ソフトウェア・パイプライニングの一実現法

中 田 育 男†

ソフトウェア・パイプライニングは，ループの命令レベル並列性を高める 1 つの方法であるが，従来の方法では，カウンタを使ってループの終了判定をするループにしか使えない，ループ終了判定の命令を変更する必要がある，繰返しの回数が少ないループには使えない，といった問題がある．それらの問題を解決する 1 つの方法として，ループの終了判定命令とその命令が依存するすべての命令を最初にスケジュールし，その後でパイプラインのスケジュールをする方法を考案した．プロローグ部でももとの終了判定命令に相当する命令を実行し，カーネルループではもとの終了判定命令をそのまま実行するようにすることで，上記の問題を解決することができる．この方法の欠点はループの終了判定命令が依存する命令に対してはパイプラインのスケジュールをすることができない点である．提案する方法を COINS コンパイラ・インフラストラクチャを使って実装し，いくつかの例題で効果が得られることを確認した．

An Implementation Method of Software Pipelining

Ikuo Nakata†

Software pipelining is an optimization that can improve the loop-executing performance by taking advantage of the instruction-level parallelism. However, the conventional software pipelining method has several problems. It can only be applied to counter controled loops. In some cases the loop control instruction should be modified. It can not be applied if the number of iterations is smaller than the number of stages of pipelining. In this paper we propose a method that solves these problems. Our method first schedules the loop control instruction and its dependent instructions, and then schedules the remaining instructions for pipelining. We implemented this method by using the COINS compiler infrastructure and showed the improvement of the loop-executing performance of several programs.

1. はじめに

ループの命令レベル並列性を高める方法としてソフトウェア・パイプライニングがある．ソフトウェア・パイプライニングに関して，従来の論文[1)~5)] やコンパイラの最適化に関する著書[6)~9)] では，ループ内の命令をできるだけ並列に実行されるように命令スケジューリングをする方法については論じられているが，ループの終了判定の命令についてはほとんど触れられていない．しかし，実際にソフトウェア・パイプライニングを実装する場合は，終了判定命令の作り方も問題になる．本論文では，その 1 つの実現法を提案する．

ソフトウェア・パイプライニングにおけるループの終了判定命令の問題を簡単な例で説明する．たとえば，ソースプログラム

```
float prod(float a[], int n){
    int i; float s = 0;
    for (i = 0; i < n; i++)
        s += a[i]*a[i];
    return s;
}
```

に対する SPARC マシン用の目的コードの中で，ループに対応する部分は，ソフトウェア・パイプライニングをしなければ，次のようなものが得られるであろう．

```
.L1:
    ld    [%i0+%i2],%f1  ;(1) %f1 = a[i]
    fmuls %f1,%f1,%f2    ;(2) %f2 = a[i]*a[i]
    fadds %f0,%f2,%f0    ;(3) %f0 = s += %f2
    add   %i2,4,%i2      ;(4) %i2 = (i++)*4
    cmp   %i2,%i1        ;(5) if (%i2 < n*4)
    bl    .L1            ;(6)  goto .L1
    nop
```

このコードでは，ld 命令の直後にロードされた値を使う fmuls 命令は，ロードが完了するまで実行を待たされてしまう．また，fmuls 命令の実行が完了する

† 法政大学情報科学研究科
Graduate School of Computer and Information Sciences, Hosei University

まで次の fadds 命令の実行が待たされてしまう．命令を並べ替えてそのような待ちによる性能低下をなくすようにするのが命令スケジューリングであり，それをループの繰返しにまたがって行うのがソフトウェア・パイプライニングである．上記のコードに対して，ソフトウェア・パイプライニングのスケジュールをして，最初に (1) を実行し，ループの次の繰返しで (2) を実行し，そのまた次の繰返しで (3) を実行するようにしたとすると，次のようなコードが得られる．このようなスケジューリングは (1)，(2)，(3) をそれぞれ stage 0，1，2 にスケジュールするといわれる．

```
 1 .L1: sub   %i1,4,%i1   ; %i1 = (n-1)*4
 2      ld    [%i0+%i2],%f1  ;(1) stage 0
 3      fmuls %f1,%f1,%f2    ;(2) stage 1
 4      add   %i2,4,%i2      ;(4) stage 1
 5      ld    [%i0+%i2],%f1  ;(1) stage 0
 6 .L2: fadds %f0,%f2,%f0    ;(3) stage 2
 7      fmuls %f1,%f1,%f2    ;(2) stage 1
 8      add   %i2,4,%i2      ;(4) stage 1
 9      ld    [%i0+%i2],%f1  ;(1) stage 0
10      cmp   %i2,%i1        ;(5)
11      bl    .L2            ;(6)
12      nop
13      fadds %f0,%f2,%f0    ;(3) stage 2
14      fmuls %f1,%f1,%f2    ;(2) stage 1
15      add   %i2,4,%i2      ;(4) stage 1
16      fadds %f0,%f2,%f0    ;(3) stage 2
```

ここで，1〜5 行がプロローグ，6〜12 行がカーネル（またはカーネルループ），13〜16 行がエピローグと呼ばれる部分である．プロローグの最初の sub 命令はループ終了判定の条件を変更する命令である．ループを抜けた後に (4) の命令が実行されているから，その分の補正をする命令である．このような補正が可能になるのは，ループの終了判定命令にカウンタを使っているからである．カウンタによる判定命令であれば，終了時の 1 回前のカウンタの値や 2 回前の値を設定しておくことができるからである．カウンタによらないループであれば，このようなことは一般にはできない．たとえば，

```
for (i = 0; c[i] > 0; i++)
    s += a[i]*a[i];
```

というループには上記の方法は使えない．この場合，c[2] < 0 であれば正しく実行できないし，もし a[i] の演算が stage 0 にスケジュールされ，c[i] の演算が stage 1 にスケジュールされたとしたら，c[i] の判定で終了するときには a[i] の演算が始まってしまっているからである．

上記のコードにはもう 1 つ問題がある．それはループの終了判定の命令が最初に実行される前に ld 命令が 3 回実行されるようになっていることである．したがってこのコードは n < 3 の場合には使えない．このような問題を解決するために，通常は，コンパイル時に n の値が分からないときは，n < 3 の場合に実行するコードも別に作っておいて，実行時にどちらかを選んで実行する方法がとられる．

以上のように，従来の方法はカウンタを使ってループの終了判定をするループにしか使えない，ループ終了判定の条件を変更する必要のある場合がある，繰返しの回数が少ないループには使えない，といった問題がある．それらの問題を解決する方法として，ループの終了判定命令とその命令が依存するすべての命令を最初にスケジュールし，その後でパイプラインのスケジュールをする方法を提案する．それを COINS コンパイラ・インフラストラクチャ[16] を使って実装した結果についても報告する．

2. 終了判定命令を考慮したソフトウェア・パイプライニング

本論文で対象とするループは次の 2 つの条件を満たすものである．

- do-while 型ループの形の機械語プログラム
- ループの本体は 1 つの基本ブロックだけからなる

通常の for ループや while-do 型のループは，ループをさらに繰り返すかどうかを判定してからループの本体の実行に入り，実行したら繰返しの判定のところに戻る形をしているが，それをそのまま機械語のコードにすると，ループを 1 回回るごとに 2 回のブランチ命令を実行することになる．もとのループを do-while 型の形に変換すれば，ブランチ命令が 1 つですむからより効率の良いループとなる．そのためには，必要ならば，ループを 1 回も実行しない場合にそのループをバイパスするような命令が作られる．ここではそのように変換された do-while 型ループの形の機械語プログラムを対象とする．1 章で最初に示した機械語のプログラムもこの形にしてある．

ループの本体に条件文を含む（複数の基本ブロックを含む）場合のソフトウェア・パイプライニングは簡単ではない[10]．各命令の条件実行の機能と rotating predicate file を持ったマシンでは，if 変換をすることによって，条件文があっても 1 つの基本ブロックに変換することができる[11)〜13]．本論文の方法をこれらに適用することも可能であるが，ここではそれには触れ

ないことにする．

ソフトウェア・パイプライニングの実現法としては，以下の 2 段階で行われるのが通常の方法である（本提案でも，この 2 段階で行う）．

(1) カーネルループをスケジュールする．
(2) カーネルループからプロローグとエピローグを作る．

ループの終了判定命令はそのスケジュールの際には考えずに，カーネルループができてから適当に考えて，カーネルループの最後に置けばよいとされている．しかし，それでは前章に述べたような問題が起きる．

そのような問題点を解決する方法として，ループの終了判定命令とその命令が依存するすべての命令を最初にスケジュールし，その後でパイプラインのスケジュールをし，プロローグ部にも終了判定命令を入れる方法を提案する．その方法を前章の例を使って説明する．

カーネルループをスケジュールする方法としては，通常，モジュール資源予約表を用意して，それにスケジュールした命令を埋めていく方法がとられる[3),7)]．モジュール資源予約表としては，ここでは簡単のために，ループ内の命令数だけの長さを持つ表を利用し，表の各スロットにスケジュールした命令とパイプラインの stage の値を埋めていくことにする．今の例では配列の長さが 7 の 1 次元配列を用いている．

各命令のスケジューリングは，命令の依存関係とレイテンシ（実行時間）の情報に基づいて行われる．対象とする機械語のプログラムの中で，命令 A が命令 B に先行しており，A で値が定義されるレジスタやメモリが B で使われていたら，B は A にフロー依存または真に依存するといわれ，A で値が定義されるレジスタやメモリが B でも定義されていたら，B は A に出力依存するといわれ，A で使われるレジスタやメモリが B で定義されていたら，B は A に逆依存するといわれる．「命令が依存する」とは，これらの 3 つの関係のうち少なくとも 1 つが成り立つことをいう．

メモリに関しては，正確には同一ロケーションに対して定義したり使ったりした場合に依存関係があるのであるが，機械語のレベルでそれを判定するのは簡単ではないので，後の章で述べる実装では，すべてのメモリアクセスは同一ロケーションに対するアクセスと見なして，依存関係を求めている．

以下の例では命令のレイテンシは**表 1** に与えられているものとしている．

表 1 各命令のレイテンシ（サイクル）
Table 1 Latency of instructions.

	レイテンシ
ld, fmuls, fadds	4
add, cmp	1

2.1 カーネルループをスケジュールする

本提案では，これをさらに以下の 2 段階に分けて行う．

2.1.1 ループの終了判定に関する命令をスケジュールする

ループの終了判定命令とその命令が依存するすべての命令を最初にスケジュールするのが本提案の特徴である．これらの命令を stage 0 で実行するようにスケジュールすることによって得られる効果は，ループ終了判定命令に関わる値を変更する必要がなくなることと，ループの実行回数が不定回のものにも使えるようになることである．

今の例では，まず終了判定の分岐命令とその遅延スロットに入る nop 命令をモジュール資源予約表の最後部にスケジュールする．次に，その分岐命令が真に依存する cmp 命令を，命令のレイテンシ分だけ前に（すなわち分岐命令の直前に）スケジュールする．次に，その cmp 命令 が真に依存する add 命令を同様にレイテンシに従ってスケジュールする．最後にその add 命令が逆依存する ld 命令をその直前にスケジュールする．その結果，モジュール資源予約表は次のようになる．

```
0
1
2   ld    [%i0+%i2],%f1     ; stage 0
3   add   %i2,4,%i2         ; stage 0
4   cmp   %i2,%i1           ; stage 0
5   bl    .L?               ; stage 0
6   nop                     ; stage 0
```

ここで，ld 命令以外はレイテンシが 1 の命令であるから詰めてスケジュールしてある．ld 命令のレイテンシは 1 ではないが，次の add 命令との依存関係は真の依存関係ではない（レジスタ %i2 が ld 命令で使われてから add 命令で定義されているという逆依存の関係）から，ld 命令も次の add 命令の直前においてある．分岐命令の飛び先はまだ決まらないので，「.L?」としてある．

2.1.2 残りの命令に対してパイプライニングのスケジュールをする

残りの命令は fmuls 命令と fadds 命令だけである．fmuls 命令が依存している命令はモジュール資源予約表のインデックス 2 にスケジュールされている ld 命令だけであり，その依存関係は真の依存であるから，

それから ld 命令のレイテンシ以上離れたところにスケジュールすればよい．そのレイテンシは 4 であるから，スケジュールする候補のインデックスは 6 である．そこにはすでに nop 命令が入っており，それがモジュール資源予約表の最後であるから，次の候補はループの次の繰返しの先頭のインデックス 0 になる（実は，nop 命令は遅延分岐命令の遅延スロットに置かれている命令であり，今の場合はそれを fmuls 命令で置き換えてしまうことも可能であるが，ここではその問題には触れないことにする）．インデックス 0 の場所が空いているので，そこにスケジュールする．この fmuls 命令はループの次の繰返しで実行されることになる．すなわち次の stage で実行されるので stage の値を 1 増やしてやる必要がある．その結果，モジュール資源予約表は次のようになる．

```
0   fmuls %f1,%f1,%f2       ; stage 1
1
2   ld    [%i0+%i2],%f1     ; stage 0
3   add   %i2,4,%i2         ; stage 0
4   cmp   %i2,%i1           ; stage 0
5   bl    .L?               ; stage 0
6   nop                     ; stage 0
```

最後に fadds 命令をスケジュールするが，それが真に依存する fmuls 命令からそのレイテンシ分の 4 だけ先はすでに埋まっている．モジュール資源予約表の先頭に戻ってみると，先頭には fmuls 命令が入っている．fmuls 命令を越えて fadds 命令をスケジュールすれば，fmuls 命令の結果のレジスタの生存区間がループの 1 回り分以上必要になってしまい，rotating register file のようなハードウェア機構[11),12),14)] を持たないマシンの場合は，その問題を解決するためにループ展開が必要になってしまう．そこで，ここではモジュール資源予約表を広げて fmuls 命令の直前にスケジュールすることにする．この fadds 命令は対応する fmuls 命令を実行した次の繰返しで実行されることになるので stage の値を fmuls 命令のそれより 1 増やして 2 とする必要がある．

以上の結果，以下のようにカーネルが完成する．カーネルの中に空の部分が存在するが，以下ではそれを無視すればよい．

```
0   fadds %f0,%f2,%f0       ; stage 2
1   fmuls %f1,%f1,%f2       ; stage 1
2
3   ld    [%i0+%i2],%f1     ; stage 0
4   add   %i2,4,%i2         ; stage 0
5   cmp   %i2,%i1           ; stage 0
6   bl    .L?               ; stage 0
7   nop                     ; stage 0
```

2.2 カーネルループからプロローグとエピローグを作る

ここでプロローグにもループ終了判定命令を入れるのが，本提案の特徴である．それによって，繰返し回数が少ないループにも使えるようになる．

今の例では，プロローグとエピローグを作った結果は以下のようになる．

```
 0 .L1: ld    [%i0+%i2],%f1     ; stage 0
 1      add   %i2,4,%i2         ; stage 0
 2      cmp   %i2,%i1           ; stage 0
 3      bge   .L5               ; stage 0
 4      nop                     ; stage 0
 5 .L2: fmuls %f1,%f1,%f2       ; stage 1
 6      ld    [%i0+%i2],%f1     ; stage 0
 7      add   %i2,4,%i2         ; stage 0
 8      cmp   %i2,%i1           ; stage 0
 9      bge   .L4               ; stage 0
10      nop                     ; stage 0
11 .L3: fadds %f0,%f2,%f0       ; stage 2
12      fmuls %f1,%f1,%f2       ; stage 1
13      ld    [%i0+%i2],%f1     ; stage 0
14      add   %i2,4,%i2         ; stage 0
15      cmp   %i2,%i1           ; stage 0
16      bl    .L3               ; stage 0
17      nop                     ; stage 0
18 .L4: fadds %f0,%f2,%f0       ; stage 2
19 .L5: fmuls %f1,%f1,%f2       ; stage 1
20      fadds %f0,%f2,%f0       ; stage 2
```

ここで，プロローグは 2 つのブロックからなる（.L1:，.L2: がそれぞれの先頭）．最初のブロックには stage 0 の命令だけが入り，分岐命令は，もとの分岐命令の条件を反転（bl を bge に）したものであり，飛び先はエピローグの 2 番目のブロックの先頭（.L5:）である．プロローグの 2 番目のブロックは stage 0 と stage 1 の命令だけからなり，飛び先はエピローグの最初のブロックの先頭（.L4:）である．

できあがったコードは，先に述べた問題点をすべて解決している．すなわち，これは繰返しの数が 1 でも 2 でも使えるコードであり，ループ終了判定の条件（最終値）の補正はしていない（条件の反転は通常いつでも可能であり，これは補正とは違う）．ループ終了判定の条件の補正が必要でないから，カウンタを使わないループにも適用可能である．

今の例では，stage の最大値は 2 であった．一般に

は，カーネルの中の stage の最大値を n としたときに，エピローグとプロローグのブロックの数はともに n であり，以下のように作ればよい．

プロローグ 0 カーネルの stage 0 だけからなる命令の列．ただし，最後の分岐命令はもとの条件を反転したもので，飛び先はエピローグ $n-1$．

プロローグ 1 カーネルの stage 0 と 1 だけからなる命令の列．それらの命令の順はカーネルの中と同じ．ただし，最後の分岐命令はもとの条件を反転したもので，飛び先はエピローグ $n-2$．

. . .

プロローグ $n-1$ カーネルの stage 0 から $n-1$ までからなる命令の列．それらの命令の順はカーネルの中と同じ．ただし，最後の分岐命令はもとの条件を反転したもので，飛び先はエピローグ 0．

カーネルループ カーネルのすべての命令の列．最後の分岐命令はもとのまま．

エピローグ 0 カーネルの stage n だけからなる命令の列．各 stage の命令順はカーネルの中と同じ（以下のエピローグでも同様）．これを実行したら，次のエピローグ 1 に進む．

エピローグ 1 カーネルの stage $n-1$ だけからなる命令の列の後に stage n だけからなる命令の列．これを実行したら，次のエピローグ 2 に進む．

. . .

エピローグ $n-1$ カーネルの stage 1 だけからなる命令の列，stage 2 だけからなる命令の列，. . .，stage n だけからなる命令の列，がこの順に並んだもの．

以上の作り方で正しいことは，次のように考えれば分かる．ループの繰返し数が 1 のときは，プロローグ 0 で stage 0，エピローグ $n-1$ で stage 1 以降をすべて実行するから正しい．ループの繰返し数が 2 のときは，プロローグ 1 で stage 0 と stage 1 を実行し，そのうちの stage 1 を実行した分については残りをエピローグ $n-2$ で実行し，stage 0 を実行した分については残りをエピローグ $n-1$ で実行するから正しい．以下同様である．

以上のように，最初に分岐命令とそれが依存する命令をスケジュールすることにより，従来の問題点を解決することができたが，それにより，別の問題点も生ずる．それは，分岐命令が依存する命令の間ではパイプラインスケジュールができないことである．したがって，そのような命令がループの大部分を占める場合は，パイプライニングの効果があまり得られなくなる可能性がある．

3. COINS での実装とその効果

COINS [16] は，コンパイラの研究・開発を容易にするための基盤として開発されたコンパイラ・インフラストラクチャであり，コンパイラを構成するために必要な標準的なモジュールを備えており，それを使ってさらに機能を拡張することも比較的容易である．その拡張機能の 1 つとして，前章の方式のソフトウェア・パイプライニングを行うものを実装した．

3.1 COINS での実装

命令スケジューリングやソフトウェア・パイプライニングとレジスタ割付けに関しては，どちらを先にするのがよいかは簡単には決まらない．レジスタ割付けをした後では，レジスタによる依存が生じて並列性が生かされなくなる可能性があるし，レジスタ割付け前では，並列性は生かされるがレジスタを多く使用するようになってレジスタ割付けが難しくなる可能性がある．COINS のレジスタ割付けは，使用するレジスタをできるだけ少なくする方針で割り付けられており，演算の途中結果はほとんど同じレジスタに割り付けられるので，その後でのソフトウェア・パイプライニングはあまり効果が出せない．たとえば，前章の例題では

```
ld     [%i0+%i2],%f1
fmuls  %f1,%f1,%f1
fadds  %f0,%f1,%f0
```

のようにレジスタ割付けが行われ，`%f2` を利用せず，`ld` の結果と `fmuls` の結果のレジスタが同一であるので，`fmuls` 命令と `fadds` 命令の順序を前章のように入れ替えることはできない．そこで，COINS の標準機能でレジスタ割付けが行われる前にソフトウェア・パイプライニングを行うこととした．COINS ではレジスタ割付け前には，演算の途中結果はすべて異なる名前の仮想レジスタに割り付けられているので，上記のようなことは起こらない．よって提案手法の効果が期待できる．

COINS では，コンパイラのバックエンドでの処理の対象となるものは低水準中間表現 LIR で表現されている．それはバックエンドのすべてのフェーズで貫かれている．命令選択によってターゲットマシンの機械語に変換された後でも LIR というマシンに依存しない形で表現されている．したがって，命令選択の後でかつレジスタ割付け前に行うソフトウェア・パイプライニングのモジュールをマシンに依存しない形で書くことができる．そこでは，LIR で表現されている各命令から，その命令が使っているレジスタ名と定義しているレジスタ名を取り出して各命令の依存関係を知

表 2 いくつかの例題の実行時間
Table 2 Execution time of example programs.

	no-opt	ssa	ssa+**pipe**	gcc-O2	gcc-O3
prod	17.83	13.31	**8.74**	13.26	13.25
行列積	4.26	1.95	**1.24**	1.94	1.94
相関係数 1	34.46	27.23	**22.16**	25.25	25.13
相関係数 2	35.21	17.58	**12.98**	15.01	15.01

ることができる．また，COINS では CPU 固有の情報は，レイテンシも含め各 CPU ごとに用意されたマシン記述ファイル（TMD ファイル）に記述されており，それらの情報を取り出すインタフェースが用意されているので，そこから得られる各命令のレイテンシの情報を使ってスケジュールすることができる．

実装にあたっては，最初に，命令の依存関係をグラフに表現してそれを使って基本ブロック内の命令スケジュールを行うものを開発し，その依存グラフを使ってソフトウェア・パイプライニングを行うものを実装した．

3.2 効　　果

表 2 に，いくつかの例題について，COINS でソフトウェア・パイプライニングを行った場合と行わなかった場合の実行時間と，参考のために gcc（バージョン 3.4.2）で -O2 と -O3 のオプションを指定した場合の実行時間を載せる．単位は秒である．

no-opt，ssa，ssa+pipe が COINS での結果である．no-opt は最適化オプションを何も指定しない場合，ssa は ssa 最適化を十分に行った場合，ssa+pipe は ssa 最適化に加えてさらにソフトウェア・パイプライニングを行った場合である．マシンは Sun Blade 1000（デュアル UltraSPARC-III，750 MHz，メモリ 1 GB）という SPARC マシンである．prod は前章の例題で配列の長さを 1000000 としたものを 1,000 回呼び出したもの，行列積は 500×500 の float 型の行列の積，相関係数 1 と 2 は，それぞれ以下のようなループが実行の主たる部分を占めるプログラムである．

相関係数 1：変数の型は double

```
for (i = 0; i < n; i++) {
   sx += x[i];  sy += y[i];
}
sx /= n;  sy /= n;
for (i = 0; i < n; i++) {
   dx = x[i] - sx;  dy = y[i] - sy;
   sxx += dx * dx;  syy += dy * dy;
   sxy += dx * dy;
}
```

相関係数 2：変数の型は double

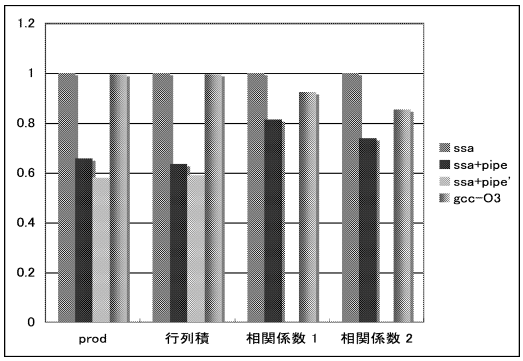

図 1 実行時間比
Fig. 1 Ratio of execution times.

```
for (i = 0; i < n; i++) {
   sx += x[i];  sy += y[i];
   sxx += x[i] * x[i];
   syy += y[i] * y[i];
   sxy += x[i] * y[i];
}
```

いずれも，n の値は 200000 で，このループを含んだ関数を 2,500 回呼び出している．

表 2 の ssa の実行時間を 1 とした場合の ssa+pipe のそれは，**図 1** に示すように，0.66，0.64，0.81，0.74 であり，ソフトウェア・パイプライニングは十分な効果をあげている．

しかし，本手法には，終了判定命令が依存するすべての命令を同じ stage 0 にスケジュールするため，それらの命令に関してはパイプライニングのスケジュールが行えないという問題がある．試みに，表 2 の中で，そのような命令の割合が大きい prod と行列積について，手動で，そのような命令も含めたソフトウェア・パイプライニングをしたところ，それぞれ 7.72 秒と 1.15 秒という結果が得られた（図 1 の ssa+pipe' が ssa との実行時間比を示す）．

prod ではソフトウェア・パイプライニングの対象となるループの命令数が 7 で，そのうちの 5 命令（約 7 割）が終了判定命令とそれが依存する命令であり，パイプライニングとしてのスケジュールをしたのは 2 命令にすぎないが，それでもその 2 命令のレイテンシが比較的大きいので，手動のもの（パイプライニングをしたのは 3 命令）に比べて約 13%の時間増ですんでいる．

行列積では対象ループの命令数が 9 で，そのうちの 5 命令（約 6 割）が終了判定命令とそれが依存する命令であり，パイプライニングをしたのは 4 命令であるが，手動のもの（パイプライニングをしたのは 6 命令）

に比べて約 8%の時間増ですんでいる．実はこの例では遅延スロットを考慮したスケジューリング結果が実行時間の差に影響を与えている．本方法でソフトウェア・パイプライニングをした結果のカーネル・ループは

```
1 L9: fadds %f2,%f3,%f2    ; stage 2
2     fmuls %f1,%f0,%f3    ; stage 1
3     add   %i5,4,%i5      ; stage 1
4     ld    [%i5],%f1      ; stage 0
5     ld    [%i4],%f0      ; stage 0
6     add   %i4,2000,%i4   ; stage 0
7     cmp   %i4,%o2        ; stage 0
8     bl    .L9            ; stage 0
9     nop                  ; stage 0
```

で，この段階では手動のものに負けてはいない．しかし，最後のフェーズで遅延スロット（9 行目の nop 命令のところ）に移す命令を選ぶとき，本方式で最初にスケジュールしたループの終了判定命令が依存する命令（5 行目以降）からは選ぶことができないので，4 行目の `ld` 命令が選ばれることになる．その命令が `fmuls` 命令と近くなってしまって，遅延スロットによる時間のロスは防げても，パイプライニングの効果が減衰している．

それに対して，手動のものでは，上記の 3 行目と 6 行目の `add` 命令が入れ替わっており，`add %i5` の命令が遅延スロットに移されて，パイプライニングの効果も保たれる．

4. 関 連 研 究

ループの終了判定がカウンタによるものでない場合のソフトウェア・パイプライニングに 1 章で述べたような問題があることは文献 15) の 11.3.3 項や文献 9) の 18.2.8 項で指摘されている．しかし，文献 15) では，その場合はパイプライニングが制限されると述べるにとどまり，その解決策は述べていない．文献 9) では投機的実行のハードウェア機構があれば解決できるとしている．

投機的実行のハードウェア機構を持ったマシンによる解決の方法は文献 11) と 12) に述べられている．前者のマシンは Cydra 5 であり，後者のマシンは Itanium である．これらのマシンは，投機的実行の機能を持つだけでなく，rotating register file と rotating predicate file を持ち，条件実行の機能を持っている．さらに，2 種類のループ（カウンタによるループとそれ以外のループ）のそれぞれのソフトウェア・パイプライニングのための特殊な分岐命令も持っている．文献 11) と 12) には，これらの機構を使った 2 種類のループのソフトウェア・パイプライニングの方法が述べられている．しかし，本論文で取りあげたもう 1 つの問題である，繰返し回数の少ないループの問題については触れられていない（実際にはカウンタによるループの場合は kernel only のループ（カーネルループだけでプロローグとエピローグの役割も果たしてしまう）も可能になることが述べられており，それはプロローグにも分岐命令があることに相当するから，繰返し回数の少ないループにも使えると思われる）．

文献 13) には，文献 11) と 12) と同じようなハードウェア機構がある場合のソフトウェア・パイプライニングの方法だけでなく，それがない場合のソフトウェア・パイプライニングの方法も述べられている．ただし，カウンタによらないループの場合は投機的実行の機能が必要であるとしている．また，繰返し回数の少ないループにも使えるようにする方法も述べられているが，それはカウンタによるループにだけ使えるものである．

我々の方法は，特殊なハードウェア機構を必要とせず，1 つの方法で両方の種類のループに使えるものであり，繰返し回数の少ないループにも使えるものである．

5. お わ り に

従来のソフトウェア・パイプライニングの方法では，ループの終了判定命令の問題はほとんど議論されていなかったが，従来の方法ではループの終了判定条件の変更が必要になる場合がある．機械語のレベルのプログラムを解析して，どのような変更をする必要があるかを判断するのは必ずしも簡単ではない．実装は COINS の低水準中間表現 LIR を対象として行ったが，LIR は機械語のレベルである．実は，そのレベルの命令の内容を解析するのが面倒であるという気持ちがあったので，それなしで済ませる方法として今回提案する方法を思いついた．

提案した方法は，ループの終了判定命令とそれが依存する命令を最初にスケジューリングしてしまってから，パイプライニングを行う方法である．それを COINS コンパイラ・インフラストラクチャを使って実装し，その効果があることを確認した．

実装はマシンに依存しない形で書かれているので，いろいろなマシンに適用することが可能である．ただし，最近の EM64T に対応していないインテル x86 型のマシンではレジスタの数が少なく，特に浮動小数点レジスタはスタックとなっていて，基本的にはスタックのトップのレジスタにしかアクセスできないので，ソフトウェア・パイプライニングはほとんど成功しない．

謝辞 COINS コンパイラ・インフラストラクチャは，平成 12 年度から 16 年度までの文部科学省科学技術振興調整費の援助を受けて作成したものである．その開発に多大な努力を傾けられた COINS プロジェクトメンバならびに関係者の方々に感謝する．COINS があったのでソフトウェア・パイプライニングを実装する気にもなったし，提案したアルゴリズムを実装するのも比較的容易であった．また，このアルゴリズムの初期のバージョンを実装した法政大学大学院情報科学研究科（現在は日立製作所）の長瀬卓真君に感謝する．本実装はそれをもとに行ったものである．また，本論文の原稿に対して貴重な改良のご意見をいただいた査読者と開和生氏（国立環境研究所）に感謝する．

参 考 文 献

1) Lam, M.S.: Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *PLDI'88*, pp.318–328 (1988).
2) Rau, B.R., Lee, M., Tirumalai, P.P. and Schlansker, M.S.: Register Allocation for Software Pipelined Loops, *PLDI'92*, pp.283–299 (1992).
3) Allan, V.H., Jones, R.B., Lee, R.M. and Allan, S.J.: Software Pipelining, *ACM Computing Surveys*, Vol.27, No.3, pp.367–432 (1995).
4) Wang, J. and Gao, G.R.: Pipelining-Dovetailing: A Transformation to Enhance Software Pipelining for Nested Loops, *CC'96* (*LNCS 1060*), pp.1–17 (1996).
5) Ruttenberg, J., Gao, G.R., Stoutchinin, A. and Lichtenstein, W.: Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler, *PLDI'96*, pp.1–11 (1996).
6) Muchnick, S.S.: *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers (1997).
7) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
8) Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers (2002).
9) Srikant, Y.N. and Shankar, P. (Eds.): *The Compiler Design Handbook—Optimizations and Machine Code Generation*, CRC Press (2003).
10) 山下義行，中田育男：ループ中に条件分岐を含む場合の最適なソフトウェア・パイプライニング，並列処理シンポジウム JSPP'94, pp.17–24 (1994).
11) Tirumalai, P., Lee, M. and Schalansker, M.: Parallelization Of Loops With Exits On Pipelined Architectures, *Proc. Supercomputing'90*, pp.200–212 (1990).
12) Muthukumar, K., Chen, D.-Y., Wu, Y. and Lavery, D.M.: Software Pipelining of Loops with Early Exits for the Itanium Architecture, *The 1st Workshop on EPIC Architectures and Compiler Technology* (*EPIC-1*) (2001).
13) Rau, B.R., Schlansker, M.S. and Tirumalai, P.P.: Code Generation Schemas for Modulo Scheduled Loops, *25th Annual International Symposium on Microarchitecture*, pp.158–169 (1992).
14) 中田育男，山下義行，小柳義夫：計算物理学と超並列計算機—CP-PACS 計画：3. 超並列計算機 CP-PACS のソフトウェア，情報処理，Vol.37, No.1, pp.29–37 (1996).
15) Johnson, M.: *Superscalar Microprocessor Design*, Prentice-Hall (1991).
16) http://www.coins-project.org/

(平成 18 年 5 月 1 日受付)
(平成 18 年 8 月 10 日採録)

中田 育男（正会員）

昭和 10 年生．昭和 35 年東京大学大学院数物系研究科数学専攻修士課程修了．同年（株）日立製作所中央研究所入社．昭和 48 年より同社システム開発研究所勤務．昭和 54 年より筑波大学電子・情報工学系教授．平成 9 年より図書館情報大学教授．平成 12 年より法政大学情報科学部教授．平成 18 年より法政大学大学院情報科学研究科客員教授．プログラミング言語とコンパイラに関する研究に従事．理学博士．本会名誉会員．