計算機アーキテクチャ　101－9
（１９９３．　８．　２０）

スーパースカラ・プロセッサにおける分岐命令の並列実行

原　哲也　安藤秀樹　中西知嘉子　町田浩久　中屋雅夫
三菱電機（株）システムLSI開発研究所
〒664　兵庫県伊丹市瑞原4-1

我々はブースティング方式を採用したスーパースカラ・プロセッサSARCHの開発を行っている。3命令同時発行を行うSARCHにより、スカラ・プロセッサに対して1.75倍の性能向上が得られたが、4命令同時発行を可能にしても性能向上は1.77倍にとどまり、現アーキテクチャでは機能ユニット数を増加させても、飛躍的な向上は困難であると考えられる。そこで、さらに多くの並列性を引き出すために分岐命令の並列実行について検討を行った。

分岐命令の並列実行を行うことで、命令レベル並列性の低い基本ブロックの合併、および、プログラム全体に占められる遅延スロット数の低減が可能となり、性能向上が期待できる。

評価を行った結果、3つの分岐命令を並列に実行することにより、1分岐実行だけのスーパースカラ・マシンではあまり性能向上が得られなかったベンチマークにおいても並列性を引き出すことができ、平均で25%の性能向上を得ることができた。

また、分岐命令並列実行による分岐処理時間増加に対して、branch on condition方式を採用することにより、実行サイクル数をほとんど増やすことなく、分岐処理時間をサイクル・タイム内に収められることが分かった。

Multiple-Branch Execution in a Superscalar Machine

Tetsuya Hara, Hideki Ando, Chikako Nakanishi, Hirohisa Machida, and Masao Nakaya
Mitsubishi Electric Corporation
System LSI Laboratory
4-1 Mizuhara, Itami, Hyogo, 664 Japan
e-mail: hara1@lsi.melco.co.jp

We have proposed a superscalar architecture using boosting, which we call SARCH. SARCH with three-instruction issue shows 1.75x speedup over a scalar machine in non-numerical applications. We, however, find that extra instruction issue bandwidth is little beneficial, and consequently further architectural improvement is required for dramatic performance increase.

Execution of multiple branches is promising because multiple basic blocks with little instruction-level parallelism (ILP) can be merged into one block so that big ILP can be exploited. Furthermore, total branch penalties in a program execution can be reduced because multiple-branch execution reduces the total branch delay cycles.

In this paper, we evaluate performance improvement through multiple-branch execution in a machine with speculative execution. The results show that our branch scheme achieves 1.25x performance improvement over a superscalar machine with a single-branch execution.

Because of complexity for multiple-branch execution, cycle time penalty is concerned. Using branch on condition scheme, we find that little cycle time penalty is imposed at expense of little cycle-count increase.

1. はじめに

非科学計算プログラムにおいては、基本ブロック内の命令数は少なく、それらの間にデータ依存関係があるため、並列性は非常に低い。一方、分岐命令を越えての実行には、既存のアーキテクチャ(R3000のような通常のRISCアーキテクチャ)では限界があり、無限のリソースを与えても、命令レベル並列度(ILP: Instruction_Level Parallelism, スカラ・プロセッサを基準とした性能向上比)は約1.5[Wall 91]と小さい。

命令レベル並列の限界は、実行が必要か否かが判明する以前に命令の実行を行う"投機的実行"を行うことによって非常に大きくすることができる[Lam 92]。

投機的実行の実現方法には大きく分けて2通りある。1つは、動的コード・スケジュール[Murakami 89]で、ハードウエアが実行時に命令レベルの並列性を抽出する方式である。この方式では、極めて複雑なハードウエアが必要であり高速化が難しく、また、スケジュール範囲がフェッチされた命令と限られているので十分な並列性の抽出が難しい。

これに対して、静的コード・スケジュールでは、コンパイラがコード・スケジュールを行い、ハードウエアは投機的実行による副作用を取り扱う簡単なサポート機能だけを実装する。この方式では、実行時に並列性の抽出を行う必要がないので、単純なハードウエアで実現できる。また、スケジュールの対象範囲をプログラム全体とすることで、広い範囲からより多くの並列性を抽出することができる。

ブースティング方式[Smith 90]は、静的コード・スケジュールの代表的な方式である。ブースティング方式では、コンパイラが分岐を越えて移動させた投機的実行される命令にラベルを付ける。ハードウエアは、投機的な命令の実行結果を、一旦バッファリングしておき、分岐の方向が決定した時点で、ラベルをヒントに有効／無効化する。この方式は、単純なハードウエアで実現できるのでサイクル・タイムのペナルティは少ない。

我々はブースティング方式を採用したスーパースカラ・プロセッサSARCHの開発を行っている[Ando 93]。SARCHは3つのALU命令、1つの分岐命令、1つのロード／ストア命令の中から3命令同時発行可能である。投機的な実行結果を保持するバッファを持たず、ブースティングは結果が書き込まれるまでに無効化可能なサイクルに限定している。また、静的分岐予測結果は用いず、分岐の両側からのコード移動を行っている。

このSARCHの命令レベル並列度は1.75[中西 92]を得ることができた。しかし、ALU数を4にして4命令同時発行を可能にしても、命令レベル並列度は1.77にとどまり、現アーキテクチャでは飛躍的な向上は困難であると考えられる。

そこで、さらに多くの並列性を引き出すために分岐命令の並列実行について検討を行った。

本稿では、ブースティング方式と遅延分岐方式を採用した4命令並列スーパースカラ・プロセッサ(4命令発行のSARCH) をスーパースカラのベースモデル（これを1分岐マシンと呼ぶ）にして分岐命令並列実行の検討について述べる。

まず、分岐命令並列実行の効果について述べ（2章）、次に、分岐命令並列実行の方式について説明する（3章）。そして、性能評価（4章）を行った後、本稿のまとめを行う（5章）。

2. 分岐命令並列実行の効果

分岐命令の並列実行によって以下の2つの効果が期待される。

(1)分岐命令の移動による基本ブロックの合併

ブースティング方式では、分岐命令を越えた命令の移動を行い命令レベル並列性の向上を図っている。ところが、分岐命令はブースティングにおける命令の無効化／有効化の制御を行うため、分岐命令をそれ自身が所属する基本ブロックから他の基本ブロックの中へ移動することはできなかった。言いかえれば、基本ブロック自体は命令のスケジュールの前後で維持されていた。

このため、後続ブロックからの命令移動が行えるブロックは命令レベル並列性を高くすることができたが、後続ブロックに移動可能な命令が存在しない等の原因で移動命令の数が不十分であった場合、そのブロックの命令レベル並列性を高くすることができなかった。

このような後続ブロックからの移動可能な命令が不足している基本ブロックにおいては、そこに含まれる命令は分岐命令と非常に少数のその他の命令のみである。最悪の場合では、分岐命令のみとなる。

分岐命令の並列実行機構により分岐命令の移動を可能にすると、複数の基本ブロックを合併することができ、命令レベル並列性の低い基本ブロックをなくすことができる。

(2)遅延スロット数の低減

分岐ペナルティを抑える方式として遅延分岐方式がある。この方式は分岐先命令をフェッチして

いるサイクルにも命令の実行を行うために、分岐命令の後続に遅延スロットを設けその中にコンパイラによって命令を移動させるものである。遅延分岐方式は、分岐先バッファ方式等の他の分岐ペナルティ低減の方式に比べて、ハードウエア量が非常に少ないという利点があるが、一方、遅延分岐方式が有効であるかどうかは、遅延スロットがいかに多くの命令で埋められるかで決定される。

前述のように、後続ブロックからの移動可能命令が少ない場合、遅延スロットをすべて埋めることは難しい。

SPECintを解析した結果［Cmelik 91］、分岐命令の頻度は24.6%であり、約4命令に1命令が分岐命令となる。4命令の並列実行が可能なスーパースカラ・プロセッサでは、毎サイクルごとに分岐命令が出現することになり、1サイクルおきに遅延スロットが生じる。すなわち、遅延スロットが占めるサイクル数は全体の50%となる。これらの遅延スロットを有効に埋めることは不可能であり、性能向上を妨げる大きな原因になると考えられる。

遅延スロットのサイクルにおける出現頻度は、一般に次の式で表せる。

分岐命令の頻度×並列度／分岐命令の並列度

分岐命令の並列実行を行うと全サイクル数に対する遅延スロットのサイクル数を減らすことができ、性能向上が期待できる。

3. 分岐命令並列実行

3.1 分岐命令並列実行の例

分岐命令の並列実行は、Multiflow Computer, Inc. のTRACEに採用されている[Colwell 87]。TRACEは、Trace Scheduling手法を用いたコンパイラが生成するコードを効率的に実行できるように設計されたVLIWマシンである。

TRACEの1命令語は2つの分岐のフィールドを持ち、1サイクルで2つの分岐命令の並列実行ができる。2つの分岐テスト命令はそれぞれのコンディションが格納されているbranch bankレジスタを参照し、その結果がいずれも0のときはfall-through命令（プログラム・カウンタの次の命令）が実行される。いずれかが1のときは優先順位の高い命令（つまり、逐次実行したときにより早く実行される命令）に基づいた分岐がなされ、優先順位の低い命令の分岐は無視される。

TRACEでは遅延分岐方式を採用していないが、頻繁に実行されるパス上の条件分岐命令についてnot-taken側が多く実行されるようにコンパイラが分岐条件の設定をやり直し、分岐する回数を抑えて分岐先命令フェッチ遅延によるペナルティを少なくしている。

3.2 分岐命令並列実行における選択肢

3.2.1 likelyパス

命令流は分岐命令によって、not-taken（分岐しない）、taken（分岐する）の2つのパスに分れる。このパスのうち多く実行される方のパスをlikelyパスと呼ぶ。静的コード・スケジューリングでは、コンパイル時にサンプル・プログラムの実行などによってlikelyパスの予測を行う。

複数の分岐命令を実行するにあたり、likelyパスに関して、以下の2つの選択肢がある。

(1)likelyパスは固定：

lilkelyパスをたどり高い確率で実行される基本ブロックのあつまりをトレースに設定し、トレース内の分岐命令はそのlikelyパスがnot-takenパス となるように分岐条件を設定し直す。各分岐命令の予測の方向が固定されているので、実行時の（分岐先を選択するための）分岐先決定処理を簡単化できる利点がある。

not-taken固定の場合、likelyパスがnot-taken側になっているのでtakenになる回数を抑えられ、遅延分岐方式を採用しない場合でも分岐先命令フェッチ遅延によるペナルティを少なくすることができる。しかし、ループを形成する分岐命令に対応できず、join点での分岐を並列実行するときに余分な分岐命令を挿入する必要がある（図1）。これ

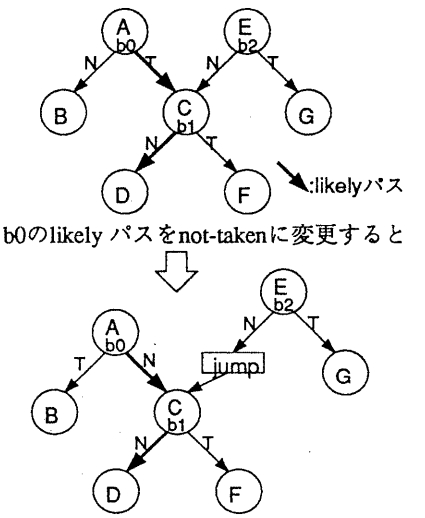

図1　likelyパス変更によるjump命令の増加

らのスケジューリングにおける制限は性能低下を引き起こす。

(2)likelyパスは任意：

トレース内の分岐命令のlikelyパスの変更は行わない。上記(1)の欠点がなく自由度の高いスケジューリングを行うことができる。しかし、分岐先の決定にlikelyパスの方向という項が入るため、処理が複雑になるという欠点がある。

分岐ペナルティは、遅延分岐方式で対処できると考えられ、likelyパスをnot-takenに固定することによって分岐ペナルティを減少させる利点はない。また、likelyパスを任意にすることによって、スケジューリングの制限から生じる性能低下を回避できる。以上の理由により、以降の検討ではlikelyパスは任意にとることとする。この方式では、分岐先決定処理が複雑になり、サイクル・タイムを延ばす可能性が考えられるが、これに関しては4章で検討を行う。

3.2.2 分岐命令並列実行数

1分岐マシンは最大4命令を同時に実行できるので、分岐命令の並列実行数について4、3、2の選択肢がある。並列に実行可能な分岐命令数が多くなると、当然分岐処理を行うハードウエアの量は増え、処理遅延時間が大きくなると共にブースティング・レベル（命令移動の時に越えることができる分岐の数）を上げる必要がありさらにハードウエアが増える。一方、並列実行可能な分岐命令数が少なくなると、十分なコード移動ができなくなる可能性がある。

並列実行可能な分岐命令の数はブースティングのレベルによって制限される。[Smith 92]によれば、効率的なブースティングのレベルは、1～3であるので、命令フィールド内にブースティング・ビットを2ビット設け、ブースティング・レベルの取り方を

・なし(0レベル)
・1レベルlikely
・2レベルlikely
・1レベルunlikely

とする（図2）

これにより、2レベルlikely上の分岐命令までブースティングできるので、並列に実行可能な分岐命令数を3とする。

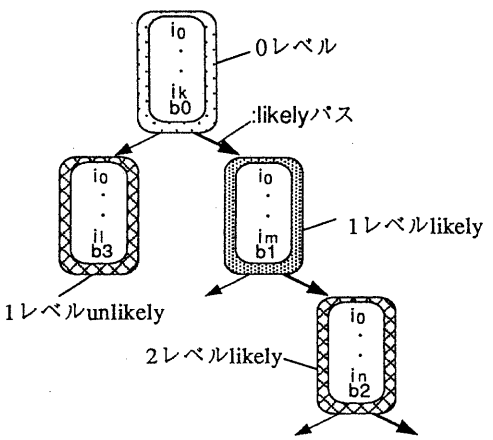

図2　ブースティング・レベル

3.3 分岐命令並列実行

3.3.1 分岐ルール

複数の分岐命令を同時に実行すると、多方向の分岐先が生じる。よって、複数の候補の中から分岐先を決定する必要がある。分岐先を選択する要素は、

・どの分岐命令を移動させたか
　（どのlikelyパスを選択したか）
・個々の命令の分岐結果（分岐方向）

であり、これらの組み合わせによって分岐先は決定される。

図3においてノードは基本ブロック、ノード内のb0～b6は分岐命令を表す。矢印は分岐命令の分岐先を示し、各分岐命令はその実行結果がnot-takenであればN、takenであればTの矢印に従って基本ブロックの実行の流れを移す。

図3に対して、3つの分岐命令を並列に実行した場合の分岐先決定のルール(分岐ルールと呼ぶ)を表1および図4に示す。

3つの分岐命令を並列実行する場合、likelyパスによって4つの分岐ルールが存在することになる。

3.3.2 処理手順

分岐命令並列実行は、コンパイラによる命令のスケジューリングと、ハードウエアでの分岐先の決定および命令の有効／無効化により行われる。

図3の分岐命令b0, b1のlikelyパスが共にnot-takenである場合（トレースがA-B-C）について分岐命令並列実行の説明を行う。

(1)命令のスケジュール（コンパイラ）

(i)トレース設定：3.2.2で述べたブースティングのレベルに従ってトレースの設定を行う。トレースは、likelyパスの方向へ基本ブロックA,B,Cと、unlikelyパスの方向へ基本ブロックI（分岐命令を除く）となる。

(ii)ブースティング設定：基本ブロックAはブース

ティング無し、基本ブロックBの命令には１レベルlikely(b1)、基本ブロックCの命令には２レベルlikely(b2)、そして、基本ブロックＩの命令には１レベルunlikely(u)の設定を行う。ただし、分岐命令はブースティング無しである。

(iii)スケジューリング：ブースティングを設定したトレースを１つの基本ブロックとみなし、スケジューリング（コード移動＆並べ換え）を行う。（図4(a)）

分岐命令はブロックの基底(実際は遅延スロットがあるのでその１つ前)のスロットに、本来先に実行されるものから順番(b0-b1-b3)に配置する。

分岐命令の移動に関して、以下のルールを設定する。

(a)分岐命令移動は、トレース上の命令に限定して下流から上流へ移動させる。したがって、図3の分岐命令b1とb2を同時にAへ移動させるような、not-takenパス、takenパスの両方からの移動は禁止である。

(b)分岐命令を含む基本ブロック内の全ての命令が、上流の基本ブロックに移動できる場合に分岐命令の移動は可能とする。

(2)分岐先の決定および命令の有効／無効化

(i)分岐ルールの選択：並列実行する分岐命令の分岐予測結果に基づいて表1に示す分岐ルールの選択を行う。b0, b1のlikelyパスがNNであるので、表1(a)の分岐ルールが選択される。

(ii)分岐方向決定：各分岐命令を実行し、それぞれの分岐方向(not-taken/taken)を求める。

以下の(iii), (iv)は分岐命令b0, b1, b3の分岐方向により決定され、同時に行われる。

(iii)分岐先決定：

(a)(b0-b1-b3の分岐方向が)NNN：分岐は行われず、fall-throughである基本ブロックDが次サイクルでフェッチされる。

(b)NNT：分岐命令b3の分岐先である基本ブロックEへ分岐する。

(c)NT-(-はdon't care)：分岐命令b1の分岐先である基本ブロックFへ分岐する。

(d)T--：分岐命令b0の分岐先である基本ブロックＩへ分岐する。

(iv)ブースティングにより投機的に実行されている命令の有効／無効化：

(a)NNN：投機的実行されている１レベルおよび２レベルlikelyの命令を有効化（コミット）し、unlikelyの命令を無効化（スカッシュ）する。

(b)NNT：likely側の命令を有効化し、unlikelyの命令を無効化する。

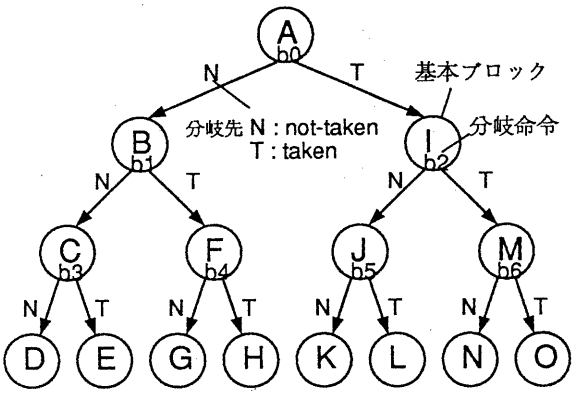

図３　分岐先決定ルール（分岐命令移動前）

表1　分岐先決定ルール(3分岐命令並列実行)

分岐方向 R0R1R2	b0b1b3 (a) NN (likelyパス)	b0b1b4 (b) NT	b0b2b5 (c) TN	b0b2b6 (d) TT
NNN	fall	fall	fall	fall
NNT	T2	fall	fall	fall
NTN	T1	T1	fall	fall
NTT	T1	T2	fall	fall
TNN	T0	T0	T0	T0
TNT	T0	T0	T2	T0
TTN	T0	T0	T1	T1
TTT	T0	T0	T1	T2

fall:fall-through　T1:b1の分岐先
T0:b0の分岐先　T2:b2の分岐先

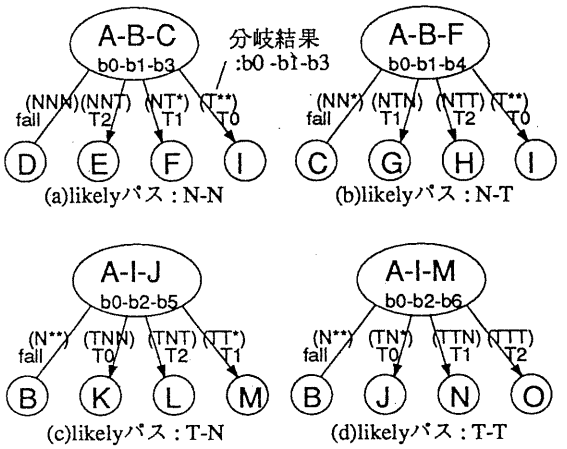

図4　分岐先決定ルール（分岐命令移動後）

(c)NT-：1レベルlikelyの命令を有効化し、2レベルlikelyとunlikelyの命令を無効化する。
(d)T--：likely側の命令を無効化し、unlikelyの命令を有効化する。

これら(i)～(iv)の処理が、likelyパスがNTのときは分岐ルール(b)、TNのときは分岐ルール(c)、TTのときは分岐ルール(d)に従って行われる。

3.3.2 処理時間

分岐遅延によるペナルティを抑えるために遅延分岐方式を用いるが、遅延スロットを1サイクルにするためには、命令フェッチが行われた後、1サイクルの間に次にフェッチする命令のアドレスを生成しておく必要がある。このためには、1サイクルで命令のデコードと分岐先決定を行わなければならず、分岐処理は時間的要求が厳しいものであると考えられる。

前項で述べたように、分岐先決定の実行時の処理は(i)分岐ルール選択、(ii)分岐方向決定、(iii)分岐先決定に分けられる。命令デコードと(i)は分岐方向決定に用いる値のレジスタ・ファイルがらの読み出しに隠蔽されると考えられるので、分岐処理のクリティカル・パスは、レジスタ読み出し→(ii)→(iii)となる。これらの処理課程は通常の分岐（レジスタ読み出し→(ii)）に比べて(iii)が加わった分だけ多くなり、サイクル・タイムを延ばすクリティカル・パスとなる可能性がある。プロセッサの性能は、（プログラム実行に必要なサイクル数）×（サイクル・タイム）で示されるので、サイクル・タイムの延びは分岐命令並列実行による実行サイクル数減少の効果を打ち消してしまう。この問題に関しては、(ii)分岐方向決定の処理時間を短縮することにより対処できると考えられる。

分岐方向を決定するための条件分岐方式には以下の2つがある。

(1)compare&branch方式

コンディション生成（レジスタの値などを用いた比較、減算等を行い、一致／不一致、正／負などのコンディションを生成する）と分岐方向決定（生成したコンディションを分岐条件でテストし、分岐するか否かを決定する）を1個のcompare&branch命令で行う。クリティカルパスが長く、分岐命令処理時間が長くなる。

(2)branch on condition方式

算術・論理演算命令またはセット命令でコンディションを生成し、分岐方向決定をbranch on condition命令で行う。branch on condition命令でレジスタ比較等を行う必要がないため、compare&branch方式に比べて分岐命令処理時間を抑えることができる。しかし、1つの分岐を処理するのに2命令必要であるために、コード・スケジューリング時に分岐を含むパスがクリティカルパスである場合にはサイクル数が増える欠点がある。

compare&branch方式をbranch on condition方式に変えることにより、コンディション生成に必要な時間相当を短縮することができる。コンディション生成では32ビット比較を行うが、この32ビット比較と(iii)分岐先決定に必要な処理時間はほぼ同じである。したがって、branch on condition方式を用いた分岐命令並列実行はcompare&branch方式の通常（1分岐）の分岐実行と同じ処理時間で行うことができ、サイクル・タイムを延ばすことはないと考えられる。

一方、branch on condition方式を用いると実行サイクル数が増加増加するが、これに関しては4章で評価を行う。

4. 性能評価

分岐命令並列実行に関して、ハンドスケジュールによる評価を行った。

4.1 評価方法

4.1.1 評価方法

サイクル数の比較による性能評価を行った。基準としたのはMIPS R3000であり、その実行サイクル数をpixieを用いて求めた。pixieはMIPSマシンで実行したプログラムの様々な統計をとるユーティリティ・プログラムで、総サイクル数の他に各基本ブロックの実行回数や分岐命令の分岐回数なども求めることができる。

1分岐マシンは我々が開発を行っているSARCHを用いて、SARCHコード・スケジューラによる最適化コードをSARCHシミュレータによって実行し、サイクル数を得た[中西92]。

分岐命令並列実行マシンは、MIPSコンパイラによるアセンブラ・コードに対して、pixieで求めた分岐確率を適用してハンド・スケジュールを行い、元コードの各基本ブロック実行回数と分岐回数から、スケジュール後コードの新たなブロックの実行回数を求め、サイクル数を算出した。

なお、ベンチマーク・プログラムとして、スタンフォード・ベンチマークを使用した。

4.1.2 アーキテクチャ・モデルおよびパラメータ

それぞれのマシンのアーキテクチャは、以下の通りである。

(1) 1分岐マシン：

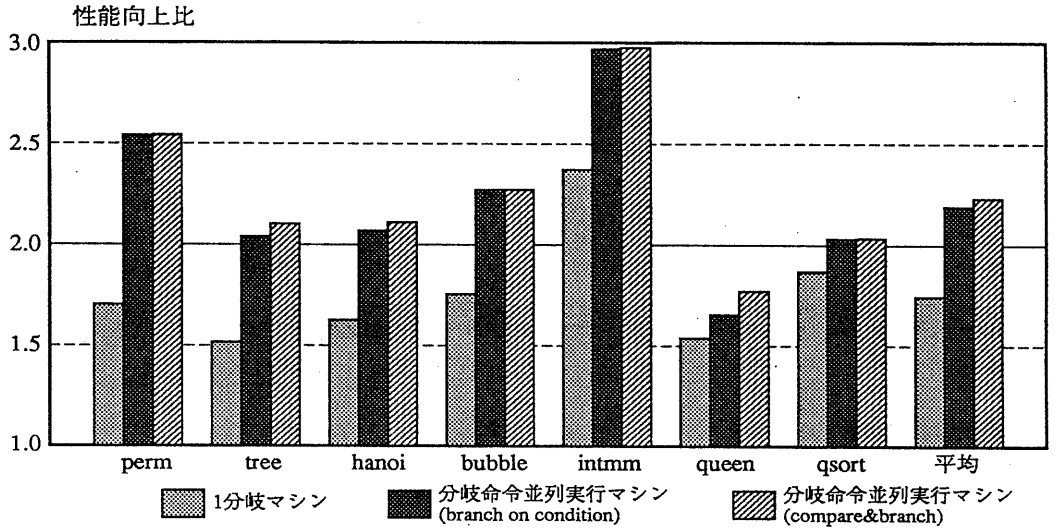

図5　評価結果

・4命令同時実行可能
・1レベルlikeky, unlikelyのブースティング
・compare&branch方式

(2)分岐命令並列実行マシン
・4命令同時実行可能
・3分岐命令同時実行可能
・1レベルおよび2レベルlikeky, unlikelyのブースティング

共にロード／ストアは1サイクルにどちらか1命令実行とする。
また、キャッシュのヒット率は100%である。

パラメータとして、条件分岐方式による実行サイクル数の違いを求めるために分岐命令並列実行マシンでは、compare&branch方式とbranch on condition方式の2つでスケジューリングを行った。

4.2 評価結果

図5に評価結果を示す。結果は、R3000において最適化を行ったベンチマーク・プログラムの実行に要した総サイクル数を基準としたサイクル数における性能向上比で表している。

4.2.1 分岐命令並列実行の効果

図5は、分岐命令並列実行の効果が大きい順に左から並べてある。

branch on condition方式の分岐命令並列実行では、1分岐マシンに比べて、幾何平均で25%の性能向上が得られている。

分岐命令並列実行の効果が大きいperm、treeと効果が小さいqueen、qsortについて考察を述べる。

・効果が大きいベンチマーク

-perm：最も多く実行する基本ブロックを分岐並列によりまとめることができたことにより性能が向上している。

-tree：中核(多くの実行回数を占める)のプロシージャがその中に多くのプロシージャ・コールを含み、そこで命令流が切れている。そのため、スーパースカラ・マシンではブースティングによるコード移動が十分に行えず、基本ブロックが多くのnopで占められていた。分岐命令並列実行を行うとnopが多い基本ブロックを効果的にまとめることができ、性能が向上している。

・効果が小さいベンチマーク

-queen：ループ内にプロシージャ・コールがあるパスもあるが、そこはあまり通らず(1752中224(12%))、命令流の切れが少ないので1分岐マシンでもブースティングによる命令移動で対処できている。また、ロード／ストア命令が多く（85命令中45）そもそもの並列性があまり高くないので、分岐命令並列を行っても1分岐マシンが引き出した並列性以上を引き出すことはできていない。

-qsort：中核のプロシージャが、外側にループがありその中にあるif-then-else文のthenパートに内側ループがある二重ループ構成であり、1分岐マシンにおいてもかなり効率のよいスケジューリングができていた。また、大きな枝別れがなく分岐命令並列実行があまり活用できていない。

一般に、プログラムが枝別れの少ない（つまり、前方向条件分岐が少ない）パスから構成されている場合、ブースティングによる命令移動だけでか

なりの並列性を引き出すことができ、分岐命令並列実行による効果はあまりない。

これに対し、プログラムが、枝別れが大きい（ツリー状に広がっている）場合には、分岐命令並列実行が効果的であるといえる。

一方、プロシージャの終わりではリターン・アドレスが動的に決まるため、リータン先から命令の移動を行えず、分岐命令並列実行を行っても空きスロットが目立った。

また、分岐を並列に実行することにより、ロード／ストア命令が新たなクリティカル・パスとなる場合が見受けられた。

4.2.2 条件分岐方式の比較

branch on condition方式はcompare&branch方式に比べ、幾何平均で1.7%しか実行サイクル数が増えていない。これは、compare&branch方式を採用した場合のサイクル・タイムの増加よりはるかに小さいものと予測される。よって、branch on condition方式を用いることにより、実行サイクル数をほとんど増やさずに、分岐処理時間をcompare&branch方式の１分岐マシンと同程度、つまり、サイクル・タイム内に抑えることができる。

スカラ・プロセッサではSPECベンチマーク・リリース1.0において6%実行サイクル数が増加すると報告されているが[Cmelik91]、これに比べてサイクル数の増加小さいのは、

・スケジューリング時に全ての分岐がクリティカル・パスとなっているわけではない。
・複数の分岐を同時に実行するため、クリティカル・パスもまとめることができた。

ことが挙げられる。

6. おわりに

以上、分岐命令並列実行の方式、および、その評価を行った。

分岐命令並列実行を行うことにより、１レベルのブースティングだけのスーパースカラ・マシンではあまり性能向上が得られなかったtreeなどのベンチマークにおいても、並列性を引き出すことができ、平均で25%（対１分岐マシン）の性能向上を得ることができた。しかしながら、queenのように分岐命令並列実行を用いても性能を向上させることができないものもある。このようなものも含めて、さらに性能を向上させるためには、if-then-else文を効率的に実行するガード付き命令方式[Hsu 86]の検討や、プロシージャ・コールの末端付近での命令不足への対処を行う必要があろう。

また、分岐命令並列実行を行うと同時にロード／ストア命令の並列実行を行うとさらに性能が向上されると考えられる。

今回、並列に実行する分岐命令の数や、ブースティング・レベルの取り方(unlikelyの代わりに３レベルlikelyとする等)については述べていないが、これらについても評価する価値があろう。

また、分岐命令並列実行による分岐処理時間増加に対して、条件分岐方式の検討を行った。branch on condition方式を採用することにより、実行サイクル数をほとんど増やさずに処理時間をサイクル・タイム内に収められることが分かった。

参考文献

[Ando 93]H. Ando, C. Nakanishi, H. Machida, T. Hara, M. Nakaya, "Speculative Execution and Reducing Branch Penalty on a Superscalar Processor", IEICE Trans. Electron., pp.1080-1093, July 1993.

[Cmelik 91]R.F.Cmelik, S.I.Kong, D.R.Ditzel, E.J.Kelly, "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", ASPLOS-IV, pp.290-302, Apr. 1991

[Colwell 87]R.P.Colwell, R.P.Nix, J.J.O'Donnell, D.B.Papworth, P.K.Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," ASPLOS-II, pp.189-192, October 1987.

[Hsu 86] P.Y.T.Hsu, and E.S.Davidson, "Highly Concurrent Scalar Processing," In Proc. 13th Int. Symp. on Computer Architecture, pp.386-395, June 1986.

[Lam 92] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," In Proc. 19th Int. Symp. on Computer Architecture, pp.46-57, June 1992.

[Murakami 89]K. Murakami, N. Irie, M. Kuga, and S. Tomita, "SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-Speed Single-Processor Architecture," In Proc. 16th Int. Symp. on Computer Architecture, pp.78-85, June 1989.

[Smith 90]M. D. Smith, M. S. Lam, and M. A. Horowitz, "Boosting Beyond Static Scheduling in a Superscalar Processor," In Proc. 17th Int. Symp. on Computer Architecture, pp.344-355, May 1990.

[Smith92] M. D. Smith, M. S. Lam, and M. A. Horowitz,"Efficient Superscalar Performance Through Boosting," ASPROS-V, pp.248-259, October 1992.

[Wall 91] D. W. Wall, "Limits of Instruction-Level Parallelism," ASPLOS-IV, pp.272-282, April 1991.

[中西92]中西、安藤、町田、中屋,"スーパースカラ・プロセッサ-SARCH-のコード・スケジューラ" 信学技報、CPSY92-38/ICD92-78 (1992年10月)