのコード生成を行う直前に、基本ブロック中での変数のライフタイム情報によって、レジスタを割り付けておく。さらに、アドレッシングモードとのマッチングの結果、必要となるレジスタをクラスに分けて管理する。そして、実際に用いられるレジスタの決定を、命令コードとのパターンマッチングの前の段階まで遅らせる。ここで再び、アドレッシングモードを実現するための一時変数を含めた変数のライフタイム情報によって、レジスタを割り付け直す。これにより、効果的なレジスタマネージメントを行っている。

また、レジスタマネージメントは、ユーザマクロのチェックの後に行われる。これによって、ユーザがある特定のレジスタを指定した場合でも、レジスタマネージメントの情報にそれを含めることが可能である。

４．本システムの問題点

本システムがもっている基本的な欠点は、コード生成の前の段階で、マシン依存の最適化を行っていないことである。

つまり、アドレッシングモードと命令コードの間に直交性を持たないアーキテクチャのマシンに対して効率の悪いコードを生成してしまう可能性が存在する。

例えば次のような四つ組において、INDEX1が加算命令では、アドレッシングモードとして吸収可能であり、除算命令では、アドレッシングモードとして吸収不可能だとすると、本システムによる生成コード列は以下のようになる。

加算に対するコード
INDEX1を計算するコード
除算に対するコード

より効率の良いコード列は、以下のようになる。

INDEX1を計算するコード
加算に対するコード
除算に対するコード

この方が加算命令で用いるアドレッシングモードのコストが低いため、より効率のよいコードである。

さらに、本システムで用いているレジスタマネージメントアルゴリズムは、命令コードとのパターンマッチングの前にレジスタ割り付けを行うため、選ばれたアドレッシングモードが利用可能でなければ、レジスタ割り付けが最適でなくなってしまう。

このような問題を解決するためには、バックトラックしてコード生成をやり直すなどの対策が必要かと思われる。

また、本システムのその他の問題点は公理データベースを用いたマッチングテンプレート合成能力が低いことである。現段階の等価変換公理には、マシン記述と四つ組の対応や、単純な代数や、ブール代数などを採用しているが、これだけでは、ほとんどのマシンに対して完全なマッチングテンプレートを合成することは不可能となっている。それゆえ、マシンの扱うことのできない演算子や、データ型に対して、ほとんどの部分をユーザマクロに依存している。

参考文献

(1) M. Ganapathi, C. N. Fischer & J. L. Hennessy : "Retargetable Compiler Code Generation", ACM, Computing Surveys, Vol. 14, NO. 4, December 1982.

(2) S. L. Graham 他 : "An Experiment in Table Driven Code Generation" ACM, Proceedings of the Sigplan 82' Symposium on Compiler Construction, pp. 32-43.

(3) R. G. G. Cattell : "Formalization Automatic Derivation of Code Generators", Computer Science : Systems Programming No. 3, UMI Research Press, 1982.

(4) A. V. Aho & J. D. Ullman : "Principles of Compiler Design", Addison-Wesley Publishing Company, 1977.

(5) B. A. dion : "Locally Least-cost Error Correctors for Context-free and Context Sensitive Parsers", Computer Science: Systems Programming No. 14, UMI Research Press, 1982.

(6) G. J. Chaitin : "Register Allocation & Spilling via Coloring", ACM, Proceedings of the Sigplan82' Symposium on Compiler Construction, pp. 98-105.