ハイパフォーマンス
コンピューティング 91−17
（２００２．８．２２）

条件分岐を含むループのソフトウェアパイプライン化: IA-64 Itanium への 3 種のアルゴリズムの実装

山下 義行（佐賀大学 理工学部 知能情報システム学科）

条件分岐節を含むループのソフトウェアパイプライン化アルゴリズムとして述語付き実行を用いる方法や Enhanced Modulo Scheduling などが知られている。本稿では、これらを IA-64 Itanium プロセッサを対象に実装し、実機で評価、それらの特徴を明らかにする。この実験は、商用プロセッサ上での最初の比較実験と思われる。

Software Pipelining for Loops with Conditional Branches:
Implementing Three Algorithms on the IA-64 Itanium Processor

YAMASHITA, Yoshiyuki (Department of Information Science, Saga Univ.)

We discuss three methods of software-pipelining for loops with conditional branches; the method to utilize the predicated execution, Enhanced Modulo Scheduling (EMS), and an improvement of EMS. Actually experimenting them on the IA-64 Itanium processor, we clarify their characteristics. This is the first experimental comparison between them on a real production processor.

1 はじめに

条件分岐節を含むループのソフトウエアパイプライン化には様々な技法が考案されている。本論文ではその中でも特に高速化が期待できる、以下の技法に注目する。

(1) 述語レジスタ(predicate register)を用いた IF 変換(IF-conversion)によってコードを基本ブロック化し、それをスケジューリングし、述語付き実行(predicated execution)を行う[1]。以下、この方法を PE と呼ぶ。

(2) IF 変換後に、述語の真偽値が相反する二つの命令をモジュロテーブルの同じスロットにスケジューリングし、逆 IF 変換(reverse IF-conversion)によって述語付き実行を必要としないコードを生成し、実行する[3]。この方法を、原論文[3]に従い、EMS と呼ぶ。

(3) 立ち上げ間隔(initiation interval、以下 II)を動的に変化させる EMS の改良法が提案されている[4]。この方法を EMS$^+$と呼ぶ。

本論文ではこれら PE/EMS/EMS$^+$のコード最適化アルゴリズムを Intel IA-64 Itanium プロセッサ(以下、単に Itanium と呼ぶ)を対象に実装し、実機上で評価する。

IA-64 プロセッサを選ぶ理由は以下の通りである。

(i) 述語付き実行が可能である。

(ii) rotating register を有する。これを用いることで、recurrence のないループ(本稿ではそのようなループを扱う)は常にリソース制約から導かれる最小の II でスケジューリング可能である[2]。この性質は実験結果を理論的に解析する上で好ましい。

(iii) モジュロ・ループ用分岐命令`br.ctop`/`br.cexit`がコードの構造を簡単化する。EMS/EMS$^+$の生成するコードは非常に複雑であるため、この利点は無視できない。

Itanium 上での PE に関する実験は既に報告されている[5]が、EMS/EMS$^+$に関する実機での報告は本論文が最初と思われる。

以下、2 節では PE/EMS/EMS$^+$とそれが生成するコード例を紹介する。3 節では実験方法を述べる。4 節ではひとつの例題ループに関する実験結果を述べる。この結果は

```
DO I = 1,1000
  IF(X(I) .GT. 0.0) THEN
    A(I) = ABS(X(I)+Y(I)+Z(I)+1.0)
  ELSE
    A(I) = MAX(Y(I)-Z(I),0.0)
  ENDIF
ENDDO
```

図 1. 条件分岐を含むループの例

```
Ltop:
  (p16)ldfd f32 = [%3],8
  (p25)fadd.d f49 = f38,f1
  (p16)ldfd f40 = [%4],8
  (p25)fadd.d f47 = f35,f43
;;
  (p16)ldfd f35 = [%5],8
  (p18)fcmp.gt.unc p24,p28=f34,f0
  (p29)fsub.d f51 = f43,f38
;;
  (p30)fmax f43 = f52,f0
  (p26)fadd.d f38 = f48,f50
;;
  (p23)stfd [%6] = f46,8
  (p27)fabs f44 = f39
  br.ctop.dptk Ltop
```

図 2. PE による図 1 のコード

各アルゴリズムの特徴をそのまま反映したものになった。5 節では THEN/ELSE 節の演算コストが大きい場合を検証する。この場合、I-cache 溢れ対策が必要であることが分かった。

2 各アルゴリズムによるソフトウェア・パイプライン化

ここでは各アルゴリズムを紹介するが、それぞれを丁寧に述べる紙面の余裕はない。詳細は参考文献を見てほしい。

2.1 PE の概要

述語付き実行とそのソフトウェアパイプライン化に関する解説は文献[1]などに詳しいからここでは省略する。

図 1 の FORTRAN ループは、図 2 の IA-64 コードに変換される。このコードではモジュロ・ループ用分岐命令 br.ctop を利用し、コードを簡単化している。述語レジスタ p16～p23 は THEN/ELSE 部の外の常に実行される命令を修飾する。 p24～p27 は THEN 部の命令を、p28～p30 は ELSE 部の命令を修飾する。

```
Ltttt:
  (p16)ldfd f32 = [%3],8
  (p19)fcmp.gt.unc p28,p0=f35,f0
  (p16)ldfd f44 = [%4],8
  (p20)fadd.d f51 = f36,f48
;;
  (p16)ldfd f36 = [%5],8
  (p22)fadd.d f59 = f53,f50
  (p27)stfd [%6] = f43,8
  (p20)fadd.d f48 = f40,f1
  br.cexit.dpnt LexitT
;;
  (p25)fabs f41 = f62
  (p29)br.cond.dptk Ltttt
  br Lftttt
;;
Lftttt:
(中略)
;;
Lfffff:
  (p16)ldfd f32 = [%3],8
  (p19)fcmp.gt.unc p28,p0=f35,f0
  (p16)ldfd f44 = [%4],8
  (p20)fsub.d f54 = f48,f40
;;
  (p16)ldfd f36 = [%5],8
  (p27)stfd [%6] = f43,8
  br.cexit.dpnt LexitF
;;
  (p25)fmax f41 = f59,f0
  (p29)br.cond.dptk Ltfffff
  br Lfffff
```

図 3. EMS によるコード

このコードは";;"によって区切られる四つの命令グループを持つ。各グループは 1MC で実行できるようにスケジューリングされているから、もし分岐ペナルティが無いならば、このコードの II は 4 Machine Cycle (以下、MC)になると期待される。

2.2 EMS の概要

PE では条件判定結果を述語レジスタに保持する。これに対して EMS では同じ情報をコードのコンテキスト(実行中のコードの位

置)に表現する。換言すれば、もし PE のコードが $2n$ 個の述語レジスタを使用するならば、EMS は 2^n 種類のコード片を生成する。そして、最近の n 回の条件判定結果に従い、その 2^n 種類のコードを渡り歩く。

図 3 は、図 1 のループを EMS で最適化した IA-64 コードである。このコードは `Ltttttt`、`Lftttttt`、...、`Lfffff` などの様々なラベルから始まるコード片を持つことに注意してほしい。ラベル `Ltttttt` から始まるコード片は過去 5 回の条件判定が真であった場合に実行されるコードである。ここに分岐命令 `br.cexit` は、`br.ctop` とは逆の動作をし、ループ実行終了時に分岐を行う。 隣接する二つの分岐命令:

```
(p29)br.cond.dptk Ltttttt
br Lftttt
```

は、`p29` が真ならば `Ltttttt` に分岐し、さもなくば `Lftttt` に分岐する多重分岐である。ラベル `Lftttt` から始まるコード片は直前の条件判定結果が偽、それ以前の 4 回の判定が真である場合に実行されるコードである。ラベル `Lfffff` から始まるコード片は過去の条件判定が全て偽であった場合に実行されるコードである。紙面の制約でコード全体を載せることはできないが、実は図 3 は 2^5=32 種類のコード片を持つ。

コード片の種類は、条件判定直後から THEN/ELSE 部の全ての命令を発行し終えるまでのソフトウェアパイプライン・ステージ数の指数に等しい。指数であるため、指数爆発による I-cache 溢れの危険性をはらんでおり、これが後に問題となる。

2.3 EMS の改良

EMS の問題点は、THEN 部と ELSE 部の演算コスト(本論文では命令数)が異なる場合に演算コストの大きい方に合わせてスケジューリングを行うことである。そのため、演算コストの小さい方のコードは効率的でない。たとえば図 1 の THEN 部、ELSE 部はそれぞれ 4 個、2 個の浮動小数点演算命令を持つ。そこで、もし THEN 部に合わせてスケジューリングを行ったならば、ELSE 部では演算スロット 2 箇所が余ってしまい、これが演算効率を落とすことになる。これを解決すべく、EMS$^+$では、演算コストの小さい方に合わせてコード生成を行う[4]。

```
Ltttttt:
  (p22)fadd.d f52 = f44,f1
  (p25)fadd.d f56 = f63,f55
;;
  (p16)ldfd f32 = [%3],8
  (p28)fabs f71 = f59
  (p16)ldfd f45 = [%4],8
  (p22)fadd.d f60 = f38,f51
  br.cexit.dpnt LexitT
;;
  (p17)ldfd f39 = [%5],8
  (p21)fcmp.gt.unc p34,p0=f37,f0
  (p33)stfd [%6] = f76,8
  (p35)br.cond.dptk Ltttttt
  br Lftttttt
;;
(中略)
;;
Lfffffff:
  (p16)ldfd f32 = [%3],8
  (p28)fmax f71 = f70,f0
  (p16)ldfd f45 = [%4],8
  (p22)fsub.d f64 = f51,f44
  br.cexit.dpnt LexitF
;;
  (p17)ldfd f39 = [%5],8
  (p21)fcmp.gt.unc p34,p0=f37,f0
  (p33)stfd [%6] = f76,8
  (p35)br.cond.dptk Ltffffff
  br Lfffffff
```

図 4. EMS$^+$によるコード

図 4 は、図 1 に関する EMS$^+$のコードである。このコード中のラベル `Lfffffff` で始まるコード片の命令グループ数は 2 である。対して図 3 の `Lfffff` の命令グループ数は 3 であった。よって、偽の条件判定が

連続するような場合、図 4 のコードは図 3 よりも 1MC だけ短い II になる。ただし高速化の代償として、図 4 では 2^7=128 種類ものコード片を用意せねばならない。

EMS/EMS$^+$のコード最適化アルゴリズムは、PE と比較し、かなり複雑である。特に Itanium のように同時発行可能な命令の組み合わせが限定されている場合には、文献[3][4]に触れられていない技術的課題が新たに生じる。ここでそれを述べる余裕はない。本稿では実験結果を報告することを優先した。最適化アルゴリズムの詳細は別途報告する予定である。

3　実験方法

3.1　コード最適化器の実装

3 種類のコード最適化器を Java を用いて開発した。最適化器への入力は、IF 変換後のデータ依存グラフ(data dependence graph)、出力は IA-64 アセンブリ・コードである。命令のレーテンシ(latency)、並列実行の制約などは全て Itanium に特化して最適化した。

3.2　使用マシンと時間の計測

実験には日立製作所 HA8000-ex/480 (Itanium 800MHz×2way, MS 1GB, RedHat Linux 7.1)を使用した。

コードの実行時間は CPU のタイマー・レジスタで計測し、一万回実行した平均値を求めた。この平均値をループ長(=10^3)で割った値は II の平均値である。

同じコードを繰り返して実行するため、データと命令は共にキャッシュ・オンと考えてよい。ただし 5 節では I-cache 溢れを考察する。

3.3　真偽値パターン

ループ実行時に条件判定の真偽値がどのような順序、頻度で現れるかは、PE の実行時間には全く影響しないが、EMS/EMS$^+$には重大である。ここでは以下の定型パターンとランダムパターンを考察する。

`tttttttt..` ― 全ての判定が真である。
`ffffffff..` ― 全ての判定が偽である。
`tftftftf..` ― 真偽値が交互に現れる。
`ttfftfff..` ― 真偽値が 2 回づつ現れる。
`tttfffft..` ― 真偽値が 3 回づつ現れる。
`t...tf...f` ― 真値が続き、偽値が続く。

ランダムパターンは乱数で作り、真値の出現確率を 0.5、0.25、0.125、0.0625、0.03125、0.015625 と変えて観察する。

4　実験 1

表 1 は図 2、3、4 の実行結果である。

4.1　コードの諸元

表中の S.II (Scheduler's II)はコード生成時に最適化器が設定/参照する II である。ループに recurrence がない場合には、通常この値はリソース制約から導かれる最小の II に設定する[2]。なお、EMS$^+$では THEN 部と ELSE 部を異なる S.II でスケジューリングしたため、二つの値を載せた。

#stages は、条件判定後から THEN /ELSE 部の命令を発行し終えるまでのソフトウェアパイプライン・ステージ数である。

code size は、文字通り、生成されたコードの大きさである。表 1 の数値はいずれも Itanium の 1 次 I-cache サイズ 16Kbyte 以下であることを注意する。

4.2　定型パターンの実行時間

PE の実行は真偽値パターンに全く依存しないから、表 1 の PE の平均 II (=平均実行時間/ループ長)は全て 5.09MC である。

定型パターンでは PE は EMS/EMS$^+$よりも遅い。PE では本来実行する必要のない命令が実行スロットを占めるためである。

表 1. 図 1 コードの諸元/平均実行時間

	PE	EMS	EMS$^+$
S.II (MC)	4	3	3,2
#stages	4	5	7
code size (byte)	128	3,702	11,264
tttttttt..	5.09	4.10	4.10
ffffffff..	5.09	4.10	3.09
tftftftf..	5.09	4.10	4.15
ttfftfff..	5.09	4.11	3.87
tttfffft..	5.09	4.12	4.11
t...tf...f	5.09	4.11	3.61
ランダム(0.5)	5.09	7.51	7.97
ランダム(0.25)	5.09	6.26	6.32
ランダム(0.125)	5.09	5.29	4.88
ランダム(0.063)	5.09	4.58	3.72
ランダム(0.031)	5.09	4.40	3.50
ランダム(0.016)	5.09	4.28	3.33

実行時間の単位は 10^3MC

表 2. 図 5 コードの諸元/平均実行時間

	PE	EMS$^+$	EMS$^+$
S.II (MC)	26	14	18
#stages	4	8	4
code size (byte)	832	114,688	9,216
tttttttt..	28.25	16.39	19.27
ffffffff..	28.25	16.33	20.23
tftftftf..	28.25	16.42	19.73
ttfftfff..	28.25	16.57	19.74
tttfffft..	28.25	16.66	19.74
t...tf...f	28.25	16.86	19.75
ランダム(0.5)	28.25	92.25	23.44
ランダム(0.25)	28.25	53.54	22.14
ランダム(0.125)	28.25	32.07	21.36
ランダム(0.063)	28.25	20.22	20.65
ランダム(0.031)	28.25	18.75	20.49
ランダム(0.016)	28.25	18.16	20.39

実行時間の単位は 10^3MC

```
DO I = 1,1000
   IF(X(I) .GT. 1.0) THEN
      Z(I) = SQRT(X(I)/Y(I)+1.0)
   ELSE
      Z(I) = SQRT(Y(I)/X(I))+1.0
   ENDIF
ENDDO
```

図 5. 条件分岐を含むループの例 2

EMS$^+$は ELSE 部のコードが最適化されているため、定型パターン ffffffff..において EMS よりも約 1MC だけ高速である。

4.3 ランダムパターンの実行時間

ランダムパターンでは、分岐予測ミスのために PE が EMS/EMS$^+$よりも高速である。真偽値が等確率(0.5)で出現するときの平均分岐ペナルティは約 4MC と読み取れる。Itanium の最大分岐ペナルティは 11MC であるから、4MC は妥当な数値であろう。出現頻度が偏るに従い、平均分岐ペナルティが減少する理由は明らかである。

5 実験 2

5.1 I-cache 溢れ

条件分岐節の規模が大きくなるに従い、定型パターンでの EMS/EMS$^+$と PE の実行時間差は拡がる。もしその差が平均分岐ペナルティを上回るならば、全ての真偽値パターンにおいて EMS/EMS$^+$は PE よりも高速になるだろう。

図 5 はそれを意図したループである。IA-64 では除算、平方根の計算をそれぞれ 10 個、14 個の浮動小数点命令に分解して計算する。よって図 5 の THEN/ELSE 部はそれぞれ 10+14+1=25 個の浮動小数点命令を持つこととなり、我々の意図に十分であろう。

表 2 はその実行結果である。THEN 部と ELSE 部の演算コストが等しい時には EMS と EMS$^+$は同じコードを生成するから、表 2 では EMS$^+$のみを示した。

まず PE と EMS$^+$(S.II=14)を比較しよう。ここに S.II=14 はリソース制約から導かれる最小の II である。定型パターンでは EMS$^+$(S.II=14)が PE よりも約 1.7 倍高速である。しかし EMS$^+$(S.II=14)はランダム・パターン(0.5)では性能が極端に落ちている。原因は I-cache 溢れである。EMS$^+$(S.II=14)のコードサイズは 115Kbyte もあり、I-cache サイズ 16Kbyte を大きく越えている。定型パターンでは、115Kbyte の中の一部しか実行しないため I-cache 溢れは起きないが、ランダムパターンでは 115Kbyte の中を均等に実行するため I-cache 溢れが起きると考えられる。

5.2 I-cache 溢れ対策

EMS$^+$(S.II=14)では#stages=8 であった。つまり 2^8=256 種類のコード片が必要であり、これがコードサイズの爆発を招いた。逆に言えば、#stages を小さくするで、コードサイズは劇的に減少する。そこで解決策として以下の方法を試みる。

- 適当な k (≥1)について、コード片の種類 $2^{\#stages}$ を $2^{\#stages-k}$ に減らし、代わりに $2k$ 個の述語レジスタを新たに使用する。

すなわち、EMS$^+$に PE の手法を取り入れるのである。これによって S.II はわずかに増大するが、コードサイズは劇的に減少する。表 2 の EMS$^+$(S.II=18)は、#stages を 4 だけ減らし、代わりに S.II を 4MC 増やし、述語レジスタ 8 個を新たに使用したコードである。コードサイズは 16Kbyte 以下に収まり、その結果、全ての真偽値パターンにおいて EMS$^+$(S.II=18)は PE よりも高速になった。なお、Itanium に関する本稿の方式では以下の近似式が成り立つ。

$$\text{code size} = 32 \cdot S.II \cdot 2^{\#stages}$$

よって、コードが I-cache に納まるような最小の k 値を推定することは比較的容易な作業である。

6 おわりに

本稿では、条件分岐節を含むループのソフトウェアパイプライン化手法を実機を用いて検証した。ここで示した二つの実験結果は、PE/EMS/EMS$^+$の特徴を鮮明に描き出したと考えられる。

本稿では Itanium を対象とした。しかし 1 節(ii)、(iii)の機能を持たないプロセッサであっても、煩雑さを厭わなければ、本稿のアルゴリズムは実装可能である。

今後は、適当な C/FORTRAN コンパイラに組み込み、より詳細なベンチマーク実験を行う予定である。また最近リリースされた Itanium2 について本稿の実験結果がどのように変化するかにも興味を持っている。

参考文献

[1] Intel IA-64 Architecture Software Developer's Manual, Vol.1, http://developer.intel.com/design/itanium/index.htm

[2] B. Ramakrishna Rau : Iterative Modulo Scheduling, Inter. Jour. Parallel Programming, Vol. 24 (1996) pp.3-64.

[3] N. J. Warter, G. E. Haab, and J. W. Bockhaus : Enhanced Modulo Scheduling for Loops with Conditional Branches, IEEE MICRO-25 (1992) pp.170-179.

[4] 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 論文集 (1994) pp.17-24.

[5] Youngsoo Choi, et. al. : The Impact of If-Conversion on Branch Prediction and Program Execution on the Intel Itanium Processor, IEEE MICRO-34 (2001) pp.182-191.