社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

高度パイプライン化SMTプロセッサ上のトレースに基づく複数パス実行方式

高木　将通†　　平木　　敬†

† 東京大学 大学院理学系研究科 情報科学専攻
〒 113-0033 東京都文京区本郷 7-3-1

E-mail: †{takagi-m,hiraki}@is.s.u-tokyo.ac.jp

あらまし　パイプライン段数増加に伴い増大する分岐予測ミスペナルティを軽減するため複数パス実行を行う．投機スレッドの fork/join の高速化のため SMT を利用し，effective fetch rate の向上のため trace を単位として複数パス実行を行う．効率の良い fork のため，パス候補の実行確率の動的な予測を用いる fork のポリシーを提案し，シミュレータを用いこのアーキテクチャを評価する．結果として，複数パス実行を行わない場合に比べ最大で 16.0%の速度向上がみられた．また，単純なポリシーよりプログラムの特性に対応しやすく，理想的な方法に対して差が小さいことが分かった．

キーワード　複数パス実行，simultaneous multithreading，SMT，トレースキャッシュ，トレース，confidence

Multi-Path Execution on SMT for Next Trace Prediction

Masamichi TAKAGI† and Kei HIRAKI†

† Department of Information Science, Graduate School of Science, The University of Tokyo,
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033 Japan

E-mail: †{takagi-m,hiraki}@is.s.u-tokyo.ac.jp

Abstract　Branch misprediction penalty of a processor grows as the pipeline deepens. We employ multi-path execution to reduce the penalty. To reduce the cost of speculative fork or join, we use Simultaneous Multithreading (SMT). To boost the effective fetch rate which decreases on multi-path execution, we use trace as an unit of multi-path execution. Because the processor can follow only limited number of paths simultaneously, we need to choose paths which are most likely to be on the correct path. For this purpose, we introduce a fork policy which predicts the execution probabilty of a path candidate dynamically. We evalutate the proposed architecture under the proposed policy and others with a simulator. In the best case, the architecture yields 16.0% speedup over the case without multi-path execution. The proposed policy produces more stable speedup than other simple policies on the programs with different characteristics, and the policy produces almost the same speedup as the ideal policies.

Key words　multiple path execution, simultaneous multithreading, SMT, trace cache, trace, confidence

1. はじめに

現在のプロセッサはクロックサイクルの短縮化，IPC の増大によってスループットの向上を追及している．前者は半導体製造技術のプロセスルールの微細化，一つのパイプラインステージのゲート遅延の短縮によってもたらされ，後者はプログラムから並列性を抽出することによって得られる．

しかし，単純なスーパースカラのもとで深いパイプライン化を行うと分岐予測ミスペナルティが増大し，これが実効的な IPC の低下の多くを占めることになる．

この分岐予測ミスペナルティを減らす一つの方法は，分岐予測の精度を上げることである．様々な分岐予測の方法が提案されてきているが [13] [2] [14]，これらの多くは，ある分岐やそれに至るパスの軌跡の近似をハッシュ関数にかけたものをインデックスとして，過去の分岐結果を記憶するカウンタのテーブルを引く方法である．

この方法には，有限インデックスによる alias 問題，cold-miss に相当する不可避の予測ミス，軌跡からでは分からないデータ依存の分岐の予測ミス，といった問題があり，これが予測困難な分岐の存在につながる．

分岐予測ミスペナルティを減らすもう一つの方法として，この予測困難な分岐の存在を受け入れ，これらの分岐に直面した際には複数のパスをたどる方法が挙げられる．

Uht ら [4] は複数パス実行により構築される，分岐命令にはさまれた命令列を辺とする木を想定し，全ての辺の実行確率の予測から実際に実行する辺を選択する方法を提案した．この方法では各辺の実行確率を算出するために実行時の分岐の予測精度と，それを使った乗算が必要である．Uht らは全ての分岐に共通な静的な予測精度を用いて近似した．これにより実行確率の乗算の必要はなくなるが，実行する木の形は実行時の予測精度を考慮していない，固定されたものになる．

本稿では実行時の予測精度の近似 [6] からある辺の実行確率を計算し，その辺を実行するか否かを選択する方法を提案し，評価する．

提案するプロセッサは，trace をフェッチの単位とする．ここで trace とは，Trace Cache [1] における trace のことである．次の trace の予測精度が低く，かつ 2 つの後続パス候補を辿ることが有効であると判断された際に 2 つのパスを辿る．分岐の結果が判明した際に，片方のパスが無効となっても，有効であるもう片方がプロセッサ内に投入されているならば，有効なパスをフェッチしなおす必要はなく，分岐予測ミスペナルティは軽減される．

このアーキテクチャの有効性を，シミュレータを用いて評価する．特に，様々な実行パスの選択ポリシーを比較する．

本稿の構成は以下の通り．2 章で 提案する複数パス実行方式について説明する．3 章で 評価方法を述べる．4 章で 評価結果を示す．5 章で 関連研究を示す．6 章で 主張をまとめる．

2. SMT プロセッサ上のトレースに基づく複数パス実行方式

2.1 Outline of the Architecture

提案するプロセッサは，trace をフェッチの単位とする．新たに thread を増やし，複数の trace を辿り始めることを以降 fork と呼ぶ．

プロセッサは，次の trace の予測に際して，予測の精度の動的な近似を行う．予測精度が高い際には，次の trace の第 1 候補を辿る．予測精度が低く，fork が効果的であると判断された際には，fork を行い，次の trace の第 1 候補と，第 2 候補を辿りはじめる．

分岐命令の結果が判明すると，間違っているパスを squash する．このとき正しい方のパスが fork によってすでにプロセッサ内に入っていれば，それをフェッチし直す必要はなく，それゆえ分岐予測ミスペナルティは軽減される．

ここで，複数パス実行によってできる，trace 列を辺として fork により別れる木を複数パス実行の木と呼ぶ．この木は最も古くかつまだ解決していない fork を根とする．

2.2 Processor Model

提案する方式を実装するプロセッサのダイアグラムを図 1 に示す．confidence counter 部，投機発行制御部が複数パス実行に特有の部分で，その他の部分は trace cache を用いたスーパースカラの SMT プロセッサに共通の部分である．パイプライン段数は 32 段，分岐予測ミスペナルティは 28 サイクルと設定した．

2.3 Simultaneous Multithreading (SMT)

fork の際には新たな thread のためのプロセッサステートを用意する必要がある．提案する方式では，Simultaneous Multithreading (SMT) [3] の手法を用いている．これにより，共通のレジスタマップを用いることができ，fork はレジスタをコピーすること

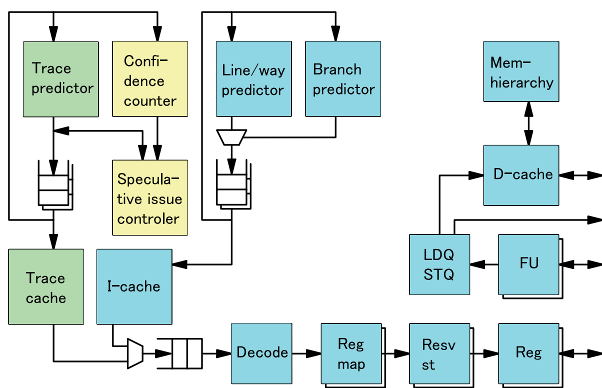

図 1　提案する方式を実装するプロセッサのダイアグラム

なく，レジスタマップをコピーすることで実行できる．これはレジスタマップに CAM を用いた場合，各 thread に対応する valid bit 間のコピーを行うこととなる．またデータ依存を共通のレジスタや load/store キューを用いて解決することができる．これらにより，複数のプロセッサエレメントがそれぞれ一つの thread を実行するアーキテクチャの場合に必要な通信が不要となる．

2.4　Trace Cache 及び Next Trace Prediction

複数パス実行時には，複数の thread から命令をフェッチする必要がある．これはフェッチユニットに大きな負担をかけ，全体としての effective fetch rate，つまりフェッチあたりの，正しいパス上にあって retire される命令数が下がる．この問題に対処するために，提案するアーキテクチャでは，Trace Cache [1] を用い，effective fetch rate の向上をはかる．この際，フェッチの単位，複数パス実行の単位は trace となる．ここで trace の名前を trace-id と呼ぶ．提案するプロセッサでは，最大 16-instruction，6 分岐命令までを含む trace を用いる．basic block を考慮せず 16-instruction まで詰めようとするため，trace の境界は basic block の境界とは必ずしも一致しない．また，レジスタを使う分岐命令があった場合にはそこで trace が終了する．この trace の作成は instruction の retire 時である．

コントロールフローの予測は trace 単位になる．この予測には次の trace-id を明示的に予測する機構 [5] を用いる．この predictor は trace-id のテーブル (correlated-table) をもつ．このテーブルには trace-id のヒストリを使ってアクセスする．このテーブルの他に，現在の trace-id を使ってアクセスするテーブル (bimodal-table) を持つ．静的に偏っている trace 列の場合はこのテーブルのみをアクセスして，correlated-table へアクセスしないようにして correlated-table の予測率向上を図っている．

提案するプロセッサはヒストリの深さが 4，各テーブルが 2^{14} エントリの predictor を用いる．またテーブルの更新は instruction の retire 時である．

複数のパスに分かれるときは，ある trace-id の次の trace-id を複数予測する必要があるが，各テーブルが次の trace-id の第 1 候補と第 2 候補を格納することにより実現する．

はじめは next trace predictor ではなく通常の分岐予測を用いて実行していて，trace を予測することにより next trace predictor を使い始めるようになる．この最初の trace の予測は instruction cache のアクセスと同時に行い，その次のサイクルにその trace を trace cache からフェッチする．それゆえ最初の trace と instruction cache から得た instruction の共通部分を捨てる必要がある．

2.5　予測精度の近似

次の trace の予測の精度の動的な近似には confidence counter の機構 [6] を用いる．提案するアーキテクチャでは，2^{14} エントリの 4-bit の resetting counter のテーブルを用いる．trace の予測の際にテーブルを引くためのインデックスと同じインデックスでこのテーブルを引く．予測が成功していた際には 1 を加え，失敗していた際には 0 にする．最後の分岐命令がレジスタを用いるものでない場合は，ある trace に含まれる全ての分岐命令の予測のどれか一つが正しくない際にその trace の予測に対するカウンタを 0 にし，そうでないときにはカウンタに 1 を加える．ある trace の最後の分岐命令がレジスタを用いるもので，予測ミスの際にはその次の trace の予測に対するカウンタの値を 0 にし，そうでないときにはカウンタに 1 を加える．このテーブルの更新は instruction の retire 時である．この resetting counter による方法は up-down counter による方法より，count 15 に含まれる trace の予測精度が高くなるという利点がある．

2.6　複数パス実行の制御

trace の作成は，fork を考慮せずに行う．fork を考慮して作成すると，複数の部分的な trace から必要な trace を再構築する必要が生じるためである．

このために fork を行った際に，trace の第 1 候補と第 2 候補が共通部分をもつ場合がある．この場合には，第 1 候補の trace は全体を使い，第 2 候補の trace は第 1 候補と異なる部分のみを使う．そして第

1 候補と第 2 候補を分ける分岐命令によって，間違ったパスを squash する．

間違ったパスを squash する際，複数パス実行の木において後続の命令を全て squash しなくてはならない．このために，各命令は複数パス実行の木における位置を覚えている．この位置は，fork に際して第 1 候補を辿ったら 0，第 2 候補を辿ったら 1 とエンコードされたビット列を用いる．この方法を使うと，木の中で位置 D が位置 A の子孫か否かは D の位置情報が A の位置情報を prefix として含むかどうかで判定できる．

この位置を用いて squash を行う．また，store キューからの fowarding のためにもこの位置情報を使う．

2.7 fork のポリシー

提案するプロセッサでは fork に使用できる thread の数は有限 (16 個) である．また，fork して同時に実行される thread 数が増加すると，フェッチユニットや実行ユニット，そしてインストラクションを収める各キューに対する負担が増大し，それぞれの thread の IPC は低下する．以上により，分岐予測ミスを回避できる数を多く，また fork 数をなるべく少なくするように選択的に fork を行う必要がある．

そのために，まず予測精度が低い予測のみ fork 候補とする．提案する方式では confidence counter の値が 15 より小さいものを予測精度が低いとみなして fork 候補とする．

さらにその fork 候補に対して fork するか否かについてさまざまなポリシーが挙げられる．

ここで，複数パス実行の木 T を想定し，各辺の実行確率を想定する．同時に辿ることのできる辺の数は有限であるので，この辺の実行確率の和が最大になるように実行したい．

このために，この確率を考慮して fork するか否かを決定するポリシーを用いる．問題となっている trace の次の trace の第 2 候補の実行確率を P とする．この値は複数パス実行の木において，その辺に至る全ての予測について，精度から得られる，第一候補あるいは第二候補が実行される確率を掛け合わせたものになる．例を図 2 に示す．ここでは予測精度は全て.7 としてあり，各辺に辺の名前と実行確率を付してある．左向きの辺が予測の第 1 候補を辿ったもので，右向きが第 2 候補を辿ったものである．例えば辺 f の実行確率は $.3 \times .3 = .09$ である．

この値を実際に計算するのは困難なので，confidence counter の値がある閾値を越える予測について，第二候補の方を辿った数を数える．このときの

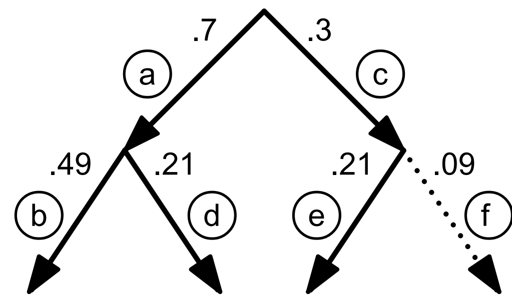

図 2　複数パス実行の木の例．辺は trace 列に対応する．円に囲まれた文字は辺の名前である．値は辺の実行確率である．予測の精度は全て.7 である．左向きの辺が次の trace の第一候補を選んだとき，右向きの辺が第 2 候補を選んだときに対応する．

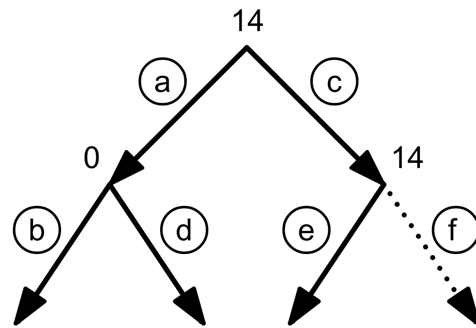

図 3　図 2 と同様であるが，値は next trace prediction における confidence counter の値である．

例を図 3 に示す．今度は各頂点に次の trace の予測の confidence counter の値を付してある．例えば辺 f は，8 より大きい confidence count における第 2 候補を 2 回選んだものになっている．

また，fork する trace の率が 40%程度のとき最も性能が向上することが経験的に分かっているので，fork したら 6 を足し fork しなかったときは 4 引く 8-bit のカウンタ C を用意し，このカウンタの大小で fork を抑制するポリシーを用いる．

これらのことから，以下のポリシーを挙げることができる．

CPR　T の現在の全ての葉に対する第二候補の辺の実行確率を用い，P のこの中での順位が，現在利用可能な thread 数（最大 N）以内なら fork する

CPL　P がある閾値 L を越えるならば fork する

CPLA　根から，問題となっている trace の次の trace の第二候補の辺までの，trace の予測における confidence counter の値の列のうち，ある閾値 H を超えるものの数が，ある閾値 M を越えないならば fork する

EE　常に fork する

MP　全て第一候補を使ってきた辺 (main-path) からのみ fork する

本稿における評価では，$N = 16$，$L = 0.01$，$H = 8$ を用いた．

M は，具体的にはカウンタ C が 64 より小さいときは 3，192 より小さいときは 2，それ以上は 1 とする．この値は経験的に得たものである．

CPLA が我々の提案する方法である．最初の 2 者は実装が困難で，比較のためのものである．ここでは P はその時点までの実行から得られる，予測テーブルのエントリに対する精度から計算する．最後の 2 者は簡単に実装可能で，比較のためのものである．

例えば，図 2 において，EE を使った場合には全ての辺を実行しようとする．MP を用いた際は，辺 c と後続の辺からは fork しないので辺 f は辿らない．CPL で $L = 0.1$ とした際にも辺 f の実行確率がそれより小さいので辿らない．CPR で $N = 3$ とした際，たとえ辺 c の次の trace を選ぶタイミングが辺 a の次の trace を選ぶタイミングより早かったとしても，辺 f は，辺 d,f のうちで 2 番目に確率が高い辺なので辿らない．

また，図 3 において，CPLA で $H = 8, M = 2$ とした際には，辺 c は 8 より大きい confidence count における第 2 候補を一度辿っていて，この先二度以上は辿らないので，辺 f は辿らない．

また，利用可能な thread がないときに精度の低い予測に出会ったときは，保存しておいて後に fork の候補に入れるという方法もあるが，本稿では単純化のためにこれを無視する．

3. 評価方法

3.1 Simulator

これまで述べてきたアーキテクチャを評価するために，execution-driven のシミュレータを用いた．

ISA は MIPS-I [8] である．整数演算命令，浮動小数点演算命令の latency，issue-rate は Alpha 21264 の値に近いものに設定した．実行ユニットの型と数を表 1 に示す．実行ユニットの latency と issue-rate を表 2 に示す．整数演算ユニットは SMT を行うのに十分な数となっている．

表 1　提案するプロセッサの実行ユニットの種類と数

Type	Number
INT	12
INT/LD/ST	4
Other INT	2
FADD	2
FMUL	2
Other Float	2

表 2　提案するプロセッサの実行ユニットの issue-rate と latency

Operation	Issue-rate	Latency
INT	1	1
IMUL	6	6
IDIV	35	35
LD	1	3(min)
FLD	1	4(min)
ST	1	-
FST	1	-
FADD	1	4
FMUL	1	4
FDIV	15	15
FSQRT	33	33

フェッチユニットは連続 2 ブロックを同時にフェッチしようとするが，instruction cache に対するこのアクセスや，data cache に対する 4 つの ld unit のアクセスは実際に bank conflict やそれに伴う retry を考慮して行われる．メモリ階層においても，contention や retry を考慮して行われる．main mamory につながる bus は split-transaction を用いている．

予測された VPC は深さ 64 の VPC キューに格納され，予測された trace-id は深さ 64 の trace-id キューに格納される．フェッチされた instruction のブロックは 128 エントリの instruction キューに格納される．このブロックは instruction cache からフェッチされた場合は 8-instruction，trace cache からフェッチされた場合は 16-instruction である．その後 8-instruction の幅をもつ decode stage に入る．その後 instruction は実行ユニットごとの深さ 16 の reservation station と深さ 256 の active list [11] に入る．サイクルあたりの retire 数は 16-insturction である．load キュー及び store キューは 64 エントリである．

メモリ階層の configuration を表 3 に示す．data TLB 及び instruction TLB は 256-block，4-way，である．TLB の miss-penality は 50，page fault の penalty は 500 である．

3.2 Benchmark Programs

ベンチマークプログラムとして SPECINT95 の compress，go，ijpeg，li を用いた．データセット及びプログラムの特性を表 4 に示す．これらのプログラムを選択したのは分岐予測ミス率がそれぞれ 6.59%,

(注1)：compress では，圧縮，展開の繰り返しは 1 回に変更され，確率テーブルや圧縮対象文字列の準備のフェーズは計測に含まれないように変更されている．

表 3　提案するプロセッサのメモリ階層のパラメタ

Name	Size (KB)	Assoc.	Banks/ Ports	Issue-rate/ Latency
Trace cache	128	2	8/1	1/2
Instr. cache	32	2	8/2	1/2
Data cache	32	2	8/4	1/2
L2 cache	128	4	8/1	2/4
L3 cache	1024	4	1/1	4/8
Main memory	-	-	16/1	8/16

表 4　評価に使ったベンチマークプログラムとそのデータセット

Benchmark	Input	Num. of instr. retired	Ave. num. of branches in a trace
compress	8000 q 2231 (注1)	$1.28 * 10^6$	1.95
go	5 5	$1.94 * 10^6$	2.52
ijpeg	specmun.ppm	$11.8 * 10^6$	1.77
li	4 queen	$3.33 * 10^6$	2.11

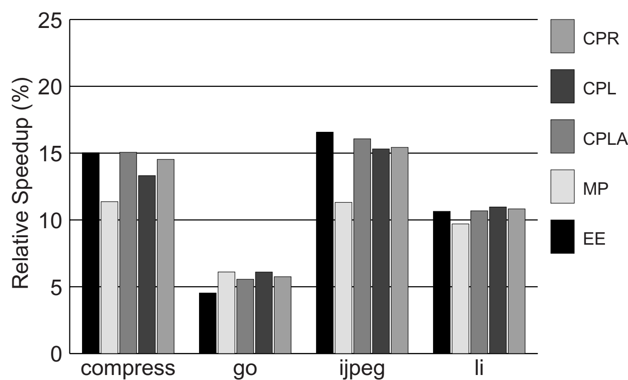

図 4　様々な fork ポリシーの元での複数パス実行における，baseline に対する相対速度向上，整数演算ユニットは 16 個

9.13%，8.32%，4.93%と比較的高いためである．プログラムは GCC を用いて”-O2” オプションをつけてコンパイルした．

4. 評価結果

ここで複数パス実行を行わないものを baseline と呼ぶ．

様々なポリシーのもとでの複数パス実行の baseline に対する速度向上を図 4 に示す．またそれぞれの予測ミス率を図 5 に示す．confidence count 15 に属する動的な trace の率を図 6 に示す．また動的な trace の fork 率を図 7 に示す．

CPLA に関して，compress では 15.0%，go では 5.5%，ijpeg では 16.0%，li では 10.6%の速度向上が見られた．分岐予測ミス率の低下はこの速度向上に対応している．

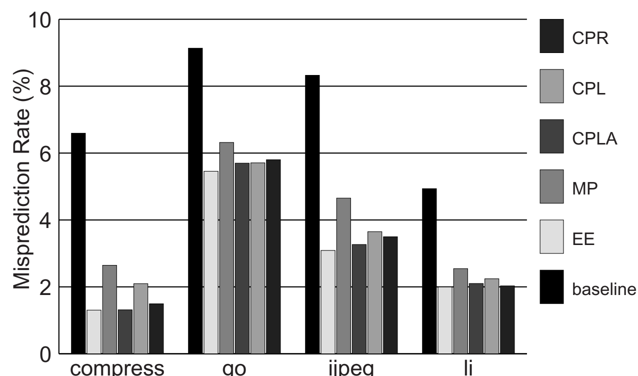

図 5　様々な fork ポリシーの元での複数パス実行における，予測ミス率

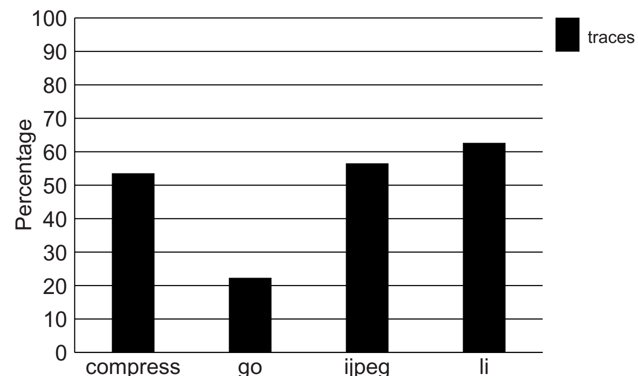

図 6　baseline における confidence count 15 に含まれる動的 trace の率

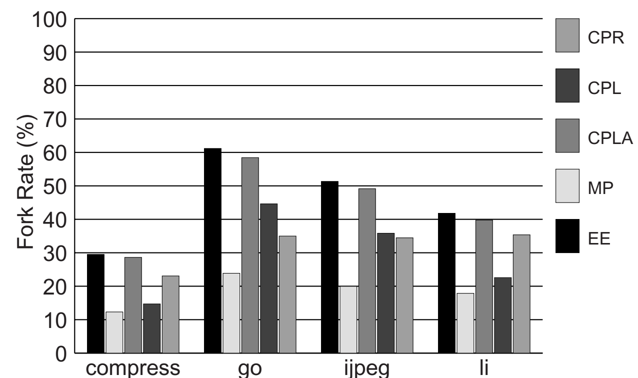

図 7　様々な fork ポリシーの元での複数パス実行における動的 trace の fork 率

まず EE と MP を比較した際に go とそれ以外でプログラムの特性が大きく違う．go では両者とも性能向上は小さい．また MP の方が EE より性能が高い．go 以外では性能向上は go より高く，EE の方が MP より性能が高い．これは以下の理由による．

go では confidence count 15 に入っている trace が，他は 50%を超えている一方で 22.2%と低い．これは次の trace の予測が困難なパスが多いことを示している．これは分岐命令自体が予測困難であることが原因であるが，1 trace に含まれる分岐命令の数の平

均が 2.52 と他より高く，1 trace として予測ミスを起こす可能性が高いことも事態を悪化させている．

これにより，go では fork する率が高く，また fork しても分岐予測を避けられる可能性が低い．fork する率が高いことは trace の fork 率のグラフからも分かる．

これらにより go においては fork 率の増大による正しいパスの IPC の低下の影響が分岐ミスを避ける効果より大きくなり，EE の性能が MP に比べて低くなる．逆にみると MP の方が fork の効率が良く，fork 率を下げるため，EE より性能がよくなる．

go 以外では反対に，confidence の機構により多くの精度の高い予測が fork 候補から除外されていて，fork する率は低く，fork により分岐予測ミスを避けられる可能性は高い．それゆえ自由に fork した方が性能は高く，全体としても性能向上が go より大きくなっている．

次に go において MP と CPLA を比較する．CPLA は MP と同じく効率のよい fork を行うため同様に EE より性能がよくなる．しかし fork 率のグラフを見ても分かるように fork 率を下げる効果は小さいため，MP より性能向上が低い．

次に go 以外において EE と CPLA を比較する．CPLA は fork 率をあまり下げず，効率の良い fork を行って，EE に近い性能を出している．しかし EE より高い性能を出すことはない．EE から CPLA にすることによって分岐予測ミスを避ける効果が低下し，thread 数増大による正しいパスの IPC の低下は緩和されるが，前者の効果が後者の効果を上回るため，性能は低下してしまう．前者の効果が大きいのは，まず先に見たように fork 率が低く，fork により分岐予測ミスを避けられる可能性が高いためである．また分岐予測ミスはクラスタ化する傾向があるため [9] [12] である．これはプログラムの phase の変化に EE は CPLA より対応しやすいからともいえる．

次に CPL と CPLA を比較する．CPL は CPLA よりも fork 率を下げる．これは CPLA では confidence count の高い予測における第 2 候補の制限数が動的に変化するが，CPL では実行確率の制限値は一定であるためである．性能向上は CPL とあまり変わらない．

次に CPR について考察する．CPR は後に起こる可能性のある fork も考慮して fork 候補の fork 可否を決定する．fork しないことにした際，より実行確率の高い候補はある時間がたった後に fork するか否か試される．またその前に squash されてなくなってしまうこともある．それゆえに fork 率が EE より小さくなる．性能向上は CPLA とあまり変わらない．

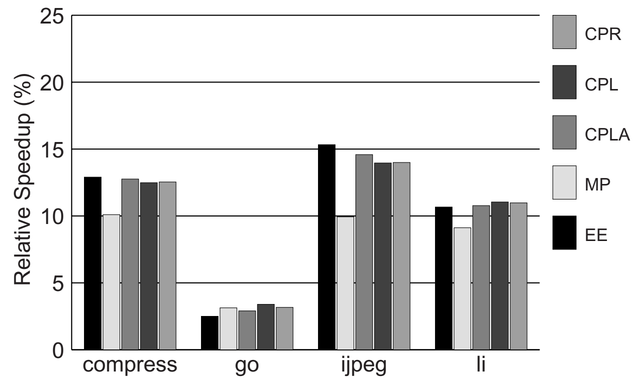

図 8　様々な fork ポリシーの元での複数パス実行における，baseline に対する相対速度向上，整数演算ユニットは 8 個

次に，今までは整数演算ユニットが 16 個であったが，これを 8 個に減らした場合の性能向上のグラフを図 8 に示す．SMT を行うのに十分な数の整数演算ユニットを用意した場合からユニットを削減し，この整数演算ユニットの数にどれだけ性能向上が依存しているかを見るためである．なお，reservation station は演算ユニットごとについているので，演算ユニットを減らすと，reservation station のエントリ数，つまり instruction window の大きさも減る．ここで性能向上は 0〜2%程度低下する．また傾向は 16 個の時と変わらない．

以上により，CPLA は EE や MP 単独よりも go とその他のプログラムに見られるようなプログラムの特性の差異に対応しやすく，また CPL や CPR といった理想的な方法に近い速度向上が単純な実装で得られるといえる．しかし，CPLA は go 以外では EE より速度が向上することはなく，平均的には EE と変わらない．

5. 関 連 研 究

A. K. Uht らは Disjoint Eager Execution (DEE) [4] と呼ぶ方法を提案した．DEE では分岐命令を見つけると複数パスを実行する．ここで複数パス実行によって構成される，分岐命令にはさまれた命令列 (最後の分岐命令を含む) を辺とする木を想定する．この木から有限個の辺を選んで実行するが，そのとき実行確率の和を最大にするように選ぶ．DEE ではパスに関して辿ることを止めたり再開したりする点，我々の方法で第一候補に対応する辺も実行の選択の対象にする点，静的な予測精度を用いる点が我々の方法の CPR と違う．DEE は静的な予測の精度を用

いるため，我々の方法より性能が低くなる．

T. H. Heil らは Selective Dual Path Execution [9] と呼ばれる方法を提案した．これは我々の提案する方法と同じく動的な confidence counter の機構を用いる．この方法では辿るパスは同時に 2 つのみであるが，我々の提案するものは 16 までである．すでに 2 つのパスを辿っているときに出会った精度の低い分岐命令は後に fork 可能になったときに fork 候補になる．我々の提案する方法では単純に無視している．

G. Tyson らは Limited Dual Path Execution [10] を提案した．この方法では分岐予測の confidence は分岐予測機構の内部状態から得ていて，我々の方法のカウンタによる機構と異なる．

S. Wallace らは Threaded Multi-Path Execution [7] と呼ばれる方法を提案した．この方法も SMT を用いる．この方法では先に評価した MP に当たるポリシーを用いている．この場合複数パス実行は分岐予測に基づいており，分岐命令単位で fork が起こる．我々の方法は trace を単位としてフェッチや fork を行う．この方法では T. H. Heil らの方法と同様に fork を一度あきらめた候補も後に候補になる方法を用いている．これによる速度向上は 2〜3%程度であり，全体の速度向上は我々の提案する方法による速度向上に満たない．また，この方法は我々の提案する方法と併用できる．

6. まとめ

深いパイプラインのもと分岐予測ミスペナルティを軽減するために複数パス実行を行うアーキテクチャを提案した．複数パス実行によって構成される，各辺を trace 列とする木に関して，動的な予測精度の近似を用いて，辺の実行確率の近似を計算して，その下限を制限することによりその辺を実行するか否かを選択する方法を提案した．シミュレーションの結果により，複数パス実行を行わない場合に比して，最大 16.0%の速度向上が得られることを示した．また単純なポリシーよりプログラムの特性に対応しやすく，実装が困難な理想的な方法に対して差が小さいことを示した．しかし，go 以外では EE より速度向上が大きくなることはなく，平均的には EE と変わらない．

EE は実装が簡単であるため，fork 選択のポリシーには EE を用い，EE による fork 率増大の負担を予測精度を上げることによって下げたり，実装上の問題を克服するという方向性が考えられる．

文献

[1] E. Rotenberg, S. Bennett, and J. E. Smith : Trace cache: a low latency approach to high bandwidth instruction fetching. *Proc. of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* , pp.24–34 (December 1996).

[2] S. McFarling : Combining Branch Predictors. DEC WRL TN-36 (June 1993).

[3] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm : Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *Proc. of 23rd Annual International Symposium on Computer Architecture* , pp.191–202 (May 1996).

[4] A. K. Uht and V. Sindagi : Disjoint Eager Execution: An Optimal Form of Speculative Execution. *Proc. of 28th Annual International Symposium on Microarchitecture* , pp.313-325 (November 1995).

[5] Q. Jacobson, E. Rotenberg, and J. E. Smith : Path-Based Next Trace Prediction. *Proc. of 30th Annual International Symposium on Microarchitecture* , pp.14–23 (November 1997).

[6] E. Jacobsen, E. Rotenberg, and J. E. Smith : Assigning Confidence to Conditional Branch Predictions. *Proc. of 29th Annual International Symposium on Microarchitecture* , pp.142–152 (December 1996).

[7] S. Wallace, B. Calder, and D. M. Tullsen : Threaded multiple path execution. *Proc. 25th Annual International Symposium on Computer Architecture,* , pp.238–249 (July 1998).

[8] C. Price : MIPS IV Instruction Set, Revision 3.2. MIPS Technologies, Inc., Mountain View, CA, (September 1995).

[9] T. H. Heil and J. E. Smith : Selective dual path execution. University of Wisconsin - Madison, (November 1996).

[10] G. Tyson, K. Lick, and M. Farrens : Limited dual path execution, Technical Report CSE-TR 346-97, University of Michigan, (1997).

[11] K. C. Yeager : The MIPS R1000 Superscalar Microprocessor. IEEE Micro, 16(2), pp.28–40 (April 1996).

[12] M. Farrens, T. Heil, J. E. Smith, G. Tyson : Restricted Dual Path Execution. Technical Report CSE-97-18, University of California, (November 1997).

[13] T.-Y. Yeh and Y. N. Patt : Alternative Implementation of Two-Level Adaptive Branch Prediction. *Proc. 19th Annual International Symposium on Computer Architecture* , pp.124–134 (May 1992).

[14] Q. Jacobson, S. Bennett, N. Sharms and J. E. Smith : Control Flow Speculation in Multiscalar Processors. *Proc. 3rd International Symposium on High-Performance Computer Architecture* , pp.218–229 (February 1997).