計算機アーキテクチャ 89−12
（1991. 7. 19）

VLIWプロセッサの機能拡張による並列性の向上

森本 真一
日本電気技術情報システム開発(株)

本稿ではVLIWプロセッサにレジスタの直前の値を復元する機能を追加し、それを利用したプログラムの並列性を向上させる変換方法を述べる。さらに、この方法と従来の変換方法（パーコレーション法、ループ展開法、ソフトウェアパイプライニング法）との比較を行なう

IMPROVEMENT OF PERFORMANCE BY ADDING NEW INSTRUCTIONS TO A VLIW PROCESSOR

Shin-ichi Morimoto
NEC Scientific Information Systems development Co.
IGARASHI Bldg, 2-11-5, Shibaura, Minato-ku, Tokyo, 108 Japan

A new program transformation method for VLIW processors is discussed. For this method, new instruction of VLIW processors that restores values of registers is introduced. The transformation method is compared with existent transformation methods such as percolation method, loop expansion and software piplining.

1. はじめに

本稿では、VLIWプロセッサに新しい機能を追加することにより、従来の変換よりも広範囲のプログラムに適用可能なプログラム変換ができる事を述べる。

VLIWプロセッサに対しては、従来からプログラムの並列性を向上させるプログラム変換方法が考案されてきた[1]。しかし、それらの変換方法では適用できる範囲に大きな制限があった。例えば、ループ展開という変換方法は、ループ本体の実行回数がコンパイル時に決定できる場合にしか適用できなかった。このため図1-1のプログラムではループ本体の実行回数がコンパイル時に決定できないので、ループ展開という変換は適用できない。

しかしプロセッサに

- 各レジスタr_iに対する記憶領域($\overline{r_i}$とする)ががあり、r_iに値が代入されると、それまでのr_iの値は$\overline{r_i}$にコピーされる。
- $\overline{r_i}$の値をr_iにコピーする演算(pop(r_i)とする)が存在する。

という機能を追加すると、従来よりも変換できるプログラムの範囲を拡大できる。例えば、図1-1のプログラムを図1-2のように変換できる。

```
loop
  r2:=r1*r1
  r3:=r2+1
  r1:=r1-1
  X(r1):=r3
until X(r1)=0
```

図1-1

```
                  r2:=r1*r1
                  r3:=r2+1
loop              loop
  r2:=r1*r1         r1:=r1-1
  r3:=r2+1          X(r1):=r3   r2:=r1*r1
  r1:=r1-1                      r3:=r2+1
  X(r1):=r3       until X(r1)=0
until X(r1)=0     Pop(r2)  Pop(r3)
```

図1-2

本稿では、2章で追加機能の説明を行ない、3章でこの機能を利用した変換法と従来のプログラム変換方法(パーコレーション法、ループ展開法)を比較する。4章ではこの機能を利用した変換とソフトウェアパイプライニング法の関連を述べる。

2. 追加機能

通常のプロセッサでは、load命令などによってレジスタの値が変更されると変更される前の値を復元する事はできない。本稿では各レジスタr_iは変更される前の値を保持する記憶領域($\overline{r_i}$とする)を持ち、r_iの値を変更する演算は、変更する前の値を$\overline{r_i}$にコピーする事とする。例えば、r_1=1、$\overline{r_1}$=0の場合に、r_1:=r_1+1を実行した後のr_1と$\overline{r_1}$の値はr_1=2、$\overline{r_1}$=1となり、その後でr_1:=r_1*3を実行した後のr_1と$\overline{r_1}$の値はr_1=6、$\overline{r_1}$=2となる。また、$\overline{r_i}$の値をr_iにコピーする演算(Pop(r_1)とする)を持つとする。例えば、r_1=6、$\overline{r_1}$=2の場合に、Pop(r_1)を実行するとr_1:=2、$\overline{r_1}$=2となる。

3. 従来の変換方法との比較

ここでは、2.で述べた追加機能を利用したプログラム変換法を説明し、既存の変換法と比較する。

3.1 条件分岐

図3.1のプログラムを考える。

定義

Sを式または文とするとき、Sの実行時に値が参照されるレジスタの集合をref(S)とする

定義

Sを式または文とするとき、Sの実行時に値が設定されるレジスタの集合をset(S)とする

図3.1のプログラムで

$$\mathrm{set}(S_1)\cap\mathrm{ref}(C)=\phi \quad -(1)$$

が成り立つ場合は、S_1をCの評価以前に実行してもCの評価結果は変わらない。

$$\mathrm{set}(C)\cap\mathrm{ref}(S_1)=\phi \quad -(2)$$

が成り立てばS_1をCの評価以前に実行してもS_1の実行結果は変わらない。このため(1),(2)が成り立

つ場合はS_1をCの評価以前に実行してもCが成立する場合のプログラムの実行結果は変わらない。しかしCが成立しない場合は、S_1をCの評価以前に実行してしまうと本来実行されないはずのS_1が実行されてしまい、set(S_1)の値が壊されてしまう可能性がある。そこで、Cが成立しない場合はPop(set(S_1))を実行すれば、set(S_1)の値はS_1の実行以前の値に戻る。つまり(1)と(2)が成り立つ場合は、図3.1のプログラムを図3.2のプログラムに変換できる。

```
if C
  then
    S1;
    S2;
  else

    S3;
end if;
```

図3-1

```
S1;
if C
  then

    S2
  else
    Pop(set(S1))
    S3;
end if;
```

図3-2

既存のプログラム変換方法で、これによく似たものとして (1)と(2)が成り立つ場合に図3.3のプログラムを図3.4のプログラムに変換するものがある。最初の変換はS_1を上に移動させるのに対して、既存の変換はS_1を下に移動させるものである。

```
S1;
if C
  then

    S2;
  else

    S3;
end if;
```

図3-3

```
if C
  then
    S1;
    S2;
  else
    S1;
    S3;
end if;
```

図3-4

最初の変換では、Cが成り立たない場合はS_1とPop(set(S_1))を余分に実行する必要があるが、既存の変換では実行時間が変わらない。このため最初の変換は、S_1を移動させる事により、S_1とPop(set(S_1))の実行時間を吸収できる場合にかぎり実施すべきである。ただし、ref(Pop(set(S_1)))$=\phi$なので、Pop(set(S_1))はS_3と並列に実行できる可能性が高い。

また最初の変換によるコードサイズの増加はPop(set(S_1))のコードサイズであり、既存の変換によるコードサイズの増加はS_1のコードサイズである。一般に、

S_1のサイズ > Pop(set(S_1))のサイズ

だから、最初の変換のほうがコードサイズの増加が少ない。

3.2 ループ展開

図3-5のプログラムを考える。ここで、S_1,S_2は

- S_1とS_2は並列に実行できない。
- S_2と次のループのS_1とは並列に実行できる。

を満たす文の列とする。このとき従来の変換法(ループ展開法)ではこのループの本体を図3-6のように展開し、それをさらに図3-7のように変換する。

```
for i in 1..N
  S1;
  S2;


end loop;
```

図3-5

```
for i in 1..N/2
  S1;
  S2;
  S1;
  S2;
end loop;
```

図3-6

S_1の部分はN+1回実行してもよい(core dumpしない)とすれば、本稿の機能を用いる事により、図3-5のプログラムは図3-8のように変換できる。

```
for i in 1..N/2
  S1;
  S2; S1;
      S2;
end loop;
```

図3-7

```
  S1;
for i in 1..N
  S2; S1;
end loop
Pop(S1);
```

図3-8

図3-7のプログラムと図3-8のプログラムの実行時間を比較すると次のようになる。

$S_1, S_2, \overline{S_1}$の実行時間をそれぞれ$t_1, t_2, \overline{t_1}$とする。またループ終了の条件判断の時間をt_cとする。ここでは、$t_1 > t_2$ とする。図3-7のプログラムと図3-8のプログラムの実行時間をそれぞれT_7, T_8とすると、

$$T_7 = (t_1+t_1+t_2+t_c)*N/2$$
$$= t_1*N + t_2*N/2 + t_c*N/2$$
$$T_8 = t_1+(t_1+t_c)*N+\overline{t_1}$$
$$= t_1*N + t_1 + t_c*N + \overline{t_1}$$

だから、

$$T_7 - T_8 = (t_2-t_c)*(N/2) - \overline{t_1}$$

となる。一般に

- $t_2 > t_c$
- $\overline{t_1}$ は $t_2*(N/2)$ より十分小さい

が成り立つから、

$$T_7 > T_8$$

となり、本機能による変換のほうがループ展開法よりも効率がよい。また、本機能による変換はループ展開法では扱えなかった実行回数が不定のループ(while文など)にも適用できる。

4. ソフトパイプライニング法との関連

ここでは、本稿の機能とソフトウェアパイプライニング法との関連を述べる。図4-1のプログラムを考える。ここで、k $(0 \le k \le N)$回目のループでの$S_j (1 \le j \le 3)$の実行をS_j^kで表わすときS_j^kは次の条件を満たすとする。

- S_j^kと$S_j^{k'}$は並列に実行できない
- S_j^kと$S_{j'}^{k'}$は並列に実行できない
- S_j^kと$S_{j'}^{k'}$は並列に実行できる

```
for i in 1..N
  S1;
  S2;
  S3;
end loop;
```

図4-1

ソフトウェアパイプライニング法は図4-2のように実行する。

ここでは、ループ終了後の文をループ本体にくりこむ事を考える。つまり図4-3のような変換を考える。ただし3.2と同様にS_1、S_2はそれぞれ$N+2$回、$N+1$回実行してもかまわないとする。

```
S1'
S2'  S1^2
for i in 1..N-2
   S3^i  S2^(i+1)  S1^(i+2)
end loop;
      S3^(N-1)  S2^N
           S3^N
```

図4-2

```
S1'
S2'  S1^2
for i in 1..N
   S3^i  S2^(i+1)  S1^(i+2)
end loop;
Pop(S1)  Pop(S2)
Pop(S1)
```

図4-3

図4-3では、S_2は$N+1$回実行されるので、ループ終了後に　Pop(set(S_2))を実行する事により、set(S_2)の値をもとに戻す事ができる。しかし、S_1は$N+2$回実行されるので、ループ終了後にset(S_1)の値をもとに戻すためにはset(S_1)の現在の値よりも2回前の値に戻す必要がある。しかし、これまで述べてきた機能では各レジスタの直前の値しか復元できないためこれは不可能である。この変換を行なうためには各レジスタの2回前までの値を保持する記憶場所が必要である。逆に各レジスタのN回前までの値を復元できる機能があれば、$N+1$個の文からなるループに対してループの実行回数がコンパイル時に決定できない場合でもソフトウェアパイプライニング法と同様の変換を行なう事ができる。

5. 参考文献

[1]中谷："VLIW計算機のためのコンパイラ技術"
情報処理, Vol.31, No.6, pp.763-772 (1990)