計算機アーキテクチャ　114－1
（１９９５．１０．１９）

分岐頻度の評価及び高速化手法について

高山秀一　坂田俊幸　宮地信哉　富永宣輝
檜垣信生　漆原誠一　西道佳人　春名修介

takayama@isl.mei.co.jp

松下電器産業（株）松下電子工業（株）

概要

近年のマイクロプロセッサは、性能向上が著しい。その要因の一つに、分岐命令の高速化が挙げられる。一方、特に組み込み用途等では価格要求が厳しく、価格に影響しない程度のハードウエア投下で分岐命令を高速化する必要がある。本稿ではマイクロプロセッサの一例をモデルとして、各種のベンチマークプログラムにおける分岐命令の実行頻度の評価を行ない、それに基づいて考案した、ハードウエア投下量が少なくかつ高い性能向上率を得る新規高速化手法を示す。新規高速化手法はループ分岐の高速化と、関数呼出し／リターンの高速化で構成される。

ABOUT THE EVALUATION OF THE BRANCH INSTRUCTIONS FREQUENCY AND THE SPEED-UP TECHNIQUE

Shuichi Takayama　Toshiyuki Sakata　Shinya Miyaji　Nobuki Tominaga
Nobuo Higaki　Seiichi Urushibara　Yoshito Nishimichi　Shusuke Hruna

Matsushita Electric Industrial Co.,Ltd. Matsushita Electronics Co.,Ltd.

abstract

Recently, the performance improvement of the microprocessors is remarkable. It is one of the factors to shorten the executive time of the branch instructions. On the other hand, the requirement of the price is severe with the embedded use especially, and branch instructions must be sped up by the injection of the hardware which doesn't influence a price. The evaluation of the executive frequency of the branch instructions in the various bench mark programs is described to this thesis by using the model of a microprocessor. And the speed-up technique is shown, which was devised based on the above evaluation, and of which hardware quantity is small and performance improvement rate is high. A new speed-up technique is composed of the speed-up of the branch instructions of the loop portions and the speed-up of the function call/return instructions.

1 はじめに

近年、マイクロプロセッサの性能向上は著しい。その要因の一つに、多くのマイクロプロセッサにおいてほとんどの命令を1サイクルで実行できるアーキテクチャを採用したことが挙げられる。

しかし、プログラム中に占める割合の高い分岐命令は、1サイクル実行が困難である。これは一般に、分岐命令は分岐先アドレスを計算した後、分岐先命令のフェッチを開始するため分岐命令の実行開始から分岐先命令の実行開始までには数サイクル必要であるという理由による。この現象を分岐ハザードが生じるという。そのため分岐命令の高速化は性能向上に有効であり、これまでに様々な分岐高速化手法が考案されてきた。

一方近年のマイクロプロセッサ、特に組み込み用途等では価格要求が厳しい。性能向上に分岐命令の高速化は有効であるが、高速化によるハードウエアの増加は価格に影響するため避けなければならない。

しかし、既存の分岐高速化手法は高い性能向上率を得ようとすると多くのハードウエア投下が必要であったり、逆に少量のハードウエア投下によって実現できる高速化手法はコンパイラ等が有効に利用できず、性能向上率が低い場合もある。

本稿ではマイクロプロセッサの一例をモデルとして、各種のベンチマークプログラムにおける分岐命令の実行頻度の評価を行ない、それに基づいて考案した、ハードウエア投下量が少なくかつ高い性能向上率を得る新規高速化手法を示す。新規高速化手法はループ分岐の高速化と、関数呼出し／リターンの高速化で構成される。

まず2章で既存の分岐高速化手法について述べ、3章で評価対象となるマイクロプロセッサのモデルの特徴について述べ、4章で各種のベンチマークプログラムにおける分岐命令の実行頻度の評価について述べ、5章で今回考案した新規高速化手法であるループ分岐の高速化と、関数呼出し／リターンの高速化について述べ、最後に6章で既存の分岐高速化手法と新規高速化手法の評価を行なう。

2 既存の分岐高速化手法

この章では既存の代表的な分岐高速化手法である、遅延分岐、分岐予測バッファについてその概要と問題点を簡単に述べる。

2.1 遅延分岐

分岐先命令の実行の前に分岐命令の後続命令を実行する手法である。この後続命令を遅延スロットの命令と呼ぶ。遅延スロットの命令は分岐命令の実行で生じる分岐ハザードのサイクルで実行されるので分岐命令の実行サイクルが見かけ上減少する。

遅延スロットに置くことの出来る命令には制限がある [1] ため、性能向上に有効な命令を全ての遅延スロットに置くことが可能とは限らない。例えば、遅延スロットに置くことの出来る命令が存在しない場合や、遅延スロットに置くことの出来る命令が存在する場合でもコンパイラ等の性能によって移動できない場合があるためである。よって全ての分岐命令が高速化される訳ではなく、実行頻度の高い分岐命令を高速化できない場合は性能向上率が低い。

2.2 分岐予測バッファ

最新の実行結果が分岐成立なのか不成立なのかと分岐先アドレスを記憶する分岐予測バッファ [1] を使用する手法である。分岐予測バッファに記憶された最新の実行結果が分岐成立の場合、分岐命令は分岐予測バッファに記憶している分岐先アドレスから命令をフェッチする。すなわち最新の実行結果を元に分岐が成立するのか不成立なのかを予測する。

分岐予測バッファに記憶できる情報の数が多いほど性能が向上するため、多くのハードウエアが必要である。

3 マイクロプロセッサのモデル

本稿で用いるマイクロプロセッサのモデルは、データ用アドレス用各4本のレジスタを備え、ロードストアアーキテクチャを採用する。転送、算術整数演算、比較、分岐等の命令を備える。

評価のために特に重要なパイプライン、分岐命令、実行サイクルについては以下に詳細を述べる。

3.1 パイプライン

命令フェッチ - 解読 - 実行／アドレス計算 - メモリアクセス の4段とする。

3.2 分岐命令の詳細

分岐命令は次の種類を備える。

- 関数呼出し命令 （以下 JSR と略）
- 関数リターン命令 （以下 RTS と略）
- 条件分岐命令 （以下 Bcc と略、cc の種類については後述）
- 無条件分岐命令 （以下 JMP と略）

なお Bcc の cc は分岐条件を表す。例えば cc が EQ の場合、直前に実行される比較命令（以下 CMP と略）の結果、両オペランドの値が等しい場合に分岐する。cc の種類は EQ 以外にも NE （両オペランドの値が等しくない）など数種類ある。

RTS 以外は分岐先アドレスを計算して分岐する。JSR は分岐するとともに戻り番地をメモリ等の特定域に格納する。RTS は特定域に格納された戻り番地を獲得して分岐する。

3.3 実行サイクル

分岐命令以外は全て1サイクルで実行されるとする。

分岐先アドレスを計算する分岐命令は、パイプラインのアドレス計算が3段目であるため、2サイクルの分岐ハザードが生じる。よって実行サイクルは3サイクルとする。

メモリアクセスによって分岐先アドレスを獲得する命令は、パイプラインのメモリアクセスが4段目であるため、3サイクルの分岐ハザードが生じる。よって実行サイクルは4サイクルとする。

実行サイクルのまとめを表1に示す。

4 分岐命令の実行頻度の評価

上記マイクロプロセッサのモデルについて各種ベンチマークプログラム中における分岐命令の実行頻度の評価を行なった。評価に用いた手法とその結果を次に述べる。

JSR	3
RTS	4
Bcc	3 （分岐成立時） 1 （分岐不成立時）
JMP	3

表 1: 分岐命令の実行サイクル

4.1 手法

対象とするプログラムを以下に示す。

- Dhrystone benchmark test
- Stanford benchmark test [2]

これらのプログラムは C 言語で記述されている。それぞれの実行形式をシミュレータで実行し、総実行時間と各種分岐命令の実行時間を求めた。

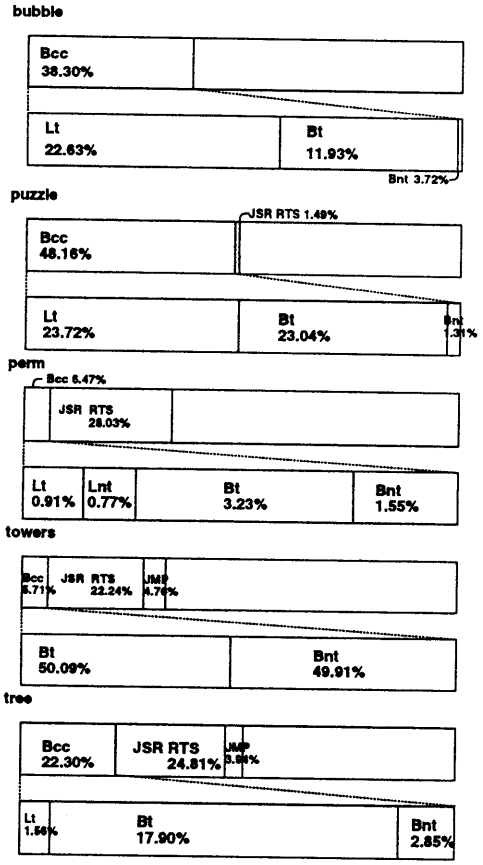
4.2 結果

各プログラムの実行時間に対する各種分岐命令の実行時間の割合を図1に示す。さらに Bcc に関してはループ処理の最初の命令（以下「ループの入口」と称す）への分岐命令とその他の分岐命令の場合に分類した。ループの入口への分岐が成立する場合を Lt、不成立の場合を Lnt と略、その他の Bcc が分岐成立する場合を Bt 不成立の場合を Bnt と略した。なお 0.5 %未満については省略した。

分岐命令の実行時間は全体の約２５～５０%と高い。その中でも特にループの入口への分岐の割合が bubble は２３%、puzzle は２４%も占める。又、perm, towers, tree は関数呼出し／リターンの割合が２０%を超える。その他でも、ループの入口への分岐と関数呼出し／リターンを合わせた割合はほとんど３０%前後である。よって、ほとんどのプログラムはループ入口への分岐と関数呼出し／リターンの実行に多くの時間を消費していることがわかる。

5 新規高速化手法

２章で述べたように、既存の分岐高速化手法はコンパイラ等の性能に依存したり、多くのハードウエアが必要である。この問題は、頻度の高



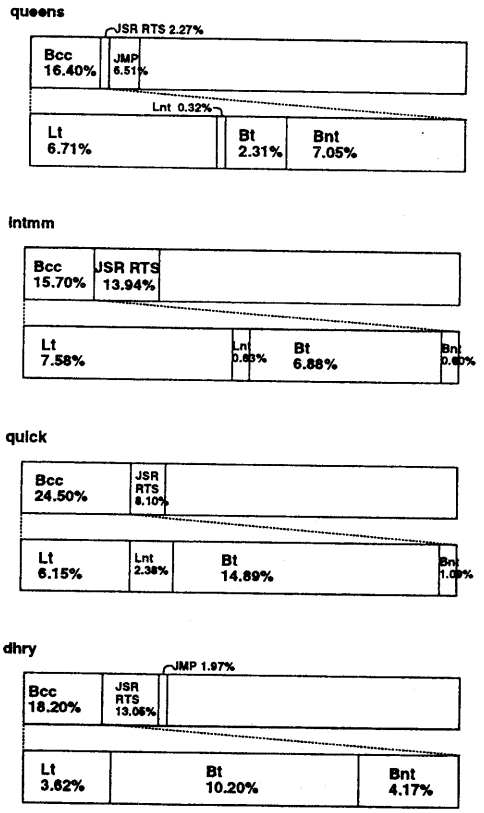


図 1: 分岐頻度

い分岐命令のみを確実に高速化することで解決できると考えた。

4 章の結果からほとんどのプログラムはループ入口への分岐と関数呼出し／リターンが多いことがわかる。この点に着目して考案した、ループ入口への分岐と関数呼出し／リターンの高速化について次に述べる。

5.1 手法 1：ループ入口への分岐の高速化

プログラム中のループ部分はコンパイラによって確実に認識できる [3]。この点に着目し、ループ入口への分岐を高速化する。以下に必要なハードウエア構成要素と、本手法を適用した実行命令列について述べる。

5.1.1 ハードウエア構成要素

次に示すような分岐先の情報を記憶するバッファを備える。

- ループ入口の命令とその次の命令のアドレスを記憶するバッファ（以下 LB と略）

このバッファは次に示す命令が使用する。

- LB にループ入口の命令とその次の命令のアドレスを記憶する命令（以下 SETLB と略）
- LB に記憶している命令とアドレスを使用して分岐する命令（以下 Lcc と略）

SETLB の実行によって命令とアドレスが記憶される。LB に記憶している命令が Lcc の分岐先命令である。又、Lcc は LB に記憶しているアドレスを使用して分岐先命令の次の命令を

```
L:
    inst1
    inst2
    :
    Bcc L
```

図2: 従来のループ命令列

```
    SETLB
L:
    inst1
    inst2
    :
    Lcc
```

図3: 本手法のループ命令列

フェッチできる。よって分岐先の命令をフェッチする必要がないので、分岐時の実行サイクルを通常の分岐命令よりも2サイクル減少できる。

ハードウエアコストに影響する要素はLBである。アドレスと命令を記憶するバッファのコストはマイクロプロセッサの全体のコストと比較すれば僅かである。

コンパイラがSETLB, Lcc命令をどのように出力するかを次に述べる。

5.1.2 実行命令列

Bcc を用いてループを構成した命令列の例を図2に示す。なお以下、instN (N = 1,2, ···) は適当な命令とする。

上記の例に対して、本手法を用いてループを構成した命令列を図3に示す。

図3に示した命令列を実行する場合、まずループ入口の直前でSETLB を実行する。この例では「inst1」と「inst2 のアドレス」を LB に記憶する。その後、Lcc は分岐成立時、LB に既に記憶した inst1 を実行し、LB に既に記憶した inst2 のアドレスを使用して inst2 をフェッチする。

```
    JSR f          f の呼出し
    :
f:
    PUSH R1        レジスタ R1 を退避
    PUSH R2        レジスタ R2 を退避
    PUSH R3        レジスタ R3 を退避
    SUB SP, lsz    サイズ lsz の
    :              ローカル領域を確保
    ADD SP, lsz    ローカル領域を解放
    POP R3         レジスタ R3 を復帰
    POP R2         レジスタ R2 を復帰
    POP R1         レジスタ R1 を復帰
    RTS            リターン
```

図4: JSR RTS

5.2 手法2：関数呼出し／リターンの高速化

コンパイラは一般に関数の入口において、関数内で使用するレジスタの退避とローカル領域の確保を行なう。この一連の処理を関数入口の処理と呼ぶ。又、関数の出口において、入口で退避したレジスタの復帰とローカル領域の解放を行なう。この一連の処理を関数出口の処理と呼ぶ。

関数入口／出口の処理が定型的である点に着目し、

- 高機能関数呼出し／リターン命令
- ファイル間関数情報指定コンパイラシステム

によって関数呼出し／リターンを高速化する。

以下高機能関数呼出し／リターン命令の詳細と、ファイル間関数情報指定コンパイラシステムについて述べる。

5.2.1 高機能関数呼出し／リターン命令

3章で述べたマイクロプロセッサのモデルについて、関数呼出しと関数入口／出口の処理の一例を図4に示す。この例では、関数 f で退避／復帰するレジスタは R1,R2,R3 である。なお関数 f のローカル領域は領域の先頭のアドレスを保持するレジスタ SP を必要なサイズだけ減じて確保される。

関数入口／出口処理における関数毎の相違点は、退避／復帰するレジスタの種類とローカル

```
    CALL f,lsz,[R1,R2,R3]   f の呼出しと
    :                       レジスタの退避と
    :                       ローカル領域の確保
f:
    :
    RET lsz,[R1,R2,R3]      関数リターンと
    :                       レジスタの復帰と
    :                       ローカル領域の解放
```

図 5: CALL RET

領域のサイズである。すなわち図 4中で PUSH、POP するレジスタの種類が違ったり、lsz の値が違う。この点に着目し、関数への分岐と共に関数入口処理を行なう命令 CALL と関数リターンと共に関数出口の処理を行なう命令 RET を考案した。

両命令のフォーマットは以下の通り。

- CALL label, lsz, [Ri,…,Rj]
- RET lsz, [Ri,…,Rj]

label は分岐先の関数名を表す。lsz はローカル領域のサイズを表す。[Ri,…,Rj] は退避／復帰するレジスタを表す。

図 4に示した例に対して CALL ／ RET 命令を適用した命令列を図 5 に示す。レジスタの退避とローカル領域の確保は CALL 命令で行なわれるので、図 4に示した入口処理の PUSH、SUB 命令は削除される。同様にレジスタの復帰とローカル領域の解放は RET 命令で行なわれるので、図 4に示した出口処理の POP、ADD 命令は削除される。

JSR ／ RTS は分岐ハザードを生じるので、分岐先の命令を実行するまで何も実行しないサイクルがある。CALL ／ RET はこの何も実行しないサイクルでオペランドにより指定されたレジスタの退避／復帰とローカル領域の確保／解放を行なう。よって CALL は JSR で生じる分岐ハザードの 2サイクルを、RET は RTS で生じる分岐ハザードの 3サイクルを短縮できる。

ハードウエアコストは、命令が増加するだけなのでほとんど増加しない。

```
{
    :
    f();
    :
}
```

図 6: a.c のプログラム

```
f()
{
    :
}
```

図 7: b.c のプログラム

5.2.2 ファイル間関数情報指定コンパイラシステム

図 5に示した命令列から明らかなように、CALL ／ RET 命令にはレジスタとローカル領域のサイズを指定する必要がある。RET の場合はコンパイラによって容易に指定することができる。しかし分割コンパイルを許す場合、CALL の呼び出す関数が別のファイルに記述されていると、呼び出す関数のレジスタやローカル領域のサイズがコンパイラでは決定できない問題がある。

例えば、関数 f の呼出しをファイル a.c に、関数 f の処理をファイル b.c に記述した場合である。その C 言語で記述した例を図 6、図 7に示す。

上記問題は、

- 生成オブジェクト中に、関数の退避／復帰すべきレジスタとローカル領域のサイズの情報を生成するコンパイラ
- 複数のオブジェクトを結合する際、CALL のレジスタとローカル領域のサイズを指定するリンカ

を考案したことで解消できた。このようなコンパイラとリンカのシステムを「ファイル間関数情報指定コンパイラシステム」と称す。

以下、ファイル a.c、b.c を例にファイル間関数情報指定コンパイラシステムの処理を説明する。

まず、ファイル a.c、b.c のプログラムを分割コンパイルした結果を図 8、図 9に示す。なお

```
        :
        CALL f
        :
```

図8: a.c のオブジェクト

```
        :
f:      .FUNCINFO lsz,[R1,R2,R3]
        :
        RET lsz,[R1,R2,R3]
        :
```

図9: b.c のオブジェクト

以下便宜上、オブジェクトの内容をアセンブラコードで記述する。

図8に示したオブジェクト中、CALL 命令のオペランドは分岐先のラベルのみである。

図9に示したオブジェクト中、.FUNCINFO は関数固有の情報を表し、そのオペランドは f のローカル領域サイズと退避／復帰するレジスタを表す。CALL ／ RET 命令でレジスタ退避／復帰とローカル領域の確保／解放が行なわれるので、図4中の PUSH、POP、ADD、SUB 命令は出力されない。

次にこの二つのオブジェクトをリンカによって結合する際、リンカは CALL f の残りのオペランドを .FUNCINO のオペランドから獲得し指定する。従って、リンカは図5 に示した命令列に相当するオブジェクトを生成することが可能となる。

以上のようにハードウエアに CALL ／ RET 命令を備え、ソフトウエアでファイル間に跨る関数呼出しのオペランドを解決することで、関数呼出し／リターンの高速化が実現される。

6 各種分岐高速化手法の評価

上述した既存の分岐高速化手法と新規高速化手法について、ハードウエアコストと実行性能について評価する。

6.1 ハードウエアコスト

各手法を実装するために必要なハードウエアコストは必要な専用バッファのコストと等しい。

手法	バッファ	
遅延分岐	1個	割り込み時分岐先アドレス退避用
分岐予測バッファ	n個	分岐先アドレス保持用×予測する命令数(n)
手法1	2個	分岐先アドレス保持用 分岐先命令保持用
手法2	0個	必要なし

表2: ハードウエアコスト

必要な資源を表2に示す。

分岐予測バッファ以外の手法は、高々数個のバッファで実装でき、コストはマイクロプロセッサ全体と比較すれば、微小である。分岐予測バッファは性能を十分に求めるには n の値が大きくなりマイクロプロセッサのコスト増加につながる。

6.2 実行性能

4章で述べたプログラムについて、2章で述べたマイクロプロセッサのモデルに各高速化手法を適用した場合の実行時間を求めた。高速化手法を適用しない実行時間を１００％とした場合の割合を図3と図4に示す。

なお、分岐予測バッファはハードウエアコスト増加量の許容範囲を考慮し、１０個のバッファを仮定した。

bubble, puzzle はループの入口への分岐が多い。そのため手法1が効果を示し、改善率が約１５％と高い。分岐予測バッファもループの入口への分岐について予測が当たり、ほぼ同じ改善率である。逆に遅延分岐の改善率は約１０％である。これはループの入口への分岐命令の遅延スロットに有効な命令を移動できなかったためである。

perm, towers, tree は関数の再帰呼出しが多く、ループの入口への分岐が少ない。そのため手法2が効果を示し、改善率が約１５～２０％と高い。遅延分岐の改善率は約１０～１５％である。これは JSR, RTS の遅延スロットに有効な命令を移動できたためである。分岐予測バッファは予測が外れ、towers では改善されない。

queens は分岐不成立が多い。それにもかかわらず、遅延分岐では多くの遅延スロットに分岐先の命令を移動したため、無駄な命令を実行す

	遅延分岐	分岐予測バッファ
bubble	92.43	84.58
puzzle	89.21	84.51
perm	86.30	91.07
towers	91.72	100.78
tree	96.61	88.52
queens	101.77	96.47
intmm	95.30	92.77
quick	96.15	94.41
dhry	91.44	97.48

表 3: 既存高速化手法の実行時間（単位%）

	手法 1	手法 2	手法 1 + 2
bubble	84.97	99.70	84.67
puzzle	84.35	99.01	83.36
perm	100.93	79.98	80.91
towers	100.00	84.11	84.11
tree	98.96	82.28	81.24
queens	96.17	98.38	94.55
intmm	96.21	90.04	86.25
quick	100.66	94.41	95.07
dhry	98.03	90.68	88.71

表 4: 新規高速化手法の実行時間（単位%）

ることになり、改善されなかった。

その他のほとんどのプログラムにおいても、手法 1 と手法 2 の改善率は既存手法に比較して高い。

これは、既存の高速化手法は全ての分岐命令を高速化しようとするために頻度が高い分岐命令が高速化されずに、効果が上がらないことがあるが、手法 1 と手法 2 は頻度が高い分岐命令をコンパイラが選別し、高速化された命令を適用することで高い効果を得ることができるためと考えられる。

7 おわりに

以上、分岐頻度の評価に基づいて考案したループ入口への分岐の高速化と関数呼出し／リターンの高速化について述べた。この新規高速化手法だけでも十分な性能を得ることができたが、プログラムによって新規高速化手法を適用できない分岐命令の頻度が多いものもある。今後はさらなる性能向上のため、このような分岐命令の高速化について研究していく予定である。

謝辞

日頃研究活動を指導していただいている松下電子工業 出口副参事、松下電器産業 間野主担当、坂尾主担当に感謝いたします。

参考文献

[1] John L.Hennessy, David A.Patterson : Computer Architecture: A Quantitative Approach

[2] M.A.Linton : Benchmarking Engineering Workstations , IEEE Design & Test June 1986 pp.25-30

[3] A.V.Aho, R.Sethi, and J.D.Ullman : Compilers Principles, Techniques, and Tools (1988)