SSA形式を利用したPredicated Execution向け命令スケジューリング手法

加藤 吉之介　　脇田 建　　佐々 政孝

東京工業大学 大学院 情報理工学研究科

概 要

VLIW アーキテクチャを採用したプロセッサでは、命令の並列度を向上させるために命令のスケジューリングが欠かせない。多くの VLIW プロセッサでは、predicated execution をサポートすることで、並列度の向上が行ないやすくなっている。predicated execution の特徴を利用した従来の研究を、並列度の低いプロセッサに対して適用すると、コード量の増加やスケジューリング結果の悪化といった欠点がある。そこで本研究では、命令スケジューリングを 2 回に分けて行なうことにより、効率的なスケジューリングを行なうアルゴリズムを提案する。提案したアルゴリズムを Open64 コンパイラへ実装し、Itanium プロセッサ上で実験を行なった。その結果、従来の手法と比較して最大 1.5%の速度向上が得られた。

Code Scheduling on SSA form for Predicated Execution

KATO Yoshinosuke　　WAKITA Ken　　SASSA Masataka

Graduate School of Information Science and Engineering,
Tokyo Institute of Technology

abstract

For processors based on the VLIW architecture, the code scheduling is indispensable in order to raise the instruction level parallelism. It is easy to improve on the instruction level parallelism in many VLIW processors supporting predicated execution. However, applying previous researches that utilize the characteristic feature of predicated execution to a processor with a low degree of parallelism is not fruitful, due to the increase in code size and the lengthy result of scheduling. To overcome these shortcomings, we propose an algorithm which performs efficient scheduling by making instruction scheduling in two phases. The proposed algorithm was implemented using the Open64 compiler and was experimented on the Itanium processor. Compared with previous techniques, we obtained an improvement up to 1.5 speed.

1　はじめに

VLIW (Very Long Instruction Word) アーキテクチャを採用したプロセッサでは、命令スケジューリングは全てコンパイラのコード生成時に行なわれる。そのため、コンパイラによる命令スケジューリングの結果がパフォーマンスに大きな影響を与える。コンパイラによるスケジューリングを行ないやすくするために、多くの VLIW アーキテクチャでは条件付き命令実行 (**Predicated Execution**) や投機的命令実行 (**Speculative Execution**) と呼ばれる機能を備えている。

Prediated execution を利用した最適化手法として、**if 変換** [1] が有名である。これはプレディケートを利用することで、条件分岐命令を削除する手法である。この手法を用いることで、複数の基本ブロックを結合し、**ハイパブロック** [2] と呼ばれる 1 つの巨大な基本ブロックを形成することが可能である。

このハイパブロックを利用した VLIW アーキテクチャに対する命令スケジューリングの研究は、今までに多く行なわれている [2, 3, 7]。しかし既存の手法は、対象とするプロセッサの命令並列度が 8 や 16 といったように、非常に高いものを想定している。そのため、これらの手法を Itanium のような並列度の低

いプロセッサに対して適用すると、いくつかの欠点がある。そこで本研究では、従来の命令スケジューリング手法の問題点を明らかにし、これを改良した新たなアルゴリズムを提案する。

2 従来の研究

2.1 インストラクション・プロモーション

ハイパブロックに対する最適化として、インストラクション・プロモーション [2] という手法がある。これは、図 1 の (a) の 3 行目、5 行目のようにプログラムの意味上必須ではないプレディケートの依存関係を解消することにより、プログラムの実行時間を決定するパス (**クリティカル・パス**) を短縮する最適化手法である。なお、本論文中で用いるコード例は全て IA-64 アーキテクチャの記法を用いる。記法の詳細については [4] などを参照して欲しい。

```
    cmp.gt p1,p2=r1,r2          cmp.gt p1,p2=r1,r2
(p1)add r3=r1,4             (p1)add r3=r1,4
(p1)sub r4=r1,r2                sub r4=r1,r2
(p2)add r3=r1,8             (p2)add r3=r1,8
(p2)sub r6=r2,r1                sub r6=r2,r1
```

(a) 通常のハイパブロック　　(b) プロモーション適用後

図 1 インストラクション・プロモーションの例

インストラクション・プロモーションが適用された命令は、元のプレディケートの値に関係なく実行されるようになる。そのため、プロモーションが適用される命令には、他の命令間の依存関係を破壊してはいけないという制約がある。[2] で利用されているコンパイラの中間表現は伝統的な形式であるため、命令間に出力依存関係や逆依存関係による依存関係が存在する。そのため、プロモーションを適用できる範囲がせまいという欠点がある。変数の名前を替えることによりプロモーションの適用範囲を拡大するアルゴリズムも提案されているが、変数の生存区間解析を行なう必要があり、複雑である。

2.2 Predicated SSA

Predicated SSA (**PSSA**) [3] では、コンパイラの中間表現に **SSA 形式** [5] を使用することで命令間の出力依存関係や逆依存関係をなくし、インストラクション・プロモーションの適用範囲を拡大することに成功している。

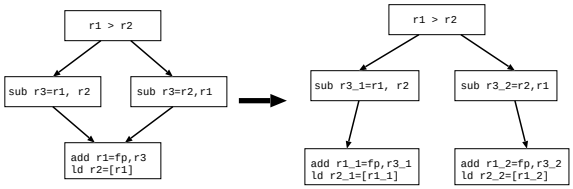

図 2 PSSA 形式への変換例

PSSA 形式への変換は、図 2 のように行なわれる。まず変数名を一意なものに変更する。PSSA 形式では、制御の合流点に複数の変数定義が到達する場合には、各パスを複製する。これにより、命令間の出力依存関係と逆依存関係を消去している。また、同時実行可能な命令数も増加させている。

この PSSA 形式を利用すると、ある命令にプロモーションを適用したことにより、他の命令間の依存関係が破壊されるということは起こらない。そのためインストラクション・プロモーションのアルゴリズムが非常に簡単になり、命令スケジューリングと同時にプロモーションを行なうことが可能となっている。

3 従来法の問題点

3.1 コード量の増加

従来の PSSA 形式を用いた最適化では、積極的に命令の複製を行なうため、コード量が非常に増加する。[3] ではコード量の増加を抑制する手法がいくつか述べられているが、命令スケジューリングの結果を向上させようとすると、それらの手法は適用できなくなってしまう。

3.2 スケジューリング結果の悪化

プロセッサの命令並列度が低い場合にも、PSSA 形式を利用した従来のアルゴリズムでは十分な最適化ができない。

```
1     ld4 r1=[a]
2     ld4 r2=[b]
3     ld4 r3=[c]
4     cmp.gt p1,p2=r1,r2
5  (p1)ld4 r4=[d]
6  (p1)ld4 r5=[e]
7  (p1)add r6=r4,r5
8  (p2)ld4 r7=[r3]
9  (p2)add r8=r7,4
10 (p1)shl r9=r6,1
11 (p2)shl r9=r8,1
12    add r10=r9,16
```

(a)

1	ld4 r1=[a]	ld4 r2=[b]	ld4 r4=[d]	ld4 r5=[e]
2	cmp.gt p1,p2=r1,r2	ld4 r3=[c]	add r6=r4,r5	
3	(p1)shl r9=r6,1	(p2)ld4 r7=[r3]		
4	(p1)add r10=r9,16	(p2)add r8=r7,4		
5	(p2)shl r11=r8,1			
6	(p2)add r12=r11,16			

(b)

図 3 PSSA 形式での最適化が困難な例

その例として、図 3 の (a) の PSSA 変換されたハイパブロックについて、並列度 4 のプロセッサでインストラクション・プロモーションによる最適化と命令スケジューリングを行なうことを考える。この場合、PSSA 形式によるアルゴリズムでクリティカル・パスを優先しながら最適化を行っていくと、図 3 の (b) がインストラクション・プロモーションと命令スケジューリングの結果として得られる。よってこのハイパブロックの実行には 6 サイクルかかることが分かる。ただし、全ての命令は 1 サイクルで実行できるとする。

しかし、インストラクション・プロモーションと命令スケジューリングを分けて行なうことで、このハイパブロックは図 4 のように 5 サイクルでスケジューリングすることが可能である。

1	ld4 r1=[a]	ld4 r2=[b]	ld4 r3=[c]	ld4 r4=[d]
2	cmp.gt p1,p2=r1,r2	ld4 r7=[r3]	ld4 r5=[e]	
3	(p1)add r6=r4,r5	(p2)add r8=r7,4		
4	(p1)shl r9=r6,1	(p2)shl r11=r8,1		
5	(p1)add r10=r9,16	(p2)add r12=r11,16		

図 4 図 3 の (a) の最適なスケジューリング結果

これは、インストラクション・プロモーションの結果、クリティカル・パスが変化したことによる。PSSA 形式によるアルゴリズムでは、クリティカル・パスが変化しても、それがスケジューリングの優先度に反映されない。そのため、スケジューリング結果はまだ最適化の余地があるものとなる。

4 改良アルゴリズムの提案

4.1 コード量の増加に対する改良

インストラクション・プロモーション後に再度コピー伝播などの最適化を施すことで冗長な命令を削除し、コード量を削減することを試みる。

4.2 スケジューリング結果の悪化に対する改良

[3] で述べられているように、PSSA 形式でのインストラクション・プロモーションは、プロセッサの並列度が十分高ければ、全ての命令を最も早いサイクルへスケジューリングすることが可能である。命令をスケジューリングできる最も早いサイクルとは、真のデータ依存関係が解消される最も早いサイクルである。よって、一度プロセッサのリソースを無限大と仮定してインストラクション・プロモーションを行なえば、スケジューリング対象のどこがクリティカル・パスなのかが判明する。

この情報を用いて再度スケジューリングすることにより、より良いスケジューリング結果を得ることを目指す。

4.3 コンパイルモデル

改良アルゴリズムを含めたコンパイル・モデルは図 5 のようになる。

1. ハイパブロックを形成
2. ハイパブロックを PSSA 変換
3. 部分冗長性除去などの最適化
4. インストラクション・プロモーションを伴うトップダウン・スケジューリング
5. 再度部分冗長性除去などの最適化
6. ボトムアップ・スケジューリング

図 5 コンパイル・モデル

5 改良スケジューリング手法

改良スケジューリング・アルゴリズムは、図 5 の 4、6 のように 2 つのフェーズからなる。以下、それぞれのフェーズを説明する。

5.1 トップダウン・スケジューリング・フェーズ

まず、プロセッサの命令並列度を無限と仮定して、命令をできるだけ早いサイクルからスケジューリングするというトップダウン・スケジューリングを行う。これにより、インストラクション・プロモーションを適用後のクリティカル・パスも正確に把握できる。

図 6 に改良したアルゴリズムを示す。なお、アルゴリズム中の op(x) は命令 x、dest(x) は命令 x のデスティネーション・レジスタ、src(x) は命令 x のソース・レジスタ、pred(x) は命令 x を修飾するプレディケート・レジスタを表す。また、fwd_earliest_schedulable_cycle(x) とは、プレディケート・レジスタの依存関係は満たさなくてよいとした場合に、命令 x を最も早くスケジューリングできるサイクルである。

```
PSSA_instruction_promotion_fwd {
  for each instruction, op(x), in the Hyperblock {
    if (pred(x) が fwd_earliest_schedulable_cycle(x) で
        定義されていない) {
      if (dest(x) の定義が複数存在する) {
        dest(x) を新しい名前へ変更する;
        dest(x) を使用している命令 op(y) を複製し、dest(x) に
        対応する src(y) を新しい名前へ変更する;
        pred(y) を pred(x) と pred(y) に対応する
        プレディケートへ変更する;
      }
      op(x) を fwd_earliest_schedulable_cycle(op(x)) へ
      スケジューリングする;
      pred(x) を常に真とする;
    } else {
      op(x) を fwd_earliest_schedulable_cycle(op(x)) へ
      スケジューリングする;
  }
}
```

図 6 改良アルゴリズム第 1 フェーズ

このアルゴリズムは、プロセッサの命令並列度を無限と見積もる以外は、PSSA 形式を利用した命令スケジューリングと同時にインストラクション・プロモーションを行なうアルゴリズム [3] と同じである。

PSSA 形式では十分な最適化が行なえなかった図 3 の (a) に対して、このアルゴリズムを適用した結果を図 7 に示す。プロセッサの命令並列度を無限と見積もることにより、図 3 の (a) の 8 行目のロード命令もプロモーションされている。この結果、各命令は可能な限り早いサイクルへとスケジューリングされる。また、インストラクション・プロモーションの適用により、クリティカル・パスが変わったことも分かる。

1	ld4 r1=[a]	ld4 r2=[b]	ld4 r3=[c]	ld4 r4=[d]	ld4 r5=[e]
2	cmp.gt p1,p2=r1,r2	add r6=r4,r5	ld4 r7=[r3]		
3	(p1)shl r9=r6,1	(p2)add r8=r7,4			
4	(p1)add r10=r9,16	(p2)shl r11=r8,1			
5	(p2)add r12=r11,16				

図 7 図 3 の (a) に改良アルゴリズムの第 1 フェーズを適用した結果 (並列度＝∞)

5.2 ボトムアップ・スケジューリング・フェーズ

図 6 のアルゴリズムにより、ハイパブロックの実行時間を最短にすることが可能である。しかし、プロセッサの命令並列度を無限と見積もっているので、このままでは対象のプロセッサ上で実行できない可能性が高い。そこで、プロセッサの並列度を考慮しながら再度スケジューリングを行う。今回は、命令を最も遅いサイクルからスケジューリングしていくボトムアップ・スケジューリング・アルゴリズムを用いる。

図 8 にアルゴリズムを示す。アルゴリズム中の bkwd_latest_schedulable_cycle(x) は、プレディケート・レジスタによる依存関係は満たさなくてよいとした場合に、命令 x をスケジューリング可能な最も遅いサイクルを表す。scheduled_cycle(x) は、図 6 のアルゴリズムによって命令 x がスケジューリングされたサイクルを表す。old_pred(x) は、図 6 のアルゴリズムを適用前に命令 x を修飾していたプレディケート・レジスタを表す。

```
PSSA_instruction_promotion_bkwd {
  for each instruction, op(x), in the Hyperblock {
    while(op(x) はスケジューリングされていない) {
      if (bkwd_latest_schedulable_cycle(op(x)) >
          scheduled_cycle(op(x))) {
        if (bkwd_latest_schedulable_cycyle(op(x)) に
            おいて、プロセッサの資源に空きがある) {
          op(x) を bkwd_latest_schedulable_cycle(op(x)) へ
          スケジューリングする;
          if (pred(x) が常に真 かつ
              old_pred(x) が定義されたサイクル <
              bkwd_latest_schedulable_cycle(op(x))) {
            pred(x) を old_pred(x) とする;
          }
        } else {
          bkwd_latest_schedulable_cycle(op(x)) を 1 減らす;
        }
      } else {
        if (scheduled_cycle(op(x)) において、
            プロセッサの資源に空きがある) {
          op(x) を scheduled_cycle(op(x)) へ
          スケジューリングする;
        } else {
          scheduled_cycle(op(x)) を 1 減らす;
        }
      }
    }
  }
}
```

図 8 改良アルゴリズム第 2 フェーズ

これにより、必要以上に早いサイクルで実行され

ていた命令を、プロセッサ資源を考慮した適切なサイクルで実行するようになるので、ハイパブロックの実行に必要な並列度を抑制できる。対象のプロセッサに、ハイパブロックを最短のサイクル数で実行するのに必要なだけの命令並列度がない場合には、クリティカル・パス長を伸ばすことによって解決する。

図 3 の (a) に第 1 フェーズを適用した結果である図 7 に対して、この第 2 フェーズを適用した結果を図 9 に示す。3.2 節と同様、対象のプロセッサの並列度を 4 とする。

図 7 では 2 サイクル目にスケジューリングされていた図 3 の (a) の 7 行目の加算命令は、3 サイクル目へスケジューリングされる。これによりこの加算命令はプレディケートの定義以降に実行されるので、フェーズ 1 により外されていたプレディケートにより再修飾する。

最終的に、PSSA 形式による最適化では図 3 の (b) のように実行に 6 サイクル必要であったハイパブロックが、改良アルゴリズムでは 5 サイクルで実行できるようになる。

1	ld4 r1=[a]	ld4 r2=[b]	ld4 r3=[c]	
2	cmp.gt p1,p2=r1,r2	ld4 r5=[e]	ld4 r7=[r3]	ld4 r4=[d]
3	(p1)add r6=r4,r5	(p2)add r8=r7,4		
4	(p1)shl r9=r6,1	(p2)shl r11=r8,1		
5	(p1)add r10=r9,16	(p2)add r12=r11,16		

図 9 図 7 に改良アルゴリズムの第 2 フェーズを適用した結果 (並列度＝ 4)

6 実験と評価

IA-64 向けの C コンパイラである Open64[6] のコード生成部を改造し、PSSA 変換と 5 節で述べた改良スケジューリング・アルゴリズムを実装した。

本研究の有効性を調べるために、2.1 節で述べた [2] による従来のインストラクション・プロモーション・アルゴリズム、2.2 節で述べた PSSA 形式によるインストラクション・プロモーション・アルゴリズムと比較を行なった。各手法の概要を表 1 に示す[1]。

手法	図 5 によるコンパイル・モデル
従来法	$1 \rightarrow 3 \rightarrow 4 \rightarrow 5$
PSSA	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$
本手法	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$

表 1 各手法の概要

評価対象は、生成された実行ファイルのサイズと実行時間である。ベンチマークプログラムとして、SPEC CINT95 ベンチマークから compress と li、go を使用した。

実験は Itanium 733MHz プロセッサ、OS Red Hat Linux 7.1 kernel 2.4.3-12smp 上で行なった。

コード量の比較結果を表 2 に、実行時間の比較結果を表 3 に示す。

	従来法	PSSA	本手法
compress	107,034	107,106	107,074
li	224,380	223,932	224,124
go	758,547	753,107	752,131

表 2 コード量の比較 (単位:Byte)

	従来法	PSSA	本手法
compress	145.8	144.6	142.5
li	11.23	11.25	11.19
go	36.43	36.29	36.76

表 3 実行時間の比較 (単位:秒)

コード量については、PSSA による手法、本手法とも積極的にコード複製を行なうが、従来法よりコード量が減少している。これは、インストラクション・プロモーション後の最適化により、冗長な命令が削除できたためであると思われる。

次に実行時間に関する比較である。実装に Open64 コンパイラを用いたため、3 手法ともハイパブロック形成前に通常の SSA 形式を用いた中間表現へと変換され、不要な依存関係は消滅している。そのため PSSA 形式と本手法による最適化の効果は、PSSA 変換時のパスの複製による、同時実行可能な命令の増加により得られていると考えられる。

compress のプログラムでは、実行時間の多くを占める部分が巨大なハイパブロックとなる。そのため、複製されるパスが多く、これによる上述の波及効果が実行時間の短縮につながったものと考えられる。それに対して li は、形成されるハイパブロックが小さく、複製されるパスも少ない。そのため、PSSA 形式では従来法より実行時間が増加している。

しかし両プログラムとも、本手法による 2 度の命令スケジューリングを行なうことで、従来法より優れた実行結果が得られている。

go において、本手法は PSSA 形式より悪化している。これは、メモリアクセス命令に関する見積もり、および、対象とする Itanium プロセッサの資源見積

[1] 実装の都合上、全手法においてインストラクション・プロモーション後の最適化を実行している。

もりが甘いために、ボトムアップ・スケジューリング時にクリティカル・パス長が増加していることによる。各命令の実行に必要なサイクル数の見積もりや、Itanium プロセッサの資源見積もりを変更することで、改善できるものと思われる。

7 まとめと今後の課題

プレディケート付き命令実行をサポートするプロセッサ向けの、既存の命令スケジューリング手法の有効性とその問題点を明らかにした。特に並列度の低いプロセッサに対して、従来のアルゴリズムを適用する場合に注目し、コード量の増加と、スケジューリング結果の悪化が問題となることを示した。これを改良する新たな命令スケジューリング・アルゴリズムを提案した。実験により、従来法と比較して最大 1.5%の速度向上を得た。コード量の増加に関しても改良法を提案し、命令の複製によるコード量の増加を抑制した。

関連研究として、本研究と同様に 2 回のスケジューリングを行なうものに [7] がある。これは効果的なレジスタ割り当てを目的とした手法である。そのため、トップダウン・スケジューリング時からプロセッサの資源制約に従ってスケジューリングしていくので、3.2 節で述べた欠点は同様に存在する。また、スケジューリング対象となる中間表現は通常形式なので、インストラクション・プロモーションを積極的に行なおうとすると名前替えを必要とする。これに対し本研究では、PSSA 形式を利用することによる簡単なインストラクション・プロモーション・アルゴリズムと、クリティカル・パスの正確な把握が可能なため、より優れた命令スケジューリング結果を得られるものと考えられる。

またハイパブロックの形成に関しては、[2] を改良した手法がいくつか提案されている。[8] では、ある基本ブロックをハイパブロックに含めた場合と通常通り実行した場合のどちらが速いかを、その都度命令スケジューリングすることによって見積もる。この結果、[2] の手法より優れたハイパブロックが形成されるが、計算量が膨大になる。本研究では命令スケジューリング時に、1 つのハイパブロックに対して 2 回スケジューリングを行なうため、この手法は適用しづらいと思われる。[9] では、各基本ブロックを縦がクリティカル・パス長、横を平均並列度 (逐次実行時のサイクル数をクリティカル・パス長で割ったもの) を持つ矩形領域として扱う。この各矩形へ、動的計画法を用いてプロセッサ資源を割り当てることにより、非常に少ない計算量でハイパブロックを形成できる。しかし、プロセッサの資源見積もりが単純であり、ALU や FPU の違いなどを表現できない。そのため、本研究の実験で対象とした Itanium のようなプロセッサには適していない。

今後の課題としては、まず投機的命令実行の実装が挙げられる。現在の実装では、ロード命令に対する投機的命令実行を行なっていない。キャッシュミスヒットによるペナルティは益々増加する傾向にあるため、ロード命令を投機的実行することはパフォーマンスの向上に大きく寄与すると思われる。

また、コード量の削減に関しては、プレディケートの依存関係を利用した、新しいアルゴリズムが提案できるのではないかと考える。

参考文献

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren: *"Conversion of control dependence to data dependence"* In Proceedings of the 10th ACM Symposium on Principles of Programming Languages, pp. 177-189, January 1983.

[2] Scott A. Mahlke, David C. Lin, William Y. Chain, Richard E. Hank, Roger A. Bringmann: *"Effective Compiler Support for Predicated Execution Using the Hyperblock"* In Proc. of the 25th Annual Intl. Symp. on Microarchitecture. pp.45-54, Dec. 1992

[3] Lori Carter, Beth Simon, Brad Calder, Larry Carter, Jeanne Ferrante: *"Path Analysis and Renaming for Predicated Instruction Scheduling"* International Journal of Parallel Programming. Vol. 28, No. 6, pp 563-586, 2000

[4] Intel. Inc: *"Intel Itanium Architecture Software Developer's Manual"*

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck: *"Efficiently computing static single assignment form and the control dependence graph"* ACM Transactions on Programming Languages and Systems, 13(4), pp. 451-490, October, 1991

[6] *"Open64 Compiler and Tools"* http://open64.sourceforge.net

[7] G. Chen and M. D. Smith: *"Reorganizing Global Scheduling for Register Allocation"* In Conference Proceedings, 1999 International Conference of Supercomputing. ACM, 1999

[8] David August, Wen-mei W. Hwu, Scott A. Mahlke: *"A Framework for Balancing Control Flow and Predication"* In Proc. of the 30th Annual Intl. Symp. on Microarchitecture. pp.92-103, Dec. 1997

[9] 田端 邦男, 小松 秀昭: *"**命令レベル並列計算機上で並列実行する領域の選択を高速に行う方法**"* 情報処理学会論文誌 Vol. 43, No. SIG1(PRO 13), pp. 97-106, 2002