計算機アーキテクチャ 149－15
（２００２．８．２２）

動的ソフトウェアパイプライニング技術の提案と性能評価

田上 裕之† 村上 和彰††

†九州大学 大学院システム情報科学府
††九州大学 システム情報科学研究院 情報理学部門
E-mail:†,††arch@c.csce.kyushu-u.ac.jp

あらまし コンピュータのハードウェアやソフトウェアを動的に最適化する「動的最適化技術」が注目されている．筆者らは，動的最適化技術の１つである動的コード変換として，本来はコンパイル時に行うソフトウェアパイプライニングを動的に，つまり対象プログラムの実行中に当該プログラムに対して動的に適用する「動的ソフトウェアパイプライニング」技術を提案する．ソフトウェアパイプライニングは条件分岐を含まないループに対して適用するための技術であり，ループ中に条件分岐を含む場合には単純に適用することができない．従来の方法は，ループ中の全てのパスを最適化の対象としてきたが，本稿で提案する動的ソフトウェアパイプライニングでは，ループ中に複数存在するパスの中から，実行時プロファイリングにより検出されたホットパスに対してソフトウェアパイプライニングを適用する．ホットパス以外のパスが実行されるときは，ソフトウェアパイプライニング化されたループから抜け出し補正コードが実行される．これにより条件分岐を含むループに対してもソフトウェアパイプライニングを適用できる．

Proposal and Evaluation of Dynamic Software Pipelining

Yasuyuki Tanoue† Kazuaki Murakami ††

†Department of Informatics, Kyushu University
††Division of Informatics, Kyushu University
E-mail:†,††arch@c.csce.kyushu-u.ac.jp

Abstract Dynamic optimization for software and hardware is gaining the attention of computer systems researchers as an effective means of boosting performance. We are challenging dynamic binary rewriting, using a software pipelining technique. Software pipelining is one compiler technique, whose principle is to overlap different iterations of a loop body in order to exploit parallelism. But loops with conditional branches are difficult to software pipeline because there are multiple paths of execution to schedule. Traditional techniques to solve this problem apply software pipelining to all paths of loops. In contrast to these techniques, Dynamic software pipelining applies it to one frequent executed path (hot path), which is one of profile information collected by a profiler. To execute the other paths, we prepare compensation code. Therefore we can apply software pipelining to loops with conditional branch.

1. はじめに

近年コンピュータシステムを構成するハードウェアやソフトウェアを動的に（そのシステムの運用時に）最適化する技術が注目されている．動的な最適化によりシステム運用後での規格変更や最適化，システムの実態に応じたシステム特性（性能，消費エネルギー等）の最適化が可能になる．また，システム開発時の最適化作業を簡略に行い，システムを使いながら徐々に最適化していくことで，システムの開発期間やコストの低減も可能になる．この動的最適化を行う際に重要になってくるのが、何をどう最適化するかということである．

我々はこの中で，実行時にオブジェクトコードの書き換えを行う動的コード変換，特にソフトウェアパイプライニング（SoftWarePipelining, 以下 SWP）を用いた動的コード変換技術について述べる．SWP とはループの異なるイタレーションに属する命令の中から命令列を選び出してループを再構成し，命令レベル並列度を上げるコンパイラの技法である．本来，SWP は条件分岐を含まないループに対して適用するための技術であり，ループ中に条件分岐を含む場合には単純に適用することができない．この問題を解決する方法として，ハイアラキカル・リダクション [1]，IF 変換 [2], 逆 IF 変換 [3] などの手法がある．しかし，これらの方法には，条件分岐先の then 部や else 部の中に実行に長い時間かかる命令があるときに効果が小さい，特殊なハードウェア機構が必要，コードサイズの指数関数的な増加などの問題がある．

そこで本稿では，条件分岐を含むようなループに対して，従来の SWP 手法とは違ったアプローチで SWP を施す新しい手法を提案する．従来の手法はループ中のすべてのパスを対象に SWP を適用するものである．一方，本稿で提案する手法では，オンラインプロファイリングの結果を利用してホットパス（頻繁に実行されるパス）にだけ SWP を行う．これにより，ループ中に条件分岐を含んでいても SWP を適用できることになる．

本稿では 2 章で，新しく提案する動的 SWP の手法について，その概念や，コード変換方法について説明する．3 章では，動的 SWP の手法の予備評価，およびその結果について考察を行い，4 章でまとめる．

2. 動的 SWP

2.1 概念

本来，SWP は条件分岐を含まないループに対して適用するための技術であり，ループに条件分岐を含む場合には単純に適用することができない．この問題を解決する方法として [1]，[2]，[3] のような手法が提案されているが，それぞれの手法には問題点がある．

そこで，本章では条件分岐を含むようなループに対して動的に SWP を適用する新しい手法を提案する．図 1に本手法の概念を示す．これまでの手法はループ中のすべてのパスを対象に SWP を適用するものであった．そのためスケジューリングする際に制約を受けたり，コードサイズの増加といった問題点が生じてきた．本章で提案する手法は，実行時のプロファイリングを利用してホットパスを見つけ，実行時にホットパスにだけ SWP を適用する．しかし，ホットパス以外のパスが実行されるときには補正が必要となってくる．この補正のためのコード（補正コード）が実行されるときは，SWP の恩恵は受けない．つまり，本手法の性能向上率は SWP 化した高頻度パスの性能向上率と，当該パスを実行する頻度に依存する．

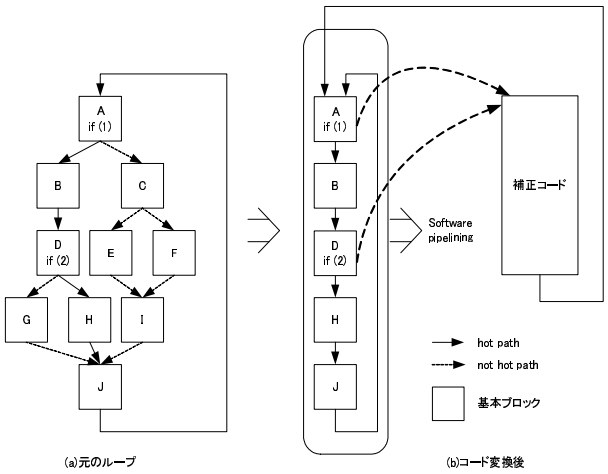

図 1: 動的 SWP の概念図

2.2 コード変換の基本方針

コード変換の基本的な流れとしては，(1) ホットパスの検出，(2) ホットパスに対して SWP を適用，(3) ホットパス以外のパスを実行するための補正コードを生成，のようになる．

(1) ホットパスの検出

プロファイリングによって，条件文を含むループからホットパスを見つける．本稿ではプロファイリングについては考慮せず，ホットパスは見つかったものとして議論する．

(2) ホットパスに対する SWP の適用

ホットパスに対して SWP を適用する．つまりホットパスはここで生成されるコードで実行される．スケジューリングの手法としては，代表的な手法であるモジュロ・スケジューリングを利用する．このとき，ホットパス中に条件分岐の命令を残したままスケジューリングを行い，実行されるパスがホットパスでないときはこの条件分岐からジャンプして (3) で生成されるコードが実行される．このとき注意しなければいけないのは，正しい実行を行うために，条件分岐命令の後の命令を，条件分岐命令の前にスケジューリングすることはできない．

(3) 補正コードの生成

ここでは，ホットパス以外のパスを実行するためのコードを別に生成する．ホットパス以外のコードの実行が終了したら，再び (2) で生成されたコードへジャンプして，ホットパスを実行する．

2.3 補正コード

2.3.1 補正コードの実行

ここでは，ホットパス以外のコードがどのように実行されるかについて詳しく説明する．まず，ループが実行される際，すべてホットパスが実行される場合，初めにプロローグが実行され，その後カーネルループが実行される．カーネルループを抜ける条件を満たしたら，エピローグを実行して実行は完了する．一方，ホットパス以外が実行されるときは，次のようになる．今，図 2のように時刻 t のカーネルループ中の条件分岐でイタレーション $(i-1)$ がホットパスでないことがわかったとする（この例では条件分岐はステージ 2 に存在）．このとき補正コードを実行して，時刻 $(t+4)$ のカーネルループに戻ってくる．補正コードの実行は以下の 3 つのブロックが順に実行される．

(1) 補正カーネル部

このブロックではカーネルループ中でホットパスでない方のパスへの条件分岐が起こった場合，その分岐命令以降で，かつ当該分岐命令と異なるステージの命令のみを実行する．つまりカーネルループにおいて，ホットパスでなかったイタレーション以外のイタレーションに属する命令を実行する．

(2) 補正エピローグ部

このブロックでは，まず図 2中のイタレーション $(i-2)$ のステージ 4 を実行する．次にイタレーション $(i-1)$ のステージ 2 の条件分岐で分岐したので，イタレーション $(i-1)$ のホットでない方のパスを実行する．さらに，イタレーション i のステージ 2,3,4，もしくはステージ 2 で分岐が起こった場合はホットでない方のパスを実行する．このブロックを実行する際には元のループのコードを用いている．つまり，条件分岐の then 部と else 部を含んでいるコードを用いている．それは，仮に補正コードを実行中にホットでないパスが実行される場合に，特別な復帰のための仕組みを必要としないためである．

(3) プロローグ部

上の 2 つのブロックの実行が完了すると，イタレーション i までは実行が完了している．そこで，このブロックではカーネルループに戻るために再びプロローグを実行する．これにより，時刻 $(t+4)$ のカーネルループに復帰することができる．

基本的なコード変換はこれで完了だか，細かい点がまだいくつか不足している．以下ではこれらについて説明する．

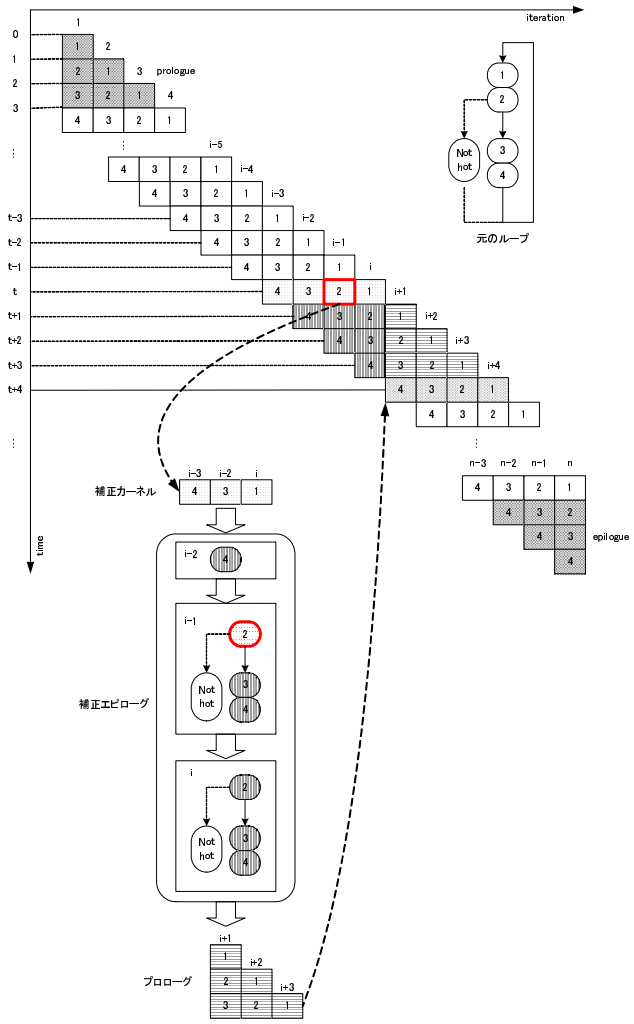

図 2: 動的 SWP の実行の様子

2.3.2 終了判定

通常の SWP では，ステージ数 S，ループの繰り返し Nのとき，カーネルループが $(N-S+1)$ 回実行されると，エピローグへと移行し，実行は完了する．しかし本手法ではカーネルループだけでなく，補正コードを実行しているので，終了判定にも工夫が必要である．

図 3にその例を示す．補正コード 1 回の実行は S 段分のカーネルループを実行することと同じである．

- 図 3(a) の場合
  カーネルループのイタレーションが $(N-S+1)$ の時（図 3では $S=4$）補正コードへの分岐が起こったとしたら，カーネルループのイタレーション Nまで実行が完了するので，そのままエピローグを実行して終了すればよい．

- 図 3(b) の場合
  カーネルループのイタレーションが $(N-S+2)$ の時補正コードへの分岐が起こったとしたら，まず補正エピローグ部を実行する．この時点でノーマルループのイタレーション $(N-S+2)$ まで実行か完了するので，その後は元のループを $(S-2)$ 回実行すればよい（図 3の例では元のループのイタレーション $N-1$,Nの 2 回）．

- 図 3(c) の場合
  カーネルループのイタレーションが $(N-S+3)$ の時補正コードへの分岐が起こったとしたら，まず補正エピローグ部を実行する．その後は元のループを $(S-3)$ 回実行すればよい．以下同様に，$(S-k)=1(1 \leq k \leq S-1)$ になるまで続ける．つまり，カーネルループのイタレーションが $(N-S+k)$ の時補正コードへの分岐が起こったとしたら，まず補正エピローグ部を実行する．その後は元のループを $(S-k)$ 回実行すればよい．

- 図 3(d) の場合
  カーネルループのイタレーションが Nの時，補正コードへの分岐が起こったとしたら，補正エピローグ部を実行すれば完了である．

以上で述べた終了条件を補正エピローグの最後に付加する．

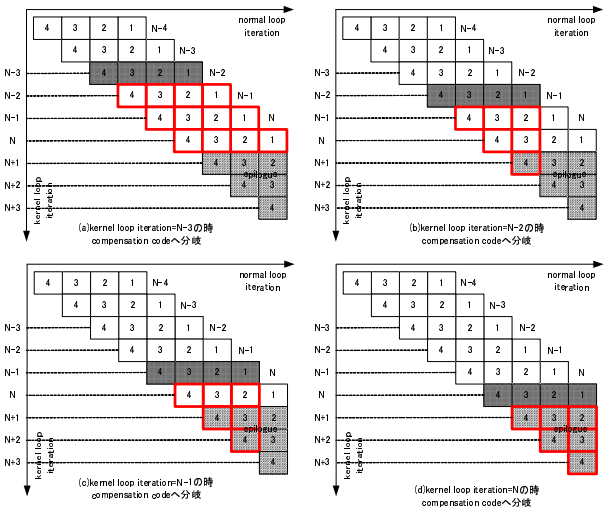

図 3: 動的 SWP の終了条件

2.3.3 分岐情報を記憶

本手法では，図 4の示すように，補正カーネル部の数はすべてのパスに対して用意するため，条件分岐の数に対して指数関数的に増加する．そこで，コードサイズを抑えるために変形エピローグ部は 1 つしか生成しない．そのための工夫として，図 4の例で示すように，分岐の情報を補正カーネル部の先頭で，適当なレジスタに確保する（ここではレジスタ rx）．補正エピローグ部では，そのレジスタの値をみて実行するべきコードを選択する．

例えば，図 4のような例を考える．図 4(c) において，カーネルループ中の Cmp3 で分岐が起こったとする．まず補正カーネルのブロック 1 の先頭において，レジスタ rx に分岐した情報を記憶する．次に補正エピローグにジャンプする．補正エピローグのブロック 1 では，Cmp3’ でレジスタ rx の値をみて Cmp3 で分岐が起ったことがわかるので，ホット

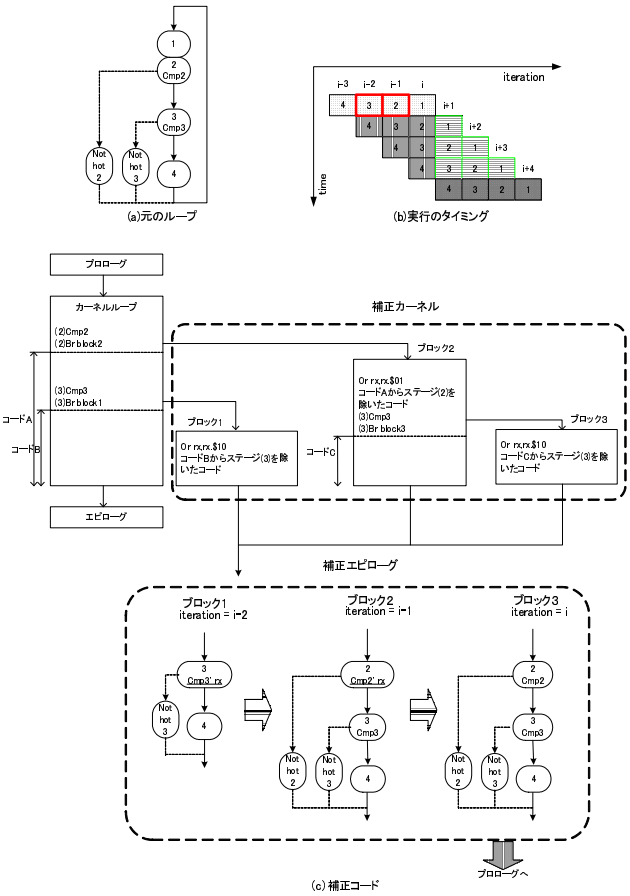

図 4: 分岐情報の記憶

パスでない方の分岐先である Not hot 3 から実行する．補正エピローグのブロック 2 では，Cmp2' でレジスタ rx の値をみて Cmp2 で分岐が起こってないことがわかるので，ステージ 3$[i-1]$ から実行される．補正エピローグのブロック 3 は通常のループを実行する時と同じように実行され，プロローグへジャンプする．図 4(c) の例では，変形エピローグのブロック 2 とブロック 3 が同じ形に見える．しかし，ブロック 2 のステージ 2 では Cmp2 命令より上に位置する命令はすでに実行済であるが，ブロック 3 の方はイタレーション i のステージ 1 しか実行されておらず，ステージ 2 はまだ何も実行されていない状態である．

2.3.4 プロローグ，エピローグ部から補正コードへの分岐

基本的な考え方はカーネルループからの分岐のときと同様．プロローグとエピローグにあるすべての条件分岐に対し，補正コードを生成する．

3. 性能評価

3.1 変換後のコードの速度向上

3.1.1 評価方法

変換後のコードの速度向上について，ハイアラキカル・リダクション,IF 変換, 逆 IF 変換, などの既存の手法との比較を行い，動的 SWP の性能を見積もった．ここでは，予備評価として人手で変換できるようなコードサイズが小さいプログラムに対して，評価を行った．評価のためのサンプルプログラムとして図 5のプログラムを用いた．またスケジューリングを行う際には，以下のマシンを仮定してスケジューリングを行った．

- 同時に発行できる命令数は 1 とする．
- Load 命令と Mult 命令は 3 サイクルで完了する．
- Add 命令は 2 サイクルで完了する．
- その他の命令は 1 サイクルで完了する．

また，図 5のサンプルプログラムでは，コード中の条件分岐がどちらに分岐するかわからない．そこで，ここではどちらに分岐するかをランダムに決めた．この時，ループ中に複数（図 5では 2 つ）存在するパスの中から 1 つのパスをホットパスとみなし，そのホットパスへ分岐する確率が 100%から 0%まで変化したときの，変換後のコードの実行サイクル数を求めた．

```
for(i=1;i<=100000;i++){
  if(A[i]>a) C[i]=b*(A[i]+a)^2;
  else C[i]=c*A[i];
}
```

(a)元のソースプログラム

```
1   L0:  Load    r4,A[i]     ;r4=A[i]
2        Nop                 ;Loadの完了待ち
3        Nop                 ;Loadの完了待ち
4        Cmp     r4,r1       ;A[i]とaの比較
5        BrLE    L1          ;A[i]<=aならL1へ
6        Add     r4,r4,r1    ;r4=A[i]+a  ここからthen部
7        Nop                 ;Addの完了待ち
8        Mult    r4,r4,r4    ;r4=(A[i]+a)^2
9        Nop                 ;Multの完了待ち
10       Nop                 ;Multの完了待ち
11       Mult    r4,r2,r4    ;r4=b*(A[i]+a)^2
12       BrA     L2          常に分岐する  ここまでthen部
13   L1: Mult    r4,r3,r4    ;r4=c*A[i]  ここからelse部
14       Nop                 ;Multの完了待ち  ここまでelse部
15   L2: Nop                 ;(両方からの)Multの完了待ち
16       Store   r4,C[i]     ;C[i]=b*(A[i]+a)^2 | c*A[i]
17       Check_loop_index    ;ループの終了条件
18       Br      L0          ;判定結果により分岐
```

(b)アセンブリコード

図 5: SWP 適用前のコード

3.1.2 評価結果と考察

図 6に分岐の偏りを変化させた時の，元のコードに対する速度向上比の変化を示す．これによれば，分岐の偏りが約 80%あれば動的 SWP の手法は，ハイアラキカル・リダクションよりも速度向上が見込める．これは，動的 SWP の方がハイアラキカル・リダクションよりも SWP を行う際のスケジューリングの自由度が大きいためである ．しかし分岐の偏りがなくなるにつれ動的 SWP の性能は落ちている．これは動的 SWP の場合，ホットパス以外のパスを実行する際の補正コードで，特別な処理を行っているためにオーバヘッドがかかってしまっているためである．

IF 変換, 逆 IF 変換との比較では，then 部をホットパスとみなしてスケジューリングした場合はピークのスピードは同等であるが，else 部をホットパスとみなしてスケジューリングした場合は分岐の偏りが約 80%以上（図 6では 20%以下の部分）であれば速度向上が見られる．これは，IF 変換, 逆 IF 変換が then 部と else 部のうち，実行サイクル数の多い方に合わしてスケジューリングしているためである．

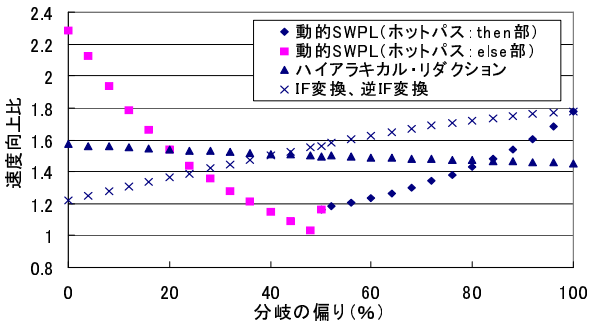

図 6: 動的 SWP と既存技術の速度向上の比較

3.2 コード変換に伴うオーバヘッド

3.2.1 評価方法

コード変換に伴うオーバヘッドとして，次のようなものが考えられる．

(1) コード変換に伴う時間的オーバヘッド

(2) コード変換の処理自身に伴う空間的オーバヘッド (変換を行うプログラムやプロファイルのためのプログラムのサイズ等)

(3) コード変換によって，変換対象プログラムのコードサイズの増加による空間的オーバヘッド

(1) や (2) のオーバヘッドを求めるには，実際にコード変換のプログラムを実装する必要があるが，今回はまだ実装ができていない．そこで本章では (3) のオーバヘッドについてのみ考察する．今回は予備評価として，いくつかの仮定を設けて動的 SWP を用いた場合と，コードサイズが問題である逆 IF 変換を用いた場合のコードサイズの増加率を式で表すことにより，その比較を行った．以下でその詳細について説明する．

仮定：

(a) ステージ数を S，ループ中の分岐の数を b，1 つの分岐命令に支配されるステージ数を d とする．支配されるステージとは，分岐命令に制御依存しているステージのことである．実際には，分岐命令に支配される d の値は，分岐命令によって異なるが，ここでは簡単化のため，どの分岐命令においても 1 つの分岐命令に支配されるステージ数は d と仮定した．

(b) 簡単化のため，SWP を行った際の各ステージに含まれる命令数は同じであるとする．この場合は各ステージに含まれる命令数を I とする．

(c) 動的 SWP によって生成されるカーネルループはノーマルループのコードサイズの $\frac{1}{k}$ 倍とする ($k \geq 1$)．カーネルループはホットパスのみをスケジューリングする．仮にある分岐の then 部のみをスケジューリングしたとすると，then 部，else 部の両方を含む元のループよりコードサイズは小さくなる．

(d) SWP によって生成されたカーネルループは各ステージが順番にスケジューリングされたとする．つまり，最初にステージ 1 が実行され，続いてステージ 2,3，… と順に実行されるとする．

(e) 対象としては，プロローグ，エピローグの部分は考慮せずカーネルループのコードサイズのみ求める．

上の仮定のもとで，動的 SWP と逆 IF 変換について分岐命令の数が変化したときのコードサイズを式で見積もる．

3.2.2 各手法のコードサイズの見積もり

- 動的 SWP
  動的 SWP における各ブロックの命令数は以下のようになる．
  - カーネルループ：$S \cdot I$
  - 補正カーネル：$\frac{1}{2} \cdot (S-1) \cdot (2^b - 1) \cdot I$
    例えば，ステージ数が 4 のとき，(d) の仮定によりステージ 1 で分岐したとすると補正カーネルに含まれるステージは 2,3,4 である．同様にステージ 2 で分岐したとすると補正カーネルに含まれるステージは 3,4，ステージ 3 で分岐したとすると補正カーネルに含まれるステージは 4，ステージ 4 で分岐したとすると補正カーネルに含まれるステージはない．すると補正カーネルの平均の大きさは $\frac{3+2+1}{4} = 1.5$ となる．これに補正カーネルの数 $2^{(\text{分岐の数}-1)}$ を掛けると補正カーネル全体のコードサイズが求まる．これを一般的に書くと，

    $$\frac{\frac{1}{2} \cdot S \cdot (S-1)}{S} \cdot (2^b - 1) \cdot I = \frac{1}{2} \cdot (S-1) \cdot (2^b - 1) \cdot I$$

    となる．
  - 補正エピローグ：$k \cdot (\frac{1}{2} \cdot S \cdot (S-1) + b \cdot S) \cdot I$
  - ノーマルループ：$k \cdot S \cdot I$

  以上の式をすべて足し合わせた式 1 が動的 SWP を行った際のコードサイズになる．

$$\{\frac{k}{2} \cdot S \cdot (S-1) + (1+k) \cdot S + k \cdot S \cdot b + \frac{1}{2} \cdot (S-1) \cdot (2^b - 1)\} \cdot I \quad (1)$$

- 逆 IF 変換
  逆 IF 変換ではコードサイズはマスクビットの情報と共に指数関数的に増大する．マスク情報は分岐命令に支配されるステージの数と同じであるので，$b \cdot d$ で表せる．よって，コードサイズは以下のようになる．

$$S \cdot I \cdot 2^{b \cdot d} \quad (2)$$

3.2.3 評価結果と考察

図 7のグラフは S，kの値を固定させ，d と b の値を変化させたときのコードサイズの増加率を示している．S，kの妥当な値として，図 5のコードに動

的 SWP を適用した場合の，$S = 4$，$k = \frac{12}{9} = \frac{4}{3}$と仮定した．図 7によれば動的 SWP もコードサイズは指数関数的に増加している．しかし，逆 IF 変換の $d = 2, 3$ の場合の比較すると，逆 IF 変換のコードサイズは大幅に増加してしまうが，動的 SWP の場合はコードサイズが d の値に依存しないのでコードサイズはそれほど増加しない．ただし，$d = 1$ の場合は，逆に動的 SWP の方がコードサイズが大きくなっている．これは，動的 SWP が分岐の数に関係なく一定の補正コードを用意するので，分岐の数が少ないときはこの部分が全体のコードサイズに対して支配的になっているためである．ハイアラキカ

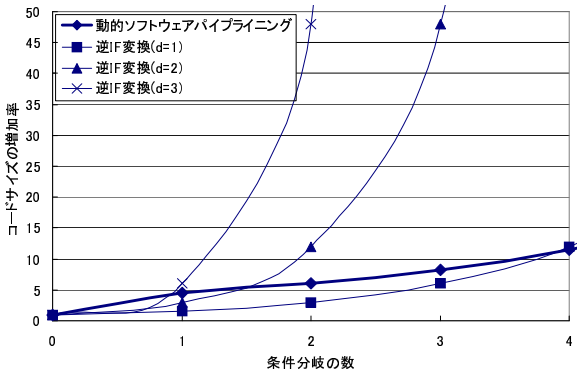

図 7: 動的 SWP と逆 IF 変換におけるコードサイズ増加率の比較

ル・リダクション,IF 変換との比較を考えると，動的 SWP の方がコードサイズは大きくなってしまうが，IF 変換のように SWP のための特別なハードウェア機構を必要としないといった利点がある．

3.3 動的 SWP の適用

ここでは，本手法の実際のベンチマークプログラムに対しての適用範囲について調べた．対象としては表 1に示す SPEC CPU 2000 整数ベンチマークを実行させて，そのトレースから始めの 200M 命令を切り捨て，その後の 300M 命令に対してパスを検出した．ただし twolf に関しては，200M 命令より前に実行が終了してしまうので，プログラムの先頭からのトレースを利用した．トレースの作成には SimpleScalar バージョン 3.0[5] を使用した．表 1中の括弧内の数字は，すべてのパスの実行数に対する，対象となるパスの実行数の割合である．また、すべてのパスの実行数の 0.1%以上実行されているパスをホットパスとみなし，動的 SWP 可能パスはこのホットパス中から検出した．これによると gzip，parser，bzip に関しては，全パス実行の 20 から 30%の部分に動的 SWP が適用できることが分かり，また，少ないパスに動的 SWP を適用することで速度向上が見込める．逆に gcc に関しては，ホットパスの実行の割合が小さく，さらに多くのパスに動的 SWP を適用しなければならないため，あまり効果は期待できない．

4. おわりに

本稿では，条件分岐を含むループに対して SWP を適用するために動的 SWP 技術を提案し，その予

表 1: ベンチマークへの適用

ベンチマーク	パス数	ホットパス数	動的 SWP 可能パス数
gzip	401	38(97.8%)	12(30.5%)
mcf	165	38(98.7%)	6(5.1%)
gcc	15874	168(45.9%)	50(17.14%)
parser	2720	126(81.5%)	15(23.7%)
bzip2	115	11(99.8%)	1(32.1%)
twolf	1556	106(87.8%)	19(19.17%)

備的評価を行った．

今回行ったのは，コードサイズの小さいサンプルプログラムに対する評価であり，評価としては不十分である．そこで，現在はベンチマークプログラムに対して評価を行えるような環境を準備している．本手法は現在のところループ中に途中から抜け出すようなパスがある場合は適用できない．これを解決する手法としては [4] で用いられている手法を適用することができると考えている．また，今回の評価は，変換後のコードの速度向上，コードサイズの増加についてのみであり，動的 SWP の一部分の影響しか考慮に入れてない．実際に動的 SWP による速度向上を求めるためには，変換後のコードの速度向上のほかに，コード変換に伴う時間的オーバヘッドについて考慮しなければならない．また空間的オーバヘッドについても，コードサイズの増加のほかに，コード変換の処理自身に伴う空間的オーバヘッドについて考慮しなければならない．

今後は，以上で述べた影響を考慮に入れ，今回提案した動的 SWP の性能の妥当性を示すことが今後の課題である．

参考文献

[1] M.Lam, “Software Pipelining:An Effective Scheduling Technique for VLIW Machines ”, In Proc. of ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation,pp.318-328,Jun.1988.

[2] J.R.Allen,K.Kennedy,C.Porterfield,and J.Warren,“Conversion of control dependence to data dependence”,In Proc. of 10th ACM Symposium on Principles of Programing Language,pp.177-188,Jan.1983.

[3] N.J.Warter,S.A.Mahlke,W.-M.W.Hwu,and B.R.Rau, “Reverse if-conversion”, In Proc. of ACM SIGPLAN ’93 Conference on Programming Language Design andImplementation, pp.290-299,Jun.1993.

[4] Daniel M.Lavery and Wen-mei W. Hwu, “Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs”, In Proc. of the 29th Annual International Symposium on Microarchitecture,pp.126-141,Dec.1996.

[5] SimpleScalar LLC, http://www.simplescalar.com/