SIMDマシンで並列実行させる同型命令列の認識方式

渡邊 坦 (電気通信大学 情報工学科)
森 教安、菊池純男 (日立製作所 システム開発研究所)

マルチメディア技術の進展に伴い、SIMD方式で並列実行されるマルチメディア向き命令が汎用プロセッサに組み込まれるようになった。マルチメディア処理では、8ビットや16ビットの複数データの組に対して同じ処理を行うことが多く、オペランドの一部が異なる同型命令列の並列実行が重要になる。そこで、並列実行の候補となる同型命令列を認識するアルゴリズムを考案し、実験確認した。それは、木構造中間語の構造的な同型性をハッシュ法で探すもので、複数の基本ブロックにわたる命令列に対しても、同型性を容易に認識できる特徴を持つ。また、特定のSIMD命令列に合わせたライブラリと同型の命令列も、他の場合と同じ処理で認識できる。

Recognition of homologous operation sequences to be executed in parallel on SIMD machines

Tan Watanabe (Computer Science, Univ. of Electro-Communications)
Noriyasu Mori, Sumio Kikuchi (Systems Development Laboratory, Hitachi, Ltd.)

Multimedia processing is prevailing and general purpose machines are emerging that have equipped with SIMD instructions to accelerate such processing. Multimedia processing often requires to execute the same operations on multiple date of short length, and it is important to parallelize homologous operation sequences that are the same to each other except for some part. An algorithm to recognize such homology is presented and confirmed by experiment. It recognizes structural homology of tree intermediate language by hash method. Homologies expanding on multiple basic blocks and homology with complicated operation sequences corresponding to some specific SIMD instructions can be easily recognized.

1. はじめに

多くの並列プロセッサ方式が研究されているが、実用化の先行している方式として、SIMD (Single Instruction Multiple Data Stream) マシンがある[1]。並列に動くプロセッサが数個の場合は、スーパースカラマシンとして、ハードウェアによる自動並列化が広く行われているが、プロセッサ数が4から10前後の場合は、応用分野を限定すると、SIMD方式が効果的に使える。マルチメディア向けプロセッサや、信号処理プロセッサなどでは、SIMD方式を採用しているものが多い。最近は、汎用プロセッサに、マルチメディア向き命令として、SIMD型並列実行命令が組み込まれるようになってきた[2, 3, 4]。

SIMDマシンでは、同じ命令を複数のデータに対して並列に実行する。データが規則的に並んだ配列の場合、この方式は非常に有効であり、ベクタープロセッサとして大きい成功を収めている[5]。

マルチメディア処理では、配列ばかりでなく、複数の構造体要素やスカラ変数に対して、同じ命令列を実行することが多い[7]。例えば、画像データをR, G, B, A要素に分けて表現した場合、INTERPOLATEをインライン展開される補間関数として、

```
r0 = INTERPOLATE(r0, buf[i].r, a0);
g0 = INTERPOLATE(g0, buf[i].g, a0);
b0 = INTERPOLATE(b0, buf[i].b, a0);
a0 = INTERPOLATE(a0, buf[i].a, a0);
```

のように、r, g, b, a の各要素に対して、同じ操作をする処理が高頻度で実行される。このように、異なるデータに対して同じ操作を実行する部分をコンパイラで並列化しようとすると、同じ操作を行う同型命令列を自動的に認識しなければならない。それが機械語の1命令に相当するような場合は認識も簡単であるが、一般には、同型命令列は数個から数十個の命令の列で表現され、途中に分岐を含むこともあるので、その認識は単純ではない。本稿では、このような同型命令列をコンパイラで認識する方法について述べる。

2. 同型命令列を認識する幾つかの方法

2.1 問題の分析

(1) 同型命令列の形態

ソースプログラムにおいて、同型命令列は、配列要素に対する反復命令や、幾つかの構造体要素に対する同じ命令の列、同じマクロを異なるデータに適用する命令の列、同じ名前の手続き参照の列などとして記述される。

配列要素に対する反復命令の並列化に対しては、ベクタープロセッサで開発された方法が利用できるので[5, 6]、ここでは詳述しない。

構造体要素に対する同型命令列では、データごとに異なるのは、

```
image[i].texR[j] = p[k];
image[i].texG[j] = p[k+1];
image[i].texB[j] = p[k+2];
```

のように、要素名やポインタ名、変数名、または添字式である。同型であることの認識のためには、変数表記において、ある特定部分の違いを無視すると、他は同じであることを認識すればよい。

(2) マクロ表現の場合

マクロで表現された同型命令列は、同じマクロ名が使われている点に着目すればよいので、一見容易そうにみえるが、CやC++のマクロは文法上の非終端記号に相当するある式や文として書かれるとは限らず、それのみでは構文解析できないかもしれない。例えば、

```
#define SWITCH(p) {int _p; _p = (p); if(0)
#deinfe CASE(q, s)  else if(_p == (q)) s
#define END  }
....
SWITCH(p->a.x);
CASE(b.x, z = e.x+1);
CASE(c.x, z = f.x+1);
END;
```

のようなマクロを利用したものでは、個々のマクロを解析しても無駄である。マクロ定義の中にマクロが使われていると、問題はさらに複雑になる。また、マクロの段階で対象を識別したとしても、それをコード生成の段階まで一つの固まりとしてまとめておくのは、途中の全部のフェーズでそのことを考慮しなければならず、厄介である。

(3) 条件判定を含む場合

幾つかのSIMDマシンは、条件実行機能を持っている。その場合、同じ命令を全プロセッサに送っても、条件フラグが真になっているプロセッサでのみ実行が行われる。例えば、

```
#define CLAMP(a, b, c) ¥
    ((a) <(b) ? (b) : ((a) > (c) ? (c) : (a))
....
```

```
s0 = CLAMP(s0, 0, 32767);
s1 = CLAMP(s1, 0, 32767);
s2 = CLAMP(s2, 0, 32767);
s3 = CLAMP(s3, 0, 32767);
```

のように条件判定を含む式であっても、条件実行機能を使えばSIMD方式で並列実行できる。並列実行の対象が条件判定を含む場合は、複数の基本ブロックにまたがる同型命令列の認識が必要になる。

（4）関数参照の場合

関数参照の列も、インライン展開すると並列実行の対象となる場合がある。関数参照のままで認識する場合は、関数名が同じで引数が異なることに着目すればよい。インライン展開のあとで認識する場合は、関数の定義内容が分岐を含むこともあるので、上と同様に、複数の基本ブロックにまたがる同型命令列を認識しなければならない。

（5）同型命令列の入れ子

複合された式や文が同型命令列であれば、その中の部分式も、一部を除いては互いに同型命令列である。その場合、部分式の並列化よりも全体の並列化を考える必要がある。すなわち、使えるプロセッサ資源の許す範囲で、できる限り大きい単位で並列化することが望ましい。

2．2　同型命令列認識方法の選択

並列実行の対象となる同型命令列の認識時点として、マクロ展開時や、構文解析時、オブジェクト最適化時、あるいはコード生成時が考えられる。マクロ展開時では、マクロ名の認識は容易でも、先に述べたような多くの問題があるので、同型性の認識時点としては不適切である。コード生成時だと、共通式削除などによって、同型性がくずれていることが多い。並列実行できるコンピュータでは、並列化で実行時間短縮ができれば、同じ演算を複数のプロセッサで行ってもよいので、通常の最適化処理におけるような、実行命令数の削減が必ずしも良い結果を生むわけではない。

従って、同型命令列を認識するのは、構文解析して中間語に変換したあと、オブジェクト最適化の一環として行うのが良い。

中間語の形態として、木構造や、3つ組、4つ組、あるいは抽象機械の命令等がある。オブジェクト最適化を行うコンパイラには、4つ組を中間語とするものも多い。命令が全て同じでオペランドのみが異なるタイプの同型命令列は、4つ組で表現されていても容易に認識できる。しかし、一部の命令が異なって他は同じ命令が並ぶ同型命令列の認識は、4つ組のような1次元的命令列では不便である。とくに、複数の基本ブロックにわたる同型命令列の認識は難しくなる。

木構造を採用すれば、ある部分木の違いを無視すると他は同じであるということを判定するのは容易である。木の構造を表現するハッシュ関数を使うと、複数の基本ブロックにわたる同型命令列の認識にも困難はない。そこで、木構造中間語における同型命令列をハッシュ法で認識する方法を考案し、具体化した。

3．　同型命令列の認識アルゴリズム

3．1　概要

ここに述べる木構造中間語は2分木であり、オペレータノードとしてのノンリーフに、オペランドとしての部分木やリーフがつながる。オペレータは算術演算子や、if, while, 添字修飾subs、構造体修飾qual、ポインタ修飾arrow などであり、リーフは単純変数や定数、ラベルなどである。その木構造は、ソースプログラムの論理構造を表現するばかりでなく、操作の違いも表現するようにしてある。同じ変数であっても、右辺値か左辺値かを区別し、同じ式であっても、添字式か一般の式かを区別する。データ型の違いはノードやリーフの属性として表現する。例えば、代入文

```
g[i]->b.y = a[3];
```

は図1のように表現する。

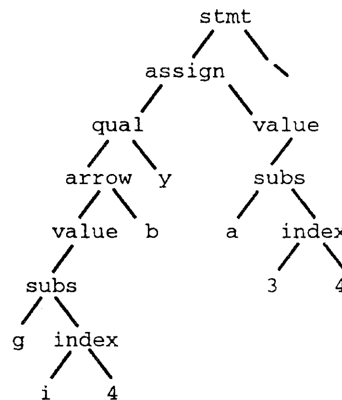

図1　中間語表現の例（g[i]->b.y = a[3];）

ここで、indexの第2子の4は、配列要素が4バイトであることを表す。

実行文に対する木の各ノードにハッシュ・コー

ドを対応させ、ハッシュ・コードの一致するノードがあれば、それで始まる部分木の構造が一致しているか否かを調べる。ハッシュ・コードは、ノードのオペレータとその型、子供達のハッシュ・コードから合成する。型の一致を考慮するのは、複数のデータに対して同じ処理を行う場合、データ型が一致していないと支障があるからである。言うまでもなく、末端オペランドのみ一致してオペレーション・ノードの異なるものは同型とみなさない。同型命令列の認識では、次の3つの場合を考える必要がある。

（1）末端オペランドの同型判定は型の一致までにとどめる

木の構造とオペレータ、末端オペランド（リーフ）のデータ型が一致すれば、末端の変数名や要素名、定数値等は異なっても、同型命令列とみなす。図2の p.x+c と p.y+d は、対応する部分（xとy、cとd）の型がそれぞれ一致すれば同型である。

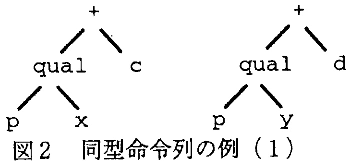

図2　同型命令列の例（1）

（2）変数部分木の違いを無視する

オペランド変数が添字やポインタ、構造体などで各個別々に修飾されていても、型が同じであり、かつ、他の部分が同じであれば、オペランド変数を事前に評価すると同型命令列となる。この場合、変数修飾を意味するオペレータがあると、その部分木のハッシュ・コードを型のみで定まるある値に設定するならば、その同型命令列認識も上記と同様の方法で可能である。図3では、p.x*c と q.t.x*d の部分木の構造は異なるが、対応する部分の型が一致すれば、両者は同型である。

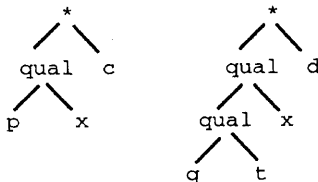

図3　同型命令列の例（2）

（3）一部の部分木の違いを無視する

変数とは限らないあるオペランドの部分木に違いがあるが、それを無視すると他の部分については一致する場合、そのオペランドを事前評価すれば同型命令列となる。従って、2つの部分木において、オペレータが同じであり、一方の子供が互いに同型で、他方の子供の型が互いに同じならば、この2つの部分木は同型とみなす。図4の a[i]*c と b[j+2]*d は、このような条件を満たしていれば同型である。

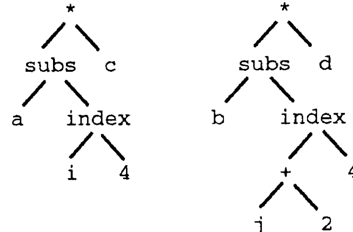

図4　同型命令列の例（3）

並列実行の対象となる命令列には、クリッピングなど、特定の形をしているものが多数ある。これらの中間語形式とSIMD命令形式をイディオムとして登録しておき、中間語にそれらと同型なものが現れると登録内容に従ってSIMD命令に変換すれば、複雑な解析を要するような命令列でも容易に並列化できる。イディオムと同型であるか否かの判定も、上記と同じ方法ででき、認識上の違いは事前に登録してあるか否かにすぎない。一部の部分木の違いを無視する方法でイディオム認識をすると、複雑な命令列の並列化も容易である。

3．2　データ構造

同型命令列認識のために、以下のデータ構造やそのリストを作る。

HomoExp（同型記述子）：
　（同型な各部分木ごとの情報）
　部分木へのポインタ。
　同型な他の部分木との相違部分へのポインタ。
　他。

Homolog（同型リスト記述子）：
　（互いに同型な部分木をつなぐリスト）
　同型として括られる部分木の先頭のものに対する同型記述子へのポインタ。
　この部分木を直接に含む部分木がまた同型であれば、その同型リストへの親ポインタ。
　左右の部分木の同型リストへのポインタ。
　他。

ハッシュ表

部分木へのポインタ。

同型リスト記述子へのポインタ。

いずれにおいても、対応するポインタがなければNULLを設定する。これらの相互関係を図5に示す。

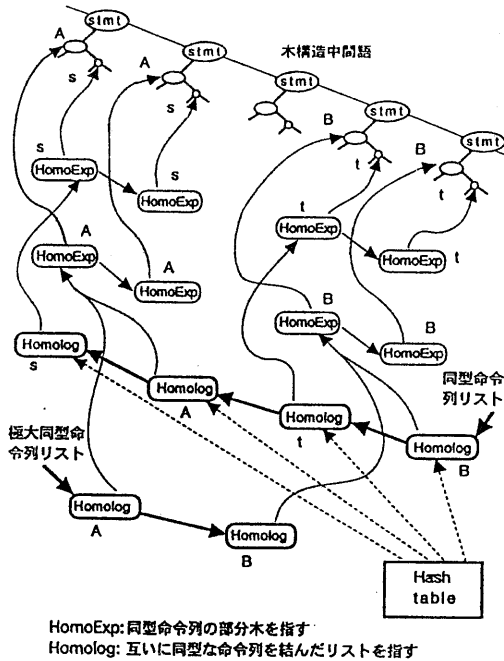

図5　同型命令列認識のためのデータ構造

親ポインタを持つものは、より大きい同型命令列の一部をなすものであり、親ポインタがNULLのものは、一番外側にある極大な同型命令列である。上記のデータ構造ができているとき、極大な同型命令列を求めるには、親ポインタを上方向にたどればよい。この極大な同型命令列に対する同型リスト記述子は、極大同型命令列リストとして、相互につないである。SIMD命令適用可否を判定するには、極大同型命令列リストを調べればよい。

3．3　認識アルゴリズム

同型命令列認識のアルゴリズムは図6の通りである。ここで、木構造同型探査ルーチンは、与えられた部分木と同型な部分木があるか否か調べる。同型性判定ルーチンは、与えられた部分木と同型なものがあれば、それに対する同型記述子や同型リスト記述子を作る。

```
同型命令列認識　{
  初期設定；
  木構造同型探査（木構造中間語のルート）；
  同型リスト記述子の親子関係ポインタを設定；
}
木構造同型探査（部分木のルート）　{
  if（部分木のルートがNULL）
    return;
  木構造同型探査（第1子のルート）；
  木構造同型探査（第2子のルート）；
  同型性判定（部分木のルート）；
}
同型式判定（部分木のルート）　{
  本ノードのハッシュ値を算出；
  if（部分木がリーフ）
    return;
  if（ハッシュ値の同じノードがある）　{
    if（両者が同型部分木）　{
      if（新しい同型部分木）　{
        前の部分木に対するHomoExpを作る；
        Homologを作る；
        ハッシュ表に連結する；
      }
      現部分木に対するHomoExpを作る；
      Homologに接続する；
    }
  }
}
```

図6　同型命令列認識アルゴリズム

4．　メモリ割り付け

並列実行される同型命令列のオペランドとなる変数は、メモリ上で一まとまりになっていることが望ましい。一まとまりになっていれば、それらを一括してロード、ストアしたり、連続してロード、ストアしたりできる。マルチメディア処理では、色彩や濃淡に関する処理など、オペランドが8ビットや16ビットのように、短く表現できるものが多い。それらのデータを一つあるいは連続する2つのレジスタに4個か8個ロードして、同一処理をその4個や8個のデータに並列に実行する命令を備えたプロセッサもある。このような場合、対象データを一括ロード、一括ストアできることが実行効率の点で重要である。

メモリ割り付けは次のようにして行う。

（1）メモリ変数に
　同型命令列のオペランド　>
　その他のオペランド
という関係を持つ同型優先順位と、推定実行頻度に基づく実行頻度順位の２つの順位を付与する。同型優先順位は、同型な部分木の枝の高さに応じて高くなるようにし、複数の同型命令列に現れる変数には、各々の同型な部分木に対して計算した優先順位の最高値を与える。
（2）大域的変数、副プログラムごとの局所的変数というように、メモリ領域ごとにわけて、同型優先順位を第１キィ、実行頻度順位を第２キィとして、メモリ変数をソートする。
（3）ソートされた順位の高い方からメモリ割り付けを行う。

これによって、大きい同型命令列に現れる変数から順番に割り付けが行われ、同型命令列に現れる変数のメモリ領域が隣接するようになる。構造体要素の配置も、同型命令列に現れる時は、ソースプログラムでの宣言順序通りでなく変えた方が良い場合もあるが、それはプログラマへの勧告にとどめざるを得ない。

5. 結果

本方式による同型命令認識アルゴリズムを実験用コンパイラに組み込んで、方式確認を行った。これでは、一部の構造体要素や配列要素が変化する命令列が同型と認識されるばかりでなく、例えば

```
  ...
┌ if (a & 32768)
│   if (a & 16386)
│     v = 0;
│   else
│     v = 256;
│ else
└   v = a * 128;
  x = v;
  ...
┌ if (b & 32768)
│   if (b & 16386)
│     v = 0;
│   else
│     v = 256;
│ else
└   v = b * 128;
  y = v;
  ...
```

に対しては、入れ子になった２つの大きいif文自体が同型命令列と認識される。

隣接する同型命令列は、メモリ配置や使用資源のSIMD命令への適合性を調べ、できる限り大きい単位で並列化する。

6. おわりに

SIMD方式による並列実行は、いわゆる並列プロセッサばかりでなく、マルチメディア命令のような形で、汎用のプロセッサでも行われるようになってきた。そこでは、ループ部分の並列実行だけでなく、オペランドの一部が異なる同型命令列の並列実行が重要になる。

本稿では並列実行の候補となる同型命令列を認識するアルゴリズムを示し、実験による方式確認を行った。それは、条件判定を含んで複数の基本ブロックにわたる同型命令列も容易に認識できる特徴を持つ。コンパイラでの自動並列化が難しいような、対象マシンの特定機能を利用する並列処理もあるが、それらを中間語ライブラリにイディオムとして登録しておくと、それと同型な命令列は、本アルゴリズムにより、一般の場合と同じ処理で、並列化対象として認識できる。

最後に、マルチメディア等に使われるプログラムの特性分析にご協力頂いた、電気通信大学情報工学科 鈴木貢助手に謝意を表する。

7. 参考文献

(1) 富田: 並列計算機構成論, 昭晃堂, 1986.
(2) Tremblay, M., O'Connor, J. M.: UltraSparc I: A four-issue processor supporting multimedia, IEEE Micro, Apr. 1996, pp. 42-50.
(3) Lee, R. B.: Accelerating multimedia with enhanced microprocessors, IEEE Micro, Apr. 1995, pp. 22-32.
(4) 浅見: 米Intel, 86系MPUに57個のマルチメディア命令を追加, 日経エレクトロニクス, Mar. 25, 1996, pp.13-14.
(5) 長島, 田中: スーパコンピュータ, オーム社, 1992.
(6) Zima, H., Chapman, B.（村岡訳）：スーパーコンパイラ, オーム社, 1995.
(7) Rogers, D. F.（山口監訳）：実践コンピュータグラフィックス, 日刊工業新聞社, 1987.