# 高レベル投機技術を用いた
# 複数パス実行プロセッサ

吉 瀬 謙 二

# 概要

マイクロプロセッサは 1971 年に発表された 4004 の誕生以来，デバイス技術とアーキテクチャの進歩により発展を続けてきた．デバイス技術に関しては 3 年間でチップに集積するトランジスタ数が 4 倍になるというペースが続いており，これらのハードウェア資源を利用して VLIW，スーパーパイプライン，スーパースカラなどの方式を利用したプロセッサが開発されている．

プロセッサ市場に視点を移すと，スーパースカラが高性能プロセッサ・アーキテクチャの主流となっている．スーパースカラ・プロセッサは，従来のスカラプロセッサとのコード互換性を維持しながら，分岐予測を用いた投機実行やアウトオブオーダ実行といったアーキテクチャの改良により，命令レベルの並列性を抽出し，サイクル当たりの処理命令数を増加させることで性能を向上させている．また，2 次キャッシュがチップ上に実装されるようになってきており，プロセッサの処理能力に対するデータ供給システムの処理能力不足を緩和しており，この点もプロセッサの性能向上に寄与している．

このように，順調に性能を向上させてきたスーパースカラ・プロセッサだが，次の問題点が，より高い命令レベル並列性の利用を制限する．

- 分岐予測に成功すれば制御依存関係による性能低下を排除できるが，予測ミスが発生した場合には正しい制御流の再投入が必要となり，投機的に処理した命令を破棄しなければならない．分岐予測の精度は 95%程度に向上しているが，残りの約 5%の分岐予測ミスによりプロセッサ内の命令列が分断され，並列性を抽出するためのウィンドウを拡大することによる十分な利得が得られない．

- ロード命令の実行を開始するためには，先行してフェッチされたストア命令のメモリ参照アドレスが計算されている必要があり，データ依存関係を持たないストア命令であっても，そのメモリ参照アドレスの計算が終了するまで，ロード命令の実行を遅らせる必要がある．この制約を曖昧なメモリ依存関係と呼び，並列性の抽出を制限する要因となっている．

前者の分岐予測ミスを削減する投機技術として，分岐成立と不成立の両方のパスにおいて投機的に処理を進める複数パス実行が提案されている．また，後者の曖昧なメモリ依存関係の解消を目指す投機技術として，ストアセットを用いたメモリ依存予測が提案されている．これらの制約を取り除いた場合には，真のデータ依存関係が命令レベル並列性を制限する大きな要因となる．従来，真のデータ依存関係を解消することは困難と考えられてい

たが，近年，真のデータ依存関係を解消する技術として，値予測を用いた投機処理の手法が提案された．これは，実際に計算をおこなってデータ値を得る代わりに，生成されるデータの値を予測することで処理を進めておくという投機技術である．

チップに集積するトランジスタ数の急激な増加が追い風となり，これらの投機技術の利用は性能向上のための現実的な選択肢となっている．本研究では，既存の RISC プロセッサとのコード互換性を維持しながら，命令レベル並列性 10 をもたらす投機技術の確立を目的とする．この高い命令レベル並列性を達成するために，制御依存関係を解消する複数パス実行，曖昧なメモリ依存関係を解消するメモリ依存予測，真のデータ依存関係を解消する値予測の各手法を議論した．本研究で提案する複数パス実行，メモリ依存予測，値予測の各投機技術を，高精度かつ積極的という意味で高レベル投機技術と呼ぶことにする．本研究では，それぞれの高レベル投機技術の検討に加え、高レベル投機技術の融合手法を検討し，以下の知見を得た．

一つ目の投機技術として複数パス実行を議論した．従来からおこなわれている単一パスの実行では，分岐結果を予測し，予測された単一のパスにおける命令列を投機的に実行する．一方，全ての分岐命令において分岐成立と分岐不成立の両方のパスを実行する Eager Execution では，多くの分岐命令を跨いでパスを展開するに従って，必要となるハードウェア資源が爆発的に増加する．このため，現実的なハードウェア量を考慮した場合には，パスを複製する分岐命令を選択するパス割付け戦略が複数パス実行の性能を決める重要な要素となる．提案されている幾つかのパス割付け戦略を比較検討した上で，各手法の利点を組み合わせる JRS-PM 方式を提案した．複数パス実行プロセッサのシミュレータを用いて，命令レベル並列性，各パイプライン・ステージで処理される命令トラフィック，データキャッシュのヒット率を測定した．評価結果より，利用できるパスの数を 16 に設定することで 59%の分岐ミスペナルティを削減できることが明らかになった．この時，サイクル当たりの平均フェッチ命令数は 43，平均実行命令数は 21 というデータを得た．これらの評価により複数パス実行の可能性が一部確認された．

二つ目の投機技術として，曖昧なメモリ依存関係を解消するメモリ依存予測を議論した．まず，曖昧なメモリ依存関係がプロセッサ性能に与える影響を測定し，曖昧なメモリ依存関係を解消することで 13%から 18%の性能向上を達成できることを示した．これまでに提案されているストアセットを用いたメモリ依存予測は，実行中のストア命令にデータ依存関係を持たないロード命令を予測する．しかし，32%のロード命令は実行中のストア命令にデータ依存関係を持ち，ストアセットだけでは曖昧なメモリ依存関係の 68%しか解消することはできない．このため，ロード命令が実行中のストア命令にデータ依存関係を持つ

場合に，複数のストア命令から，データ依存関係を持つストア命令を特定するストア特定予測を提案した．シミュレーションによる評価により，ストアセットを用いた予測手法と，ストア特定予測を同時に利用することで 85%の曖昧なメモリ依存関係を解消できることが判明した．

三つ目の投機技術として，真のデータ依存関係を解消する値予測を議論した．これまでの研究では，予測する命令の前回の演算結果を今回の予測値として利用する Last-value 予測や，過去の 2 回の演算結果の差分と前回の演算結果の和を予測値とするストライド値予測などが検討されてきたが，これらの予測機構には予測ミスの回数が多いという欠点がある．この欠点の解消を目指して，2 レベル・ストライド値予測を提案した．2 レベル・ストライド値予測では，ストライド値予測でミスした際のグローバルな条件分岐命令の履歴を保存し，同様のミスの発生を抑えることで，予測ミスの削減と正しく予測できる割合の向上を目指す．シミュレーションにより提案手法を評価し，予測を有効とする確信度評価の閾値を 8 に設定した場合に，最も高い 14%の性能向上を確認した．この時，全実行命令の 34%の命令の演算結果を正しく予測できることが判明した．

提案した 3 つの高レベル投機技術は，制御依存関係，曖昧なメモリ依存関係，真のデータ依存関係，という異なる依存関係を解消する．これらの高レベル投機技術は，それぞれの単体利用によりプロセッサの性能向上を達成できるが，複数の高レベル投機技術を同時に利用することで，更なる性能向上が可能となる．本研究では，これらの高レベル投機技術の融合手法を提案し，複数パス実行プロセッサのシミュレータを用いて，その性能を評価した．その結果，複数パス実行，メモリ依存予測，値予測を単体で利用した場合には，それぞれ 23%, 15%, 10%の性能向上率しか得られないが，3 つの高レベル投機技術の融合により 50%という高い性能向上率を達成できることが明らかになった．

# 目 次

# 図 目 次

# 表 目 次

# 第 1 章

# 序論

## 1.1 本研究の背景

1971 年に発表された 4004 の誕生以来，マイクロプロセッサは急激な性能向上を続けてきた．しかしながら，一方では，科学技術演算に加えて，動画や音声の圧縮・解凍といった大規模データに対する処理が身近なものとなり，より高い計算パワーが必要とされている現状がある．このような中，計算機システムの中枢を担うマイクロプロセッサの高速化に対する要求はますます大きくなっている．

マイクロプロセッサの性能向上は，主に，デバイス技術とアーキテクチャの進歩に支えられている．デバイス技術に関しては 3 年間でチップに集積するトランジスタ数が 4 倍になるというペースが続いており，これら増加を続けるハードウェア資源を利用して VLIW，スーパーパイプライン，スーパースカラなどのアーキテクチャを採用するプロセッサが開発されている．これらのアーキテクチャでは動作周波数の向上と，サイクル当たりに完了する命令数を向上させることで性能を向上させている．米国半導体工業会によれば，今後もほぼ同様のペースでデバイス技術が進歩すると予測されており，これら豊富なハードウェア資源を効率良く利用して，高い性能を達成するプロセッサ・アーキテクチャの開発が求められている．

現在のプロセッサ市場に視点を移すと，スーパースカラが高性能プロセッサ・アーキテクチャの主流となっている．スーパースカラは，従来のスカラプロセッサとのコード互換性を維持しながら，分岐予測を用いた投機実行やアウトオブオーダ実行といったアーキテクチャの改良により，より高い命令レベルの並列性を抽出することで性能を向上させている．また，2 次キャッシュがチップ上に実装されるようになってきており，プロセッサの処

理能力に対するデータ供給システムの処理能力不足を緩和しており，この点もプロセッサの性能向上に寄与している．

このように，順調に性能を向上させてきたスーパースカラ・プロセッサだが，次の問題点が，より高い命令レベル並列性の利用を制限する．

- 分岐予測に成功すれば制御依存関係による性能低下を排除できるが，予測ミスが発生した場合には正しい制御流の再投入が必要となり，投機的に処理した命令を破棄しなければならない．分岐予測の精度は95%程度に向上しているが，残りの約5%の分岐予測ミスによりプロセッサ内の命令列が分断され，並列性を抽出するためのウィンドウを拡大することによる十分な利得が得られない．

- ロード命令の実行を開始するためには，先行してフェッチされたストア命令のメモリ参照アドレスが計算されている必要があり，データ依存関係を持たないストア命令であっても，そのメモリ参照アドレスの計算が終了するまで，ロード命令の実行を遅らせる必要がある．この制約を曖昧なメモリ依存関係と呼び，並列性の抽出を制限する要因となっている．

スーパースカラ・プロセッサは，そのウェイ数を2から4に引き上げ、リオーダバッファのエントリ数を60程度に増加させることで，利用できる命令レベル並列性を向上させてきた．しかし，上に挙げた制約により，これらのパラメタの拡大による性能向上には限界が見え始めている．

制御依存関係や曖昧なメモリ依存関係といった制約を積極的に解消し，より高い命令レベル並列性の利用を可能とする新しいアーキテクチャが必要とされている．

## 1.2 本研究の目的

プロセッサ内で守らなければならない制御依存関係やデータ依存関係といった制約をこれ以上解消できないとすると，現在の高性能プロセッサが利用している以上の命令レベル並列性の抽出は困難となる．しかしながら，これらの制約は投機技術を用いることで緩和できることが近年の研究で明らかになっている．

例えば、1番目の命令が生成した演算結果を2番目の命令が入力オペランドとして利用するという状況を考える．真のデータ依存関係を満たす限りにおいては，これら2つの命令を同時に処理することはできない．しかしながら，1番目の演算結果をなんらかの手法を用いて予測できるとすれば，2番目の命令は予測された演算結果を用いて，1番目の命令

と同時に実行することができる．このように，予測を用いて制約を解消し，予測が正しいという仮定のもとで処理を続けることを**投機実行**と呼び，投機実行のためのさまざまな技術（**投機技術**）が検討されている．

先に，スーパースカラにおける問題点の一つ目として挙げた制御依存関係の解消を目指す投機技術として，分岐成立と不成立の両方のパスにおいて投機的に処理を進める複数パス実行が提案され，その可能性が議論されている．また，二つ目に挙げた曖昧なメモリ依存関係の解消を目指す投機技術として，メモリ依存予測が提案されている．これらの制約を取り除いた場合には，真のデータ依存関係が命令レベル並列性を制限する大きな要因となる．従来，真のデータ依存関係を解消することは困難と考えられていたが，近年，真のデータ依存関係を解消する手法として，値予測を用いた投機処理の手法が提案された．これは，実際に計算をおこなってデータ値を得る代わりに，生成されるデータの値を予測することで処理を進めておくという投機技術である．

チップに集積するトランジスタ数の急激な増加が追い風となり，これらの投機技術の利用は性能向上のための現実的な選択肢となっている．本研究では，既存の RISC プロセッサとのコード互換性を維持しながら，命令レベル並列性 10 をもたらす投機技術の確立を目的とする．この高い命令レベル並列性を達成するために，制御依存関係を解消する**複数パス実行**，曖昧なメモリ依存関係を解消する**メモリ依存予測**，真のデータ依存関係を解消する**値予測**の各手法を議論する．本研究では，複数パス実行，メモリ依存予測，値予測の各投機技術を高精度かつ積極的という意味で高レベル投機技術と呼ぶこととし，これら**高レベル投機技術の融合手法**を検討する．

## 1.3 本論文の構成

本論文は，本章を含めて 9 章で構成される．

まず第 2 章において，高い並列性を目指すプロセッサ・アーキテクチャと，その問題点，次世代プロセッサ・アーキテクチャに対する要望を整理する．

第 3 章では，本研究の評価に利用するベンチマークプログラムを定義する．

第 4 章では，高レベル投機技術による性能向上を明確にするための比較対象として利用するベースラインプロセッサを定義し，その性能を明確にする．

第 5 章から，第 7 章において依存関係の積極的な削減を目指す高レベル投機技術を提案し，その性能を評価する．第 5 章では，分岐予測ミスを削減する複数パス実行を議論する．第 6 章では，曖昧なメモリ依存関係を解消するメモリ依存予測を議論する．第 7 章では，

真のデータ依存関係を解消する値予測を議論する．

第8章では，複数パス実行，メモリ依存予測，値予測という3つの高レベル投機技術の融合手法を議論する．

最後に第9章で，本論文を総括する．

# 第 2 章

# プロセッサの動向

マイクロプロセッサは 1971 年に発表された 4004[FJMs96] の誕生以来，主に，デバイス技術とアーキテクチャの進歩により発展を続けてきた．デバイス技術に関しては 3 年間でチップに集積するトランジスタ数が 4 倍になるというペースが続いている．これらの豊富なハードウェア資源を利用して VLIW，スーパーパイプライン [PdWW89]，スーパースカラ [マイ 94] などのプロセッサ・アーキテクチャが提案され，実用化されている．これらのアーキテクチャは，サイクル当たりに完了する命令数の向上，言い換えれば，高い命令レベル並列性 (**Instruction Level Parallelism: ILP**) を抽出することで性能を向上させている．

本章では，命令レベル並列性の利用を目指すプロセッサ・アーキテクチャと，その問題点，次世代プロセッサ・アーキテクチャに対する要求を整理する．

## 2.1 スーパースカラ・アーキテクチャ

本節では，現在の高性能アーキテクチャの主流となっているスーパースカラと，そこで利用されている要素技術を議論する．説明を簡潔にするために，命令の実行を以下の 4 つのステージに分割して考える．

**命令フェッチ (IF)** 命令キャッシュから命令をフェッチする．

**命令デコード (ID)** 命令をデコードする．

**オペランドフェッチ (OF)** レジスタファイルあるいはメモリから，オペランドをフェッチする．

**実行 (EX)** 入力オペランドに対し演算をおこなう．また，演算結果をレジスタファイルまたはメモリに格納する．

上に述べた命令フェッチ，命令デコード，オペランドフェッチ，実行というそれぞれの処理に等しい時間を必要とし，かつ，それぞれの処理がプロセッサの異なるハードウェアを利用する場合には，命令間の処理をオーバーラップさせることでスループットを向上させることができる．この手法をパイプライン処理と呼ぶ．

パイプライン処理をおこなうプロセッサと，非パイプライン処理のプロセッサの比較を図 2.1に示す．図 2.1では，それぞれの命令の実行ステージに色をつけた．実行ステージのみで利用されるハードウェアの使用率を見てみると，非パイプライン処理では 25%だった使用率を，パイプライン処理により向上できることがわかる．

ただし，パイプライン処理は命令内に存在する並列性を利用して，サイクル当たりに処理される命令数を 1 に近づける技術である．パイプライン処理は，命令レベルの並列性を利用している訳ではない．

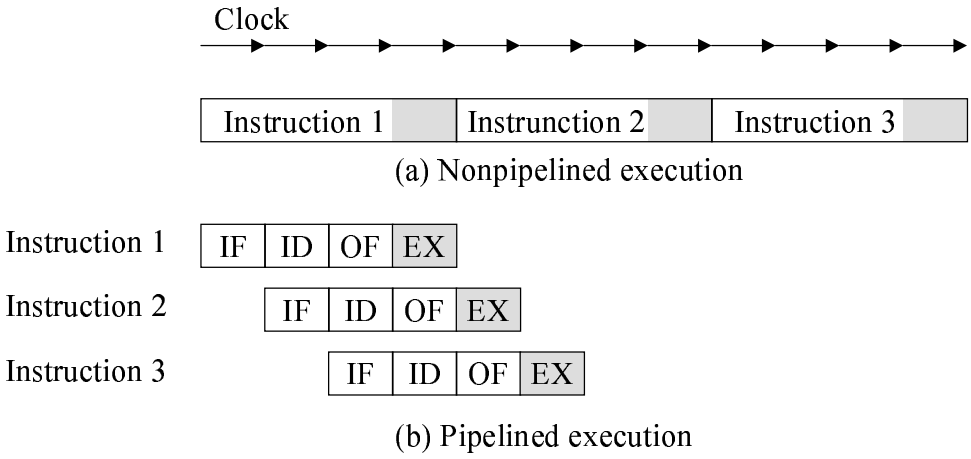



図 2.1: 非パイプライン処理とパイプライン処理の比較

命令を処理するパイプラインを 2 本以上に拡張して，依存関係のない命令を並列に実行するアーキテクチャをスーパースカラ [マイ 94] と呼ぶ．スーパースカラという用語は，パイプラインを 1 本しか持たないことを意味するスカラという用語と区別するために利用される．

もし，全ての命令間に制御依存関係やデータ依存関係といった依存関係が存在しないとすれば，命令の並列実行が可能となる．このためには，命令を処理するためのパイプライ

ンを複数用意すれば良い．スカラプロセッサと，2本のパイプラインを利用するスーパースカラとの比較を図2.2に示す．図2.2では全ての命令間に制御依存関係やデータ依存関係といった依存関係が存在しなことを仮定している．図左のスカラ・プロセッサでは4サイクル目から毎サイクル1命令が完了する．これに対して，図右のスーパースカラ・プロセッサでは4サイクル目から毎サイクル2命令が完了する．

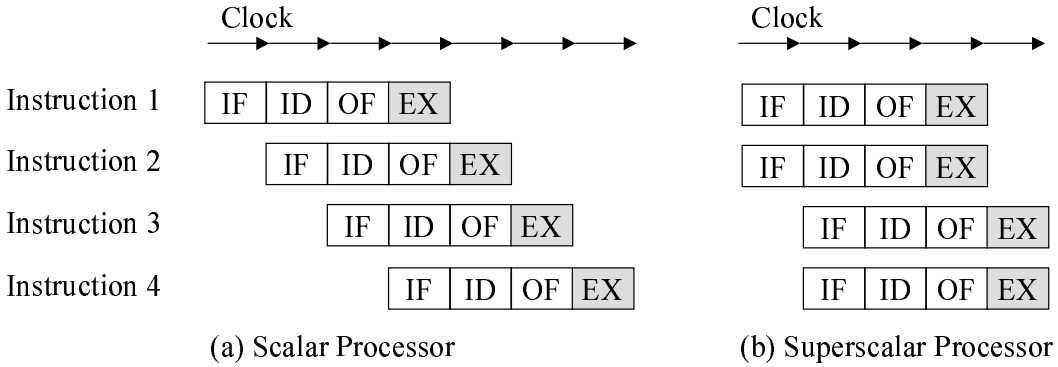



図 2.2: スカラプロセッサとスーパースカラプロセッサの比較

パイプラインの本数を増やすことで，ピークの命令レベル並列性を向上させることができる．しかしながら，実際には，処理される命令間に存在する様々な依存関係により，利用できる命令レベル並列性が制限される．

### レジスタリネーミング

命令レベル並列性を制限する制約の一つにデータ依存関係がある．レジスタを介したデータ依存関係は次の3種類に分類できる [HP95]．

- **真のデータ依存 (true data dependence)**: 先行命令 $i$ があるレジスタに値を書き込む前に，後続命令 $j$ が当該レジスタを読み出そうとする際に発生する制約
- **逆依存 (antidependence)**: 先行命令 $i$ があるレジスタから値を読み出す前に，後続命令 $j$ が当該レジスタに書き込もうとする際に発生する制約
- **出力依存 (output dependence)**: 先行命令 $i$ があるレジスタに値を書き込む前に，後続命令 $j$ が当該レジスタに書き込もうとする際に発生する制約

これらのデータ依存関係の中で逆依存と出力依存は記憶場所の再利用が原因で発生する．このため，プロセッサに豊富な数の物理レジスタを実装し，動的に，論理レジスタ番号を物理レジスタ番号に変換することで逆依存と出力依存を解消できる．この手法をレジスタリネーミングと呼ぶ．

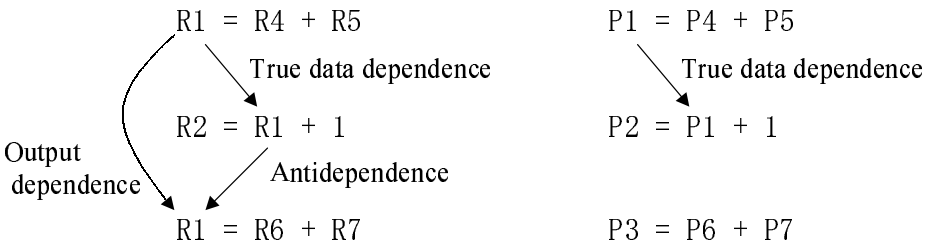



図 2.3: レジスタリネーミングによる逆依存と出力依存の解消

レジスタリネーミングにより，逆依存と出力依存を解消する例を図 2.3に示す．図の左はレジスタリネーミングをおこなう前の命令列で，R1 から R7 は論理レジスタ番号を表している．この例では，2 番目の命令で R1 の利用が終了し，3 番目の命令の出力レジスタに R1 を再利用している．この R1 の再利用が原因となり出力依存と逆依存が発生している．図 2.3右には，レジスタリネーミングをおこなった後の依存関係を示している．P1 から P7 は物理レジスタ番号を表している．変換前には 1 番目と 3 番目の命令において同じ論理レジスタ番号 R1 が利用されていたが，それぞれを P1, P3 という異なる物理レジスタに割り当てることで，逆依存と出力依存を解消している．

### アウトオブオーダ実行

より高い命令レベル並列性を抽出するためには，フェッチされた順番とは関係なく，オペランドが揃った命令から実行を開始できるような仕組み（アウトオブオーダ実行）を導入すればよい．一方，フェッチされた順番で，命令の実行を開始するプロセッサインオーダ実行プロセッサと呼ぶ．

アウトオブオーダ実行を実現するためには，プロセッサをフェッチした順番で進行する部分と，フェッチ順と関係なく処理が進行する部分に分割する．前者をプロセッサのフロントエンド，後者をプロセッサのバックエンドと呼ぶことにする．また，フロントエンドとバックエンドの間に，実行の開始を待つ命令を蓄えておくバッファを配置する．このバッ

ファを命令ウィンドウと呼ぶ．命令ウィンドウは，それぞれの機能ユニット毎に分散されて配置される場合にはリザベーション・ステーションと呼ばれることがある．

アウトオブオーダ実行を採用した場合においても，命令のリタイアはインオーダに処理される必要がある．このために，利用されるバッファをリオーダ・バッファと呼ぶ．

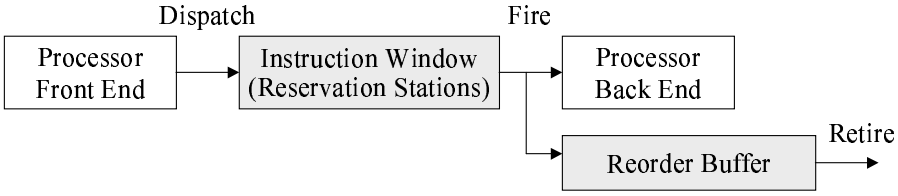



図 2.4: アウトオブオーダ実行プロセッサ

アウトオブオーダ実行を支える命令ウィンドウ，リオーダバッファとプロセッサの関係を図 2.4に示す[1]．フロントエンドから，命令ウィンドウに命令を格納する動作をディスパッチ (**dispatch**)，オペランドが揃い命令実行が開始されることを発火 (**fire**)，リオーダバッファからエントリが開放され命令の処理が完了することをリタイア (**retire**) と呼ぶことにする．

## 2.2 スーパースカラの限界

スーパースカラ・プロセッサは，レジスタ・リネーミングやアウトオブオーダ実行といった技術を用いて性能を向上させてきた．しかしながら，次の問題点が，より高い命令レベル並列性の利用を制限している．

- 分岐予測に成功すれば制御依存関係による性能低下を排除できるが，予測ミスが発生した場合には正しい制御流の再投入が必要となり，投機的に処理した命令を破棄しなければならない．分岐予測の精度は 95 %程度に向上しているが，残りの約 5 %の分岐予測ミスによりプロセッサ内の命令列が分断され，並列性を抽出するためのウィンドウを拡大することによる十分な利得が得られない．

[1]インオーダ実行プロセッサに対する拡張という意味を明確にするために，図 2.4では，フロントエンドとバックエンドとは別のユニットとして命令ウィンドウとリオーダバッファを描いた．実際には，これらのバッファはバックエンドの一部である．

- ロード命令の実行を開始するためには，先行してフェッチされたストア命令のメモリ参照アドレスが計算されている必要があり，データ依存関係を持たないストア命令であっても，そのメモリ参照アドレスの計算が終了するまで，ロード命令の実行を遅らせる必要がある．この制約を曖昧なメモリ依存関係と呼び，並列性の抽出を制限する要因となっている．

スーパースカラ・プロセッサは，そのウェイ数を 2 から 4 に引き上げ、リオーダバッファのエントリ数を 60 程度に増加させることで，利用できる命令レベル並列性を向上させてきた．しかし，上に挙げた制約により，これらのパラメタの拡大による性能向上には限界が見え始めている．

## 2.3 半導体技術予測

米国半導体工業会 (Semiconductor Industry Association: SIA) のロードマップ [Ass99] から，ハイパフォーマンス・プロセッサにおけるプロセス技術，集積できるトランジスタ数，チップサイズに関する技術予測を表 2.1にまとめる．

| **Year** | **1999** | **2001** | **2003** | **2005** |
|---|---|---|---|---|
| Process Technology ($nm$) | 180 | 180 | 130 | 100 |
| Functions per chip($milliontransistors$) | 110 | 220 | 441 | 882 |
| Chip size ($mm^2$) | 450 | 450 | 567 | 622 |

表 2.1: Technology Roadmap for Semiconductors

これらの予測の中でも，特に，トランジスタ数の急激な増加には目を見張るものがある．トランジスタ数は 2 年間で 2 倍というベースの増加が予測されている．

これら，急激なペースで増加する大規模なハードウェア資源の有効活用が，次世代プロセッサ・アーキテクチャにおける重要な課題となる．

## 2.4 投機技術

プロセッサ内で守らなければならない制御依存関係やデータ依存関係といった制約を，これ以上解消できないとすると，更なる命令レベル並列性の抽出は困難となる．しかしなが

ら，これらの制約は投機技術を用いることで緩和できることが近年の研究で明らかになっている．

例えば，図 2.3に示した真のデータ依存関係を例にとると，2番目の命令は1番目の命令の演算結果 (R1) を利用するために，真のデータ依存関係を満たす限りにおいて，これら2つの命令を同時に処理することはできない．しかしながら，1番目の演算結果をなんらかの手法を用いて予測できるとすれば，2番目の命令は予測された演算結果を用いて，1番目の命令と同時に実行することができる．

このように，予測を用いて制約を解消し，予測が正しいという仮定のもとで処理を続けることを**投機実行**と呼び，投機実行のための技術を**投機技術**と呼ぶ．

投機実行は，それ自体が新しいアイデアというわけではない．商用プロセッサにおいても，従来から，分岐予測を用いた投機実行が採用されているし，近年のプロセッサにおいても Alpha21264[Kes99] では曖昧なメモリ依存関係を解消するための投機技術が利用されている．

## 2.5 次世代プロセッサ・アーキテクチャ

制御依存関係，データ依存関係，曖昧なメモリ依存関係といった制約から，ウェイ数，命令ウィンドウのエントリ数，機能ユニット数の拡大といったスーパースカラの拡張による性能向上には限界が見え始めている．このような中，より高い命令レベル並列性の利用を目指した新しいアプローチを持つ次世代プロセッサ・アーキテクチャが議論されている．

### 2.5.1 オンチップ・マルチプロセッサ

従来のマルチスレッド・アーキテクチャの流れを汲むオンチップ・マルチプロセッサ（または，マルチプロセッサ・オン・チップ，チップ・マルチプロセッサ）は，1チップ内に複数の汎用プロセッサを集積する．

オンチップ・マルチプロセッサに関しては，Multiscalar[SET95], Hydra[OAH+96], SKY[小林 98], MUSCAT[鳥居 97] といった多くの研究プロジェクトが存在し，その実現を目指して研究が進められている．また，オンチップ・マルチプロセッサの性能は文献 [坂井 97] などで議論されている．

### 2.5.2 投機アーキテクチャ

投機技術を積極的に利用して，データ依存関係，制御依存関係，曖昧なメモリ依存関係といった制約を解消し，より高い命令レベル並列性の抽出を狙うアーキテクチャを**投機アーキテクチャ**と呼ぶことにする．

投機アーキテクチャは，シングルスレッドのプログラムを高速に実行するアーキテクチャであり，従来の RISC コードとの互換性を維持することが可能である．

大規模データパスアーキテクチャ[中村 97b]，High-Performance Substrate[PEFS97]，Superflow[LS97b] といったアーキテクチャが投機アーキテクチャに分類される．

本研究で議論する**高レベル投機技術を用いた複数パス実行プロセッサ**は，分岐予測ミスを削減する**複数パス実行**，曖昧なメモリ依存関係を解消するメモリ**依存予測**，真のデータ依存関係を解消する**値予測**の 3 つの投機技術を融合する**投機アーキテクチャ**のプロセッサである．

## 2.6 本章のまとめ

命令レベル並列性を抽出するために，プロセッサ・アーキテクチャに改良が加えられてきた．パイプライン処理は命令処理をオーバーラップさせることでスループットを向上させる技術であり，サイクル当たりに完了する命令数を 1 に近づける．スーパースカラにより複数命令の並列実行が可能となり，サイクル当たりに 1 を超える命令の完了が可能となる．より高い命令レベル並列性の抽出を目指し，アウトオブオーダ実行とレジスタリネーミングが採用され，現在の高性能プロセッサでは 2 から 3 の命令レベル並列性が利用されている．

現在の主な高性能プロセッサは 4 ウェイのスーパースカラを採用しているが，これ以上の並列性の抽出は，ハードウェア規模を大きくするだけでは困難である．なぜなら，分岐予測ミス，真のデータ依存関係，曖昧なメモリ依存関係の存在が並列性の抽出を妨げているからである．

このような中，より高い命令レベル並列性の利用を目指した次世代アーキテクチャとして，従来のマルチスレッド・アーキテクチャの流れを汲むマルチプロセッサ・オンチップや，投機技術を用いて依存関係の解消を目指す投機アーキテクチャが検討されている．

本研究で議論する高レベル投機技術を用いた複数パス実行プロセッサは，分岐予測ミスを削減する複数パス実行，曖昧なメモリ依存関係を解消するメモリ依存予測，真のデータ

依存関係を解消する値予測の 3 つの投機技術を融合する投機アーキテクチャのプロセッサである.

# 第 3 章

# ベンチマークプログラム

SPECint95 の 8 本のプログラムを用いて高レベル投機技術の性能を議論する．浮動小数点演算系の SPECfp ではなく，整数演算系のプログラムを選択した理由は，整数演算系のプログラムの挙動がより複雑で並列性の抽出がより困難となるためである．SPECint95 に含まれる 8 本のプログラムの説明を表 3.1にまとめる．

| Benchmark | Descriptiom |
|---|---|
| 099.go | Artificial intelligence; plays the game of “Go” |
| 124.m88ksim | Motorola 88100 chip simulator; runs test program |
| 126.gcc | Based on the GNU C compiler version 2.5.3; builds SPARC code |
| 129.compress | Compresses and decompresses file in memory |
| 130.li | XLISP interpreter |
| 132.ijpeg | Graphic compression and decompression |
| 134.perl | An interpreter for the Perl language |
| 147.vortex | An object oriented database |

表 3.1: SPECint95 Benchmark Programs

本研究では，Alpha AXP アーキテクチャ [高澤 93] をターゲットとしてコンパイルされたコードを用いて評価をおこなう．Alpha AXP アーキテクチャは標準的な 64 ビットの RISC アーキテクチャであり，遅延分岐を採用していないために複数パス実行をおこなうプロセッサとの相性がよい．表 3.1にまとめたプログラムのソースは C 言語で記述されており，これらのプログラムは DEC C コンパイラ，最適化オプション -O4 を用いてコンパイルする．

SPEC95 の標準的な実行を用いた評価は，解析に多くの時間を必要とする．そこで，それぞれのプログラムの実行命令数が 1 億から 2 億命令程度に収まる様に入力ファイルと実

行パラメタを変更した．利用した入力パラメタを以下に示す．

**099.go** 9 9

**124.m88ksim** train/ctl.in

**126.gcc** -quiet -funroll-loops -fforce-mem -fcse-follow-jumps -fcse-skip-blocks
-fexpensive-optimizations -fstrength-reduce -fpeephole -fschedule-insns
-finline-functions -fschedule-insns2 -O genrecog.i -o genrecog.s

**129.compress** 30000 q 2131

**130.li** train.lsp

**132.ijpeg** -image_file specmun.ppm -compression.quality 50
-compression.optimize_coding 0 -compression.smoothing_factor 50
-difference.image 1 -difference.x_stride 10 -difference.y_stride 10
-verbose 1 -GO.findoptcomp

**134.perl** scrabbl.pl scrabbl.in (“admits” in 1/5 input)

**147.vortex** train/vortex.in (PART_COUNT 10)

| Program | 実行命令数 | 整数演算 | 分岐命令 | ロード | ストア | 浮動小数点演算 |
|---|---|---|---|---|---|---|
| go | 138,967,546 | 51.6% | 13.5% | 27.5% | 7.2% | 0.0% |
| m88ksim | 127,469,036 | 54.5% | 12.2% | 23.7% | 9.4% | 0.0% |
| gcc | 151,954,906 | 46.0% | 17.4% | 24.7% | 11.7% | 0.0% |
| compress | 142,040,343 | 55.2% | 11.2% | 22.4% | 9.7% | 0.0% |
| li | 208,167,377 | 39.6% | 19.1% | 26.2% | 15.0% | 0.0% |
| ijpeg | 172,380,978 | 66.8% | 6.3% | 21.4% | 5.3% | 0.0% |
| perl | 153,961,901 | 43.3% | 16.2% | 27.2% | 13.0% | 0.0% |
| vortex | 185,174,474 | 42.6% | 14.0% | 28.5% | 14.7% | 0.0% |
| 平均 | 160,014,570 | 50.0% | 13.7% | 25.2% | 10.8% | 0.0% |

表 3.2: ベンチマークプログラムの実行命令数

これらの入力パラメタを用いた場合の実行命令数，全実行命令に対する整数演算命令，分岐命令，ロード命令，ストア命令，浮動小数点演算命令の実行割合を表 3.2にまとめる．浮動小数点演算命令の実行回数は数百回と非常に少なく，割合で見た場合には 0.0%となっている．

評価結果をまとめる際には，スペースの都合で表 3.2に示したプログラム名ではなく，compress であれば comp のように，先頭の 4 文字を用いてプログラムを表現することがある．

# 第 4 章

# ベースラインプロセッサ

高レベル投機技術による性能向上を明確にするために，比較対象として用いるベースラインプロセッサを定義する．本章で定義するベースラインプロセッサは，アウトオブオーダ実行をおこなうスーパースカラ・プロセッサである．半導体集積度の向上によるトランジスタ数の増加を想定して，資源競合の幾つかを排除する．このため，利用できる命令レベル並列性で 2 から 3 といわれている現在の商用のプロセッサと比較して，本章で定義するベースラインプロセッサの命令レベル並列性は 5.28 と非常に高い．

## 4.1 ベースラインプロセッサ

ベースラインプロセッサでは，命令フェッチ (IF)，命令デコード (ID)，レジスタリネーミング (RN)，オペランドフェッチ (OF)，実行 (EX)，メモリアクセス (MEM) という 6 段の命令パイプラインを採用する．Alpha21264[Kes99], R10000[Yea96] などでは 7 段の命令パイプラインを採用している．PowerPC 620[LTT95] などでは 5 段の命令パイプラインを採用している．CISC 系のプロセッサでは 10 段を超える長い命令パイプラインを採用しているものがあるが，RISCプロセッサとして考えた場合には，ベースラインプロセッサで採用する 6 段の命令パイプラインは標準的な設定といえる．

図 4.1にベースラインプロセッサのブロック図を示す．ベースラインプロセッサはアウトオブオーダ実行をおこなう標準的なプロセッサ構成 [マイ 94] を持つ．プロセッサの動作を命令パイプラインのステージに従って説明する．(1) フェッチ・ユニットは命令キャッシュから命令をフェッチする．(2) デコード・ユニットは命令をデコードする．(3) リネーム・ユニットは論理レジスタを物理レジスタに変換し，レジスタに関する出力依存関係と逆依存関係を解消する．(4) オペランドフェッチ・ユニットはレジスタファイルまたはリオーダ

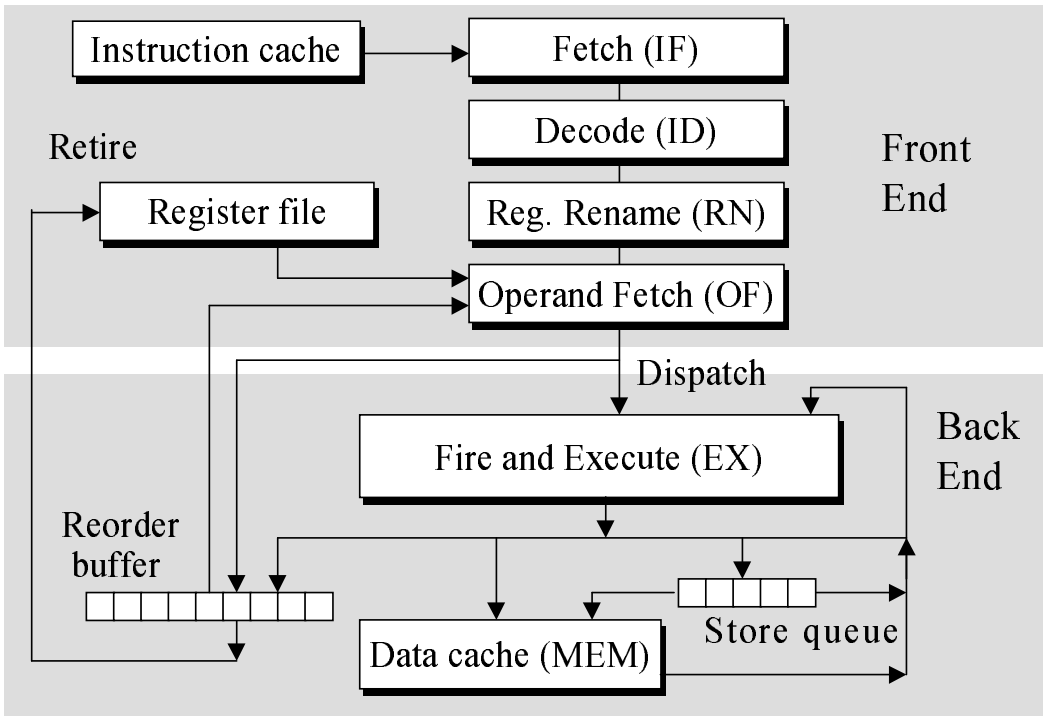



図 4.1: ベースラインプロセッサのブロック図

バッファから，オペランドの値または，値が利用できない場合には対応するタグを生成し，命令を命令ウィンドウに格納する．(5) 発火できる命令を検出し，機能ユニットは命令を実行する．(6) ロード・ストア命令はデータキャッシュまたはストアキューにアクセスする．

命令フェッチ・ステージからオペランドフェッチ・ステージまではインオーダに（命令がフェッチされた順番で）命令が処理される．これらをプロセッサのフロントエンドと呼ぶことにする．実行ステージ以降は入力オペランドが用意でき次第処理が開始され，アウトオブオーダに命令が処理される．これらをプロセッサのバックエンドと呼ぶことにする．

命令をフロントエンドから，バックエンドに進める動作をディスパッチ (**dispatch**) と呼ぶことにする．命令の処理が終了し，アーキテクチャ・ステートを更新するとともにリオーダバッファから命令を開放する動作をリタイア (**retire**) と呼ぶことにする．

ベースラインプロセッサでは命令のリタイアをインオーダに処理する．このためにリオーダバッファを利用する．リオーダバッファへのエントリ割り当てはオペランドフェッチ・ステージの終了時におこなう．この時，リオーダバッファのエントリに余裕がない場合には，一旦プロセッサのフロントエンドの処理をストールさせ，必要となるエントリ数を確保し

た後にフトントエンドの処理を再開する.

以降，ベースラインプロセッサの構成を検討し，その性能を明確にする．ただし，要素技術の幾つかは利用する命令セット・アーキテクチャに依存する．Alpha AXP アーキテクチャ [高澤 93] の利用を前提として議論を進める.

## 4.2 分岐予測

分岐予測の性能はプロセッサ性能に大きな影響を与える．分岐予測を用いなければ，分岐命令をフェッチするたびに，分岐先のアドレスが計算されるまで分岐命令以降の命令フェッチをストールする必要があり，十分な命令をプロセッサに供給できない．一方，予測ミスをおこなさい理想的な分岐予測を利用すると，後に評価する複数パス実行が性能を低下させているように見えてしまい，性能比較の基準としては適切ではない．以上の理由から，ベースラインプロセッサでは，現実的で，高い精度の分岐予測を利用することが望ましい.

本節では，Alpha AXP アーキテクチャにおける分岐命令（制御命令）を分類し，それぞれの分岐命令で利用する分岐予測を検討する.

Alpha AXP アーキテクチャにおける分岐命令は，次の 5 種類に分類できる.

**CALL_PAL 命令** オペレーティングシステムを実現するために使われるサブルーチンの集合として定義されるを特権アーキテクチャ･ライブラリを PAL(Privileged Architecture Library) コードと呼ぶ．PAL コードを起動するために利用される CALL_PAL 命令は分岐命令の一つだが，ベースラインプロセッサでは PAL コードを跨いだ投機実行をおこなわない．このため，CALL_PAL 命令の分岐先アドレスを予測する必要はない.

**条件分岐命令** レジスタ Ra をテストし，指定された関係が真ならば，21 ビットの変位と PC の加算で計算されるターゲット・アドレスに分岐する．そうでなければ，次の命令に実行が移る．条件分岐命令は PC 相対のみであり，21 ビットの符号付き変位は，前後 $2^{20} = 1M$ 個の命令の範囲内で分岐を可能とする.

**無条件分岐命令** 更新された PC（次の命令アドレス）をレジスタ Ra に書き込み，21 ビットの変位と PC の加算で計算されるターゲット・アドレスに分岐する．無条件分岐命令は PC 相対のみであり，21 ビットの符号付き変位は，前後 $2^{20} = 1M$ 個の命令の範囲で分岐を可能とする.

**ジャンプ命令** 更新された PC（次の命令アドレス）をレジスタ Ra に書き込み，レジスタ Rb が示すターゲット・アドレスに無条件で分岐する．

**RET 命令** RET(return from subroutine) 命令は一般には上のジャンプ命令に分類されるが，異なった分岐予測を利用するために別の分類とした．その動作は，ジャンプ命令と同様に，更新された PC（次の命令アドレス）をレジスタ Ra に書き込み，レジスタ Rb が示すターゲット・アドレスに無条件で分岐する．

CALL_PAL 命令を跨いだ投機処理をおこなわないため，CALL_PAL 命令に対する分岐予測は必要ない．無条件分岐命令は，前回の実行におけるターゲット・アドレスを分岐履歴テーブル (Branch Target Buffer, BTB) に保存し，この値を用いて分岐先を予測すればよい．ベースラインプロセッサでは，無条件分岐命令の分岐先アドレスを 100%の精度で予測できるとする．これらを除く 3 種類の分岐命令について，異なった分岐予測を利用する．それぞれの分岐予測の詳細を検討する．

### 4.2.1 条件分岐命令のための分岐予測

条件分岐命令の分岐予測には，2ビットカウンタ方式と 2 レベル適応型の一つ gshare[McF93a] のハイブリッド予測を利用する．図 4.2に gshare の構成を示す．分岐履歴レジスタ (Global Branch History Register) は最近の m 回のグローバルな分岐結果を保存するレジスタで，条件分岐命令の分岐結果が生成されるたびにレジスタの値を左に 1 ビットシフトし，LSB(Least Significant Bit) に分岐結果 (分岐成立の場合には 1 を不成立の場合には 0) を格納する．PHT(Pattern History Table) は 2 ビットの飽和型カウンタを $2^n$ エントリ持つテーブルである．分岐予測は次の 2 段階でおこなう．(1)BHR と PC の排他的論理和により PHT のインデックスを生成する．(2) 生成したインデックスを用いて PHT 内の 2 ビット飽和型カウンタを選択し，そのカウンタの MSB が 1 の場合には分岐成立と予測する．テーブルの更新は，分岐結果が成立だった場合に予測に用いた PHT の 2 ビットカウンタの値を 1 増加し，分岐不成立だった場合にカウンタの値を 1 削減する．

利用する PC のビット数と BHR のビット数を n とした時の gshare を gshare(n) と記述することにする．gshare(n) のハードウェアコストはビット数で計算して下の式となる．ここで最初の n は BHR のためのコスト，次の 2 が 2 ビットカウンタを用いることによる係数で，$2^n$ が PHT で必要となるエントリ数である．n が大きくなると BHR のコストは無視できるので，n を 1 増やすたびに gshare のハードウェアコストは 2 倍となる．

$$Hardware\ Cost = n + 2 \times 2^n \ \ bit$$

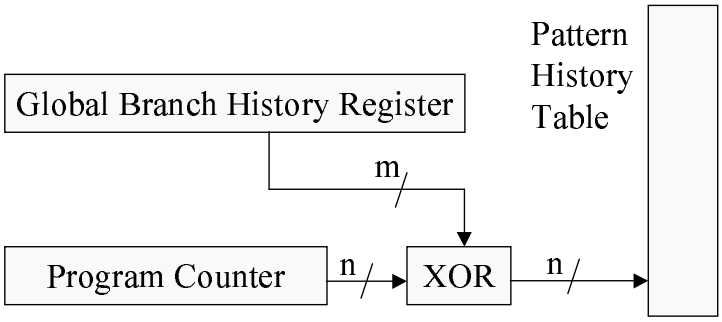



図 4.2: gshare 分岐予測のブロック図

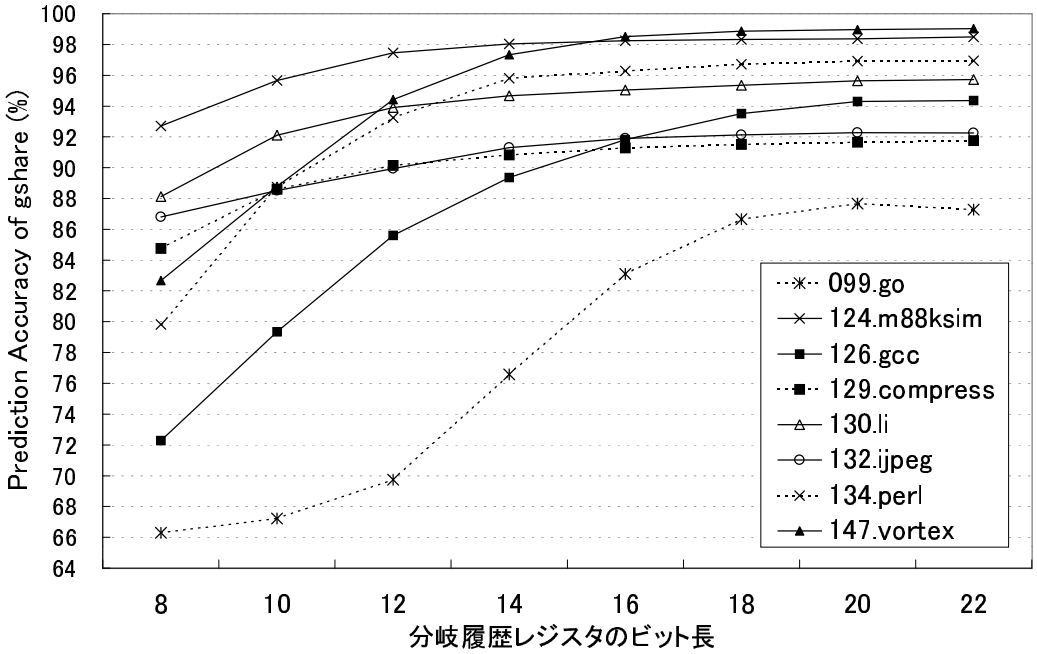



図 4.3: gshare における分岐履歴レジスタのビット長と分岐予測成功率

n を変化させて測定した gshare(n) の予測成功率を図 4.3に示す．このデータはスカラ実行のプロセッサを想定して測定した結果である．この結果より，gshare(20) が高い予測成功率を達成することがわかる．この時のハードウェアコストは約 2Mbit(256KB) となるが，数年先のプロセッサを考慮した場合には 256KB のハードウェア・コストは許容できない範囲ではない．また，分岐予測のハードウェアコスト削減を目指した研究 [野口 98, 山田 99] が盛んにおこなわれていることを考慮し，ハイブリッド予測機構を構成する要素として gshare(20) を用いることにする．

ハイブリッド予測機構を構成するもう一つの予測機構には，24K エントリの 2BC を利用する．2BC の予測結果と gshare 予測結果の選択に用いるセレクタには，24K エントリの 3 ビットカウンタを利用する．それぞれのカウンタは値 3 で初期化されており，2BC のみで分岐予測がヒットした場合にカウンタの値をインクリメントし，gshare のみで分岐予測がヒットした場合にはカウンタの値をデクリメントする．カウンタの値が 4 から 7 の時（カウンタの MSB が 1 の時）は 2BC の予測結果を，0 から 3 の時は gshare の予測結果をハイブリッド分岐予測機構の予測結果とする．

### 4.2.2 RET 命令のための分岐予測

リターン命令の分岐先アドレスは Return Address Stack(RAS) を用いて予測する．関数呼び出しの際に，ハードウェア・スタックにリターン・アドレスを積んでいく（プッシュする）．リターン命令がフェッチされた時点で RAS の先頭に格納されている命令アドレスをポップし，この値を分岐先アドレスとして予測する．プログラム実行において longjump あるいは setjump が利用されて関数呼び出しの関係が破壊される場合と，スタックの深さをこえて関数呼び出しが起きる場合を除いて，RAS を利用することで RET 命令のターゲットアドレスを正しく予測することができる．

### 4.2.3 ジャンプ命令のための分岐予測

近年のプロセッサは，ジャンプ命令の分岐先アドレスを予測するために branch target buffer(BTB) を用いている．BTB は前回の実行における分岐先アドレスを保存する履歴テーブルであり，分岐命令の PC を用いて履歴テーブルの読み出しと書き込みをおこなう．ジャンプ命令に関しては，BTB により 64%の予測成功率を達成できる [DH98] という結果が報告されているが，更なる予測成功率の向上を目指してパスベースの BTB[DH98] が提案されている．

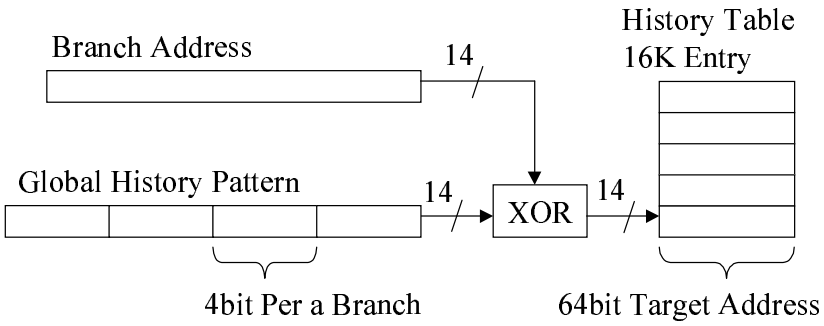



図 4.4: パスベース BTB のブロック図

ベースラインプロセッサではパスベースの BTB を用いてジャンプ命令の分岐先アドレスを予測する．予測機構の構成を図 4.4に示す．パスベース BTB では，分岐命令の PC(Branch Address) とグローバルな履歴パターン (Global History Pattern) の排他的論理和によりインデックスを作成する．インデックスを用いて履歴テーブルのエントリを指定し，そこに格納されている分岐先アドレスを予測アドレスとする．グローバル履歴パターンとは，ジャンプ命令が実行されるたびに分岐先アドレスの一部（命令アドレスの 0，1 ビット目は必ず 0 となるので，これらを除いた 2 ビット目からの n ビット）を連結したビット列である．ベースラインプロセッサで採用した構成では，ジャンプ命令が実行されるたびに 2 ビット目から 5 ビット目までの 4 ビットを連結してグローバル履歴パターンを作成する．また，16K エントリの履歴テーブル，14 ビットの履歴パターンを利用する．このため，過去の 3.5 回分のジャンプ命令の履歴を利用できる．

## 4.3 分岐予測ミスの検出

実行ステージにおいて分岐命令の分岐先アドレスが計算される．ベースラインプロセッサでは，メモリアクセス・ステージにおいて，予測された分岐先アドレスと計算された分岐先アドレスとを比較することで予測ミスを検出する．ただし，先行してフェッチされたすべての分岐命令に対する予測が正しかったことを確認して，予測ミスを検出する．このため，先行してフェッチされ，アドレス計算が終了していない分岐命令が存在する場合には，当該分岐命令をメモリアクセス・ステージでストールさせる．これにより，分岐予測ミスの検出をインオーダに処理できる．

予測ミスが発生した場合には，予測ミスを引き起こした分岐命令に続いてフェッチされ

た全命令をプロセッサからフラッシュし，正しい分岐先アドレスから命令フェッチを再開することで分岐予測の予測ミスから回復する．

## 4.4 ベースラインプロセッサにおける仮定

ベースラインプロセッサでは，資源競合により発生する制約を排除するために次の仮定を設けている．整数演算ユニット，浮動小数点演算ユニット，ロードユニット，ストアユニット，分岐ユニットは，無制限に利用できると仮定する．また，これらの実行ユニット間はフォワーディングパスで完全に結合されているとし，ある実行ユニットで生成されたデータは，次のサイクルには別の実行ユニットで利用できるとする．実行中の命令の演算結果を保存するリネーム・レジスタは，無制限に利用できると仮定する．これにより，リネーム・レジスタの不足によりプロセッサがストールすることはない．また，豊富なリネーム・レジスタを用いたレジスタリネーミングにより，レジスタを介した出力依存関係と逆依存関係を解消する．

2ウェイセットアソシアティブ，ラインサイズ64バイト，容量128KB，ライトスルーのL1データキャッシュを利用する．このキャッシュにミスした場合には，L2キャッシュからラインを転送する．このために20サイクルを必要とする．L2データキャッシュは，ミスを起こさない大容量のキャッシュを仮定する．これらのキャッシュのポート数には制限を加えていない．このため，毎サイクル，必要な量のデータを供給できる．

命令フェッチ機構による制約を排除するために，サイクル当たり任意のアドレスから16命令をフェッチできるとした．命令キャッシュはミスを起こさず，毎サイクル最大で16命令を供給できる理想的なものを仮定する．16命令の中には複数の分岐命令が存在する可能性があるが，複数の分岐命令に対して，先に示した分岐予測の結果を利用できるとする．

## 4.5 ロード命令とストア命令を発火するための条件

ロード命令を発火するためには，オペランドが利用できるという条件の他に，先行してフェッチされた全てのストア命令のメモリ参照アドレスが計算されているという条件を満たさなければならない．この条件を満たすことで，メモリアクセス・ステージにおいて，先行するストア命令とのメモリを介したデータ依存関係の解析が可能となり，ストアキューまたは，データキャッシュから正しいデータをロードすることができる．

ストア命令は2種類のオペランドを取る．一つは，ストアするデータ，もう一つは，メ

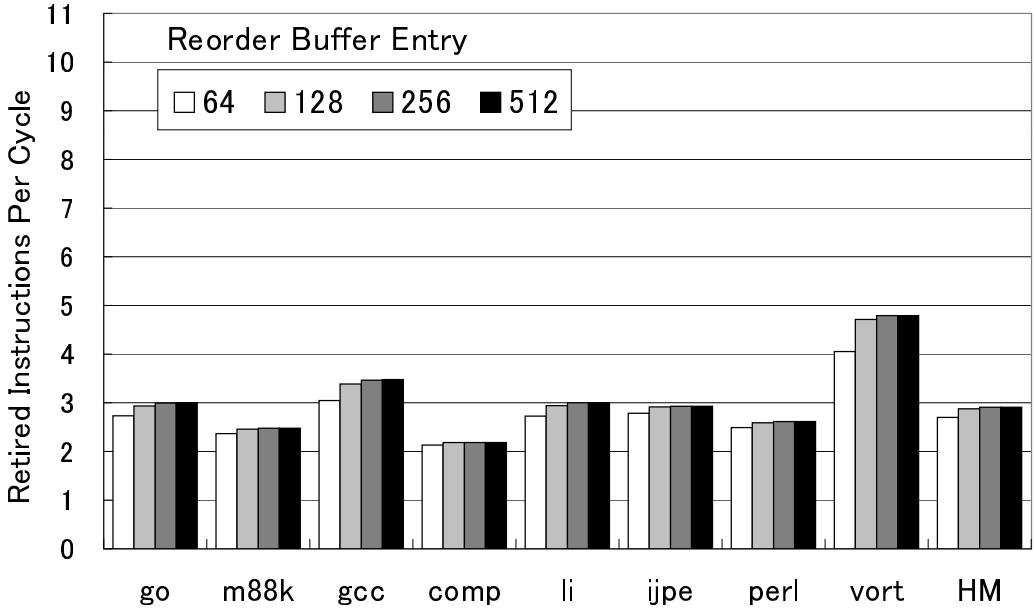



図 4.5: 全てのオペランドが揃った時点でストア命令を発火

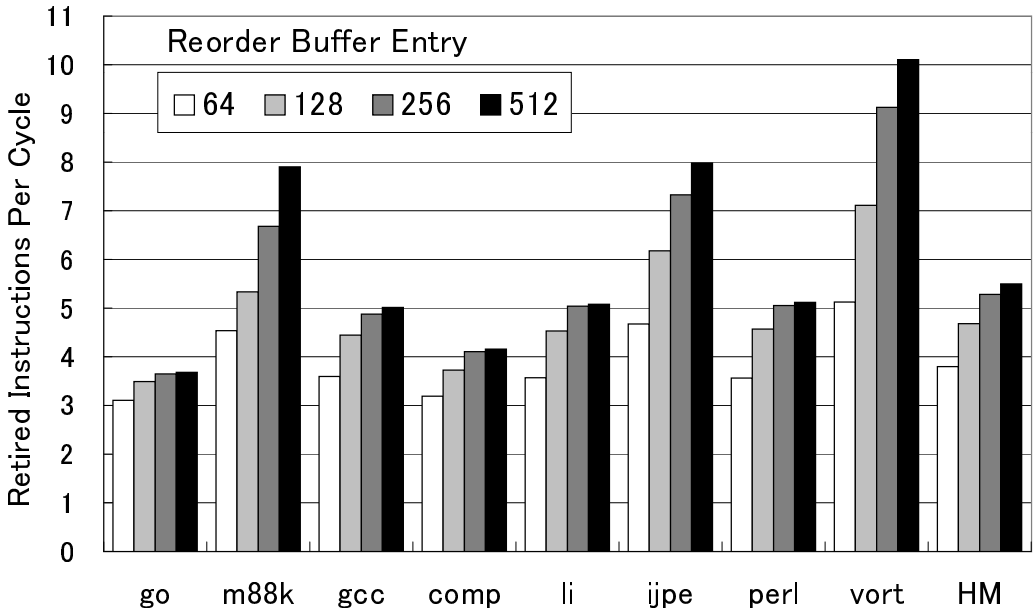



図 4.6: アドレス計算のためのオペランドが揃った時点でストア命令を発火

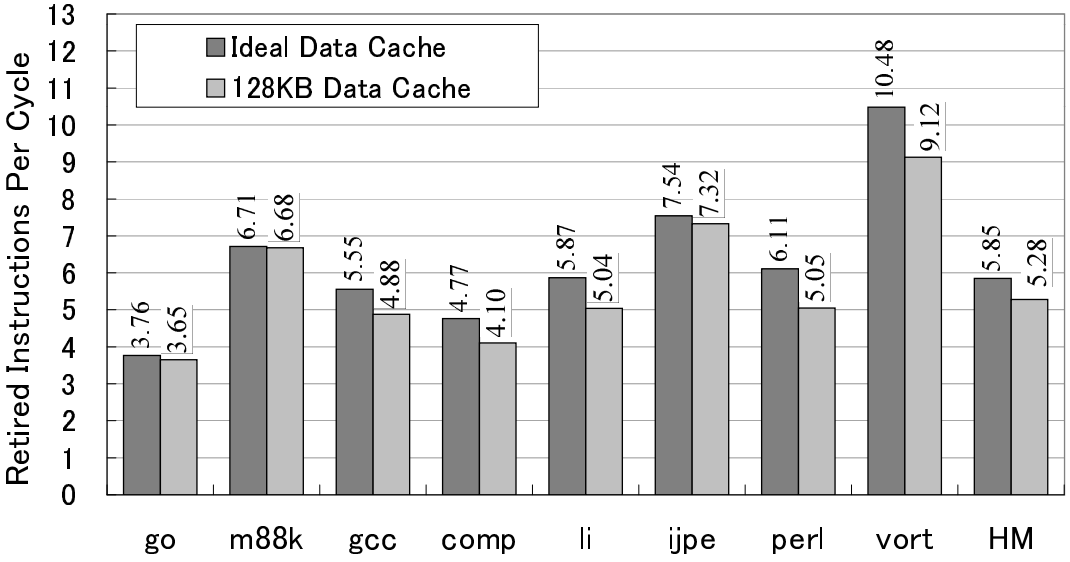



図 4.7: データキャッシュのミスを削除した場合の IPC

モリ参照アドレスを計算するためのオペランドである．通常のプロセッサでは，これら 2 つのオペランドが用意できた時点でストア命令のメモリ参照アドレスの計算（実行ステージ）を開始する．この設定を用いて，リオーダバッファのエントリを 64，128，256，512 に変化させた場合の IPC を図 4.5に示す．本研究において，**IPC(Instructions Per Cycle)** という言葉は，特に指定が無い限り，サイクル当たりにリタイアした平均の命令数を表す．これは，分岐予測ミス等でフラッシュした命令を除いた実行命令数 (表 3.2にまとめた実行命令数) を実行に必要となったサイクル数で割ることで計算される．

一方，アドレス計算のためのオペランドが利用可能となった時点でストア命令の実行を開始するように設計することができる．ただし，この場合には，実行ステージが終了してアドレスが計算された時点で，ストアすべきデータが準備できていない場合には，再度，パイプライン・ステージの進行を止める必要がある．この設定で測定した IPC を図 4.6に示す．

図 4.5と図 4.6との比較でわかるように，ストア命令のメモリ参照アドレスをできる限り早く計算することが IPC の向上につながる．リオーダバッファのエントリ数を大きくした場合に，その差は明らかとなる．リオーダバッファのエントリ数が 512 の時には，2 つのオペランドが揃ってから実行を開始する場合の IPC 2.9 に対して，早い段階でメモリ参照アドレスを計算する場合には IPC が 5.5 に向上する．これらの結果より，ベースラインプロセッサでは後者の設定を採用する．

| | |
|---|---|
| Pipeline Stages | IF/ID/RN/OF/EX/MEM |
| Fetch and Retire | 16 instructions/cycle |
| Branch Predictor | Hybrid of gshare and 2BC, 64 entry RAS, Path-based BTB |
| L1 D-Cache | 128KB, Write through, 2-way set associative, Line 64B, 20 cycle penalty for cache miss |
| L1 I-Cache, L2 Cache | Ideal |
| Reorder Buffer | 256 entry |

表 4.1: ベースラインプロセッサの主な構成パラメタ

図 4.6の結果からリオーダバッファのエントリ数と性能向上の関係を見ることができる．リオーダバッファのエントリ数を大きくすることで，より高い IPC を抽出できるが，その向上率は次第に小さくなっていく．ベースラインプロセッサでは 256 エントリのリオーダバッファを想定する．

## 4.6 データキャッシュ・ミスの影響

ベースラインプロセッサでは，128KB のデータキャッシュを利用する．データキャッシュのミスを削減することによる並列性の変化を図 4.7に示す．

図 4.7から，データキャッシュのミスを削減することで約 10%の IPC 向上を達成できることがわかる．データキャッシュのミスを削減する手法として，データプリフェッチやデータキャッシュの明示的な利用に関する研究がおこなわれているが，評価する際のパラメタの増加を抑えるために，本研究ではデータキャッシュの構成を先に示した構成に固定して評価する．

## 4.7 ベースラインプロセッサの性能

定義したベースラインプロセッサの主な構成パラメタを表 4.1にまとめる．

ベースラインプロセッサの命令レベル並列性を図 4.8 にまとめる．命令レベル並列性の平均には，8 つのプログラムの調和平均を用いる．グラフ中では，調和平均を HM と表記することがある．図 4.8にまとめたように，ベースラインプロセッサの IPC は 5.28 となった．

表 5.2に，ベースラインプロセッサにおけるデータキャッシュと分岐予測のミス率をまと

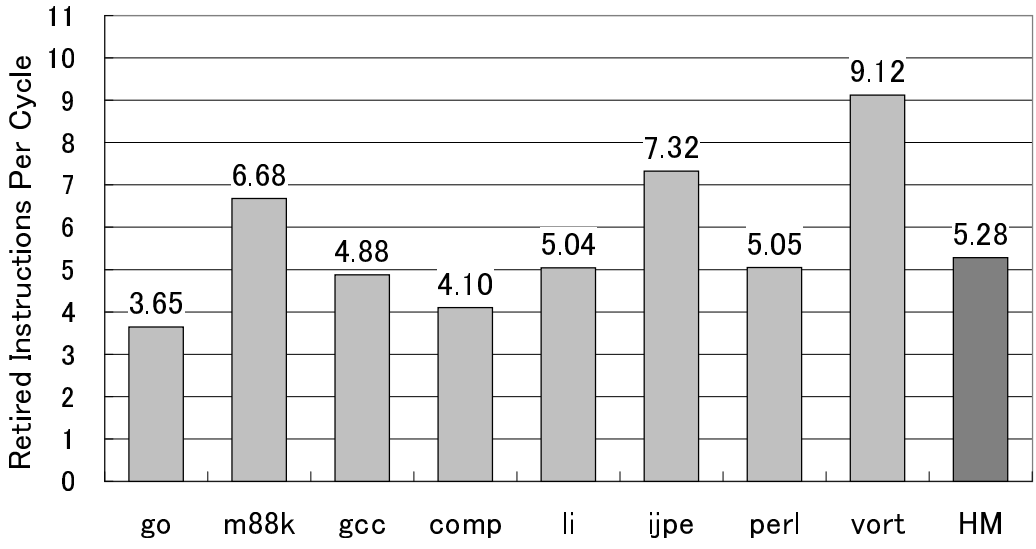



図 4.8: ベースラインプロセッサの IPC

| Program | Data Cache | Hybrid | Path-based BTB | RAS |
|---|---|---|---|---|
| go | 0.34% | 9.19% (97.4%) | 27.8% (2.64%) | 0.00% (0.00%) |
| m88ksim | 0.70% | 1.50% (80.8%) | 40.8% (19.1%) | 0.00% (0.00%) |
| gcc | 1.50% | 3.87% (85.8%) | 17.0% (14.1%) | 0.00% (0.00%) |
| compress | 1.96% | 8.21% (99.9%) | 21.5% (0.01%) | 0.00% (0.00%) |
| li | 1.87% | 4.24% (84.7%) | 14.8% (12.7%) | 0.56% (2.50%) |
| ijpeg | 0.27% | 7.16% (99.8%) | 2.50% (0.18%) | 0.00% (0.00%) |
| perl | 2.00% | 2.01% (90.2%) | 5.89% (8.63%) | 0.30% (1.15%) |
| vortex | 1.15% | 0.77% (95.4%) | 8.94% (4.58%) | 0.00% (0.00%) |
| Average | 1.22% | 4.62% (91.7%) | 17.4% (7.74%) | 0.10% (0.45%) |

表 4.2: データキャッシュのミス率と分岐予測のミス率

める．データキャッシュのミス率は 2.0%以下に抑えられ，平均のミス率は 1.22%となった．

表 5.2の 3 列から 5 列に，分岐予測のミス率をまとめる．括弧の中には，全分岐予測ミスに占める予測ミスの割合を示した．ハイブリッド分岐予測のミス率は全てのプログラムで 10%以下に抑えられており，平均のミス率は 4.6%となった．パスベース BTB の平均ミス率は 17.4%となった．標準的な BTB を用いた場合のミス率は 36%というデータがあり，パスベースの BTB を用いることで，ミス率が半減していることがわかる．RAS の予測ミスは li と perl の 2 つのプログラムで発生している．li では longjump が利用されているため RAS ミスが発生している．perl では 64 エントリのハードウェア・スタックに溢ることが原因で予測ミスが発生している．しかし，ミス率の平均は 0.45%と非常に低い．

予測ミス率では，パスベース BTB が最も悪い結果となった．しかし，全分岐予測ミスに占める割合で見た場合には，条件分岐の実行頻度が非常に高いために，条件分岐の予測ミスが全分岐予測ミスの 91.7%と大部分を占めることがわかる．

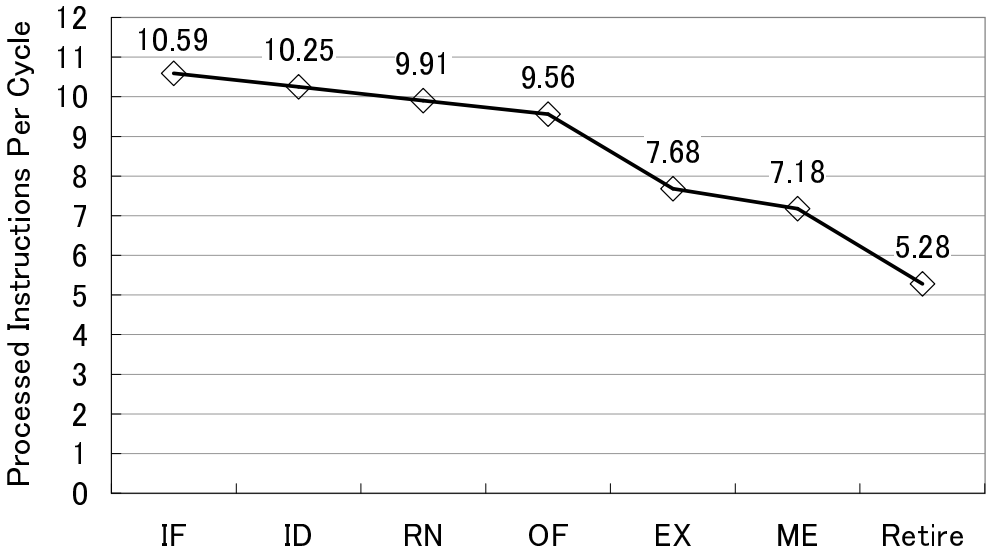



図 4.9: 各パイプライン・ステージにおいて処理される命令数

各パイプライン・ステージにおいて処理された平均命令数を図 4.9にまとめる．図 4.9の横軸は，命令パイプラインの各ステージを示している．ただし，右端にはリタイアする平均命令数 (IPC) を加えてある．縦軸はそれぞれのステージで処理された平均命令数を示している．リタイアする命令数 (IPC) の 5.28 に対して、フェッチする命令数は 10.59 と約 2 倍の命令をフェッチしていることがわかる．ベースラインプロセッサでは，分岐予測以外

の予測を用いていない．分岐予測のヒット率が 100%であるとすれば，フェッチした命令数とリタイアする命令数は等しくなる．

## 4.8 本章のまとめ

本章では，高レベル投機技術による性能向上を評価する際の比較として利用するベースラインプロセッサを定義し，その性能を測定した．

定義したベースラインプロセッサの IPC は 5.28 である．以降の章では，高レベル投機技術を利用して IPC の向上を目指す．

# 第 5 章

# 複数パス実行による分岐予測ミスペナルティの削減

分岐命令によるペナルティの削減を目指して精度の高い分岐予測が提案されている．2レベル適応型の一つ gshare[McF93a] と 2ビットカウンタ方式のハイブリッド分岐予測 [ECP96] では，SPECint95 を用いた評価において 95%に近いヒット率を達成する．しかしながら，残りの約 5%の分岐予測ミスが並列性の抽出を妨げる大きな制約となっている．

分岐ペナルティを削減する別のアプローチとして，条件付実行を用いた分岐命令の削除 [小沢 99] や，分岐成立と不成立の両方のパスにおいて投機的に処理を進めておくことで分岐予測ミスのペナルティを削減する複数パス実行が議論されている [安島 98, US95, Che98, 中村 99, KPG98, HS96].

本章では，複数パス実行により分岐ペナルティを削減することでプロセッサの性能向上を目指した新提案とその評価について述べる．複数パス実行で分岐ペナルティを削減するためには，パスの複製をおこなう分岐命令の選択手法（パス割付け戦略）の効率が重要となる．ここではまず，従来のパス割付け戦略 [US95, Che98, 中村 99, KPG98, HS96] を比較検討する．そして，各手法の利点を組み合わせる JRS-PM 方式を提案し，提案手法によって高い性能を達成可能となることを示す．評価には，命令レベル並列性に加えて，各パイプライン・ステージで処理される命令トラフィックと，データキャッシュのヒット率を測定し，複数パス実行の可能性を議論する．

## 5.1 関連研究

シングルパス実行と，Eager Execution と呼ばれる全ての分岐命令でパスを複製する複数パス実行の様子を図 5.1に示す．図 5.1における丸印は分岐命令間の命令列を表しており，これを分岐パス (branch path) と呼ぶ．ただし，静的解析で用いる基本ブロックと混同する恐れがない場合には，分岐パスのことを**基本ブロック**と呼ぶことにする．図では，それぞれの基本ブロックに，そこに至る分岐予測の確信度の積によって計算される**累積確信度**を付けた．基本ブロック間の矢印は制御依存関係を表しており，矢印に付けた確率は，その分岐命令に対する分岐予測の確信度である．図 5.1では全ての分岐予測の確信度を 70% とした．複数パス実行において，各分岐時点で常に確信度の高い方の分岐を選ぶことによってできる 1 本のパスをメインパスと呼ぶ．

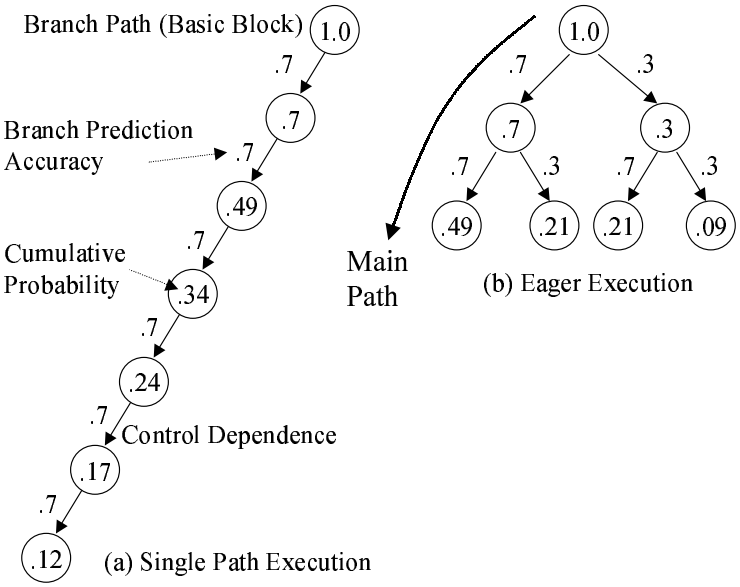



図 5.1: シングルパス実行と Eager Execution の比較

Eager Execution では，多くの分岐を跨いでパスを展開するに従って，必要となるハードウェア資源が爆発的に増加する．このため，効率的にパスを複製する手法が課題であり，研究が進められている．

### 5.1.1 確率モデルを用いた複数パス実行

累積確信度の最も高い幾つかのパスを同時に実行する方式を，確率モデルを用いた複数パス実行と呼ぶことにする．これは，コントロールフロー先行展開におけるパス展開方式として，我々が文献 [中村 97b] で提案しているものである．また，文献 [US95] では Disjoint Eager Execution(DEE) として，文献 [Che98] では Adaptive Branch Tree(ABT) として記述されている．

#### Adaptive Branch Tree

文献 [Che98] では，累積確信度の計算と，最も高い累積確信度を持つ基本ブロックの選択を動的に処理する Adaptive Branch Tree(ABT) を提案している．文献 [Che98] では，分岐予測の確信度を浮動小数点形式ではなく，$-\log_2(x)$ という関数を用いて整数に変換してテーブルに保存する方式を提案している．このように変換した形式を用いることで，確信度の加算によって累積確信度を計算できるようになる．ただし，動的な分岐予測の確信度計算は検討されていない．

| $p$ | -$\log_2 p$ | -$\log_2 p \times 2^5$ | 8-bit int rep. |
|---|---|---|---|
| 0.004 | 7.965784 | 254.91 | 254 |
| 0.09 | 3.473931 | 111.16 | 111 |
| 0.1 | 3.321928 | 106.30 | 106 |
| 0.5 | 0.1 | 32.00 | 32 |
| 0.9 | 0.152003 | 4.86 | 4 |
| 0.97 | 0.043943 | 1.40 | 1 |

表 5.1: Transform Probability into Integers

確信度を 8 ビットの整数に変換する例を表 5.1に示す．表 5.1の左端から順番に，浮動小数点で示した確率 $p$，$-\log_2 p$ の値，$-\log_2 p$ の値を 8 ビットの整数に変換するために $2^5$ を掛けた値，右端が変換後の 8 ビットの整数を表している．この様に確率を表現した場合には，最大の累積確信度の選択は，最も小さい整数を選択するという作業で実現することができる．

文献 [Che98] では，最も高い累積確信度を選択する機構について議論している．パスの複製で必要となる最大の累積確信度を持つ分岐命令の選択のために，全てのパス複製の候

補をソートする必要はない．今，5 つの分岐命令 (L1, L2, L3, L4, L5) がパス複製の候補として存在し，それぞれの候補は累積確信度によってソートされているとする．図 5.2では，最も大きい累積確信度が L1, 以下, L2，L3, L4, L5 と続くとする．この状態からパスを複製する場合には，最も高い累積確信度を持つ分岐命令 L1 でパスを複製し，その子となる 2 つのパスの確信度 Child 1, Child 2 を計算する．その後，L1 の子で高い確信度を持つもの (Child 1) と L2 を比較することで，最も大きい累積確信度を持つ候補 (N1) を得ることができる．また，図に示した構成の比較器を用いて，ソートされた候補 (N2, N3, N4, N5, N6) を得る．図の構成では，全ての状態を更新するために 5 サイクルを必要とするが，パイプライン処理することで，毎サイクル 1 回のパス複製が可能となる．

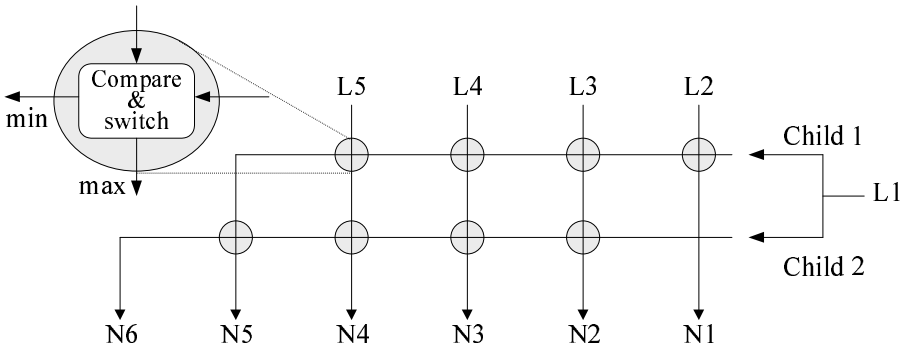



図 5.2: 最大の累積確信度を選択する機構

### Bit Order 方式

文献 [中村 99] では，動的に分岐予測の確信度を計算する Bit Order 方式を提案している．この方式では，それぞれの分岐命令における過去の $n$ 回の予測成功・不成功の履歴を $n$ ビットの履歴レジスタに格納する．履歴レジスタはシフトレジスタになっており，分岐予測の成功または不成功により，1 または 0 を最下位ビットに連結していく．

5 ビットの履歴レジスタから確信度を求める変換規則を図 5.3に示す．過去の 5 回の分岐予測の中で，予測に成功した回数 (count) に応じて図 5.3 に示した 5 ビットのビット列を生成する．このビット列が分岐予測の確信度を表している．

それぞれのパスに至る累積確信度は，図 5.3の変換規則で得られたビット列を，制御依存グラフのルートから葉の方向に向かって連結していくことで生成する．Bit Order 方式は，それぞれの分岐予測結果に対する確信度の計算，フェッチ・ポイントに至る累積確信度の

```
count=5 -> 11111,   count=4 -> 11001
count=3 -> 10011,   count=2 -> 01100
count=1 -> 00110,   count=0 -> 00000
```

図 5.3: Bit Order 方式における動的な確信度予測

計算，複数のフェッチ・ポイントから最も高い累積確信度を持つものの選択，を動的におこなう点に特徴がある．

### Static Tree Heauristic

Static Tree Heauristic[US95] では，全ての分岐命令で共通に利用する確信度を設定し，この確信度と利用できるハードウェア資源からパス展開の形を静的に定義する．このため，それぞれの分岐予測に対する確信度の計算，フェッチ・ポイントに至る累積確信度の計算，複数のフェッチ・ポイントから最も高い累積確信度を持つものの選択，といったハードウェアが不要となり複数パス実行で追加すべきハードウェアを簡略化できる．

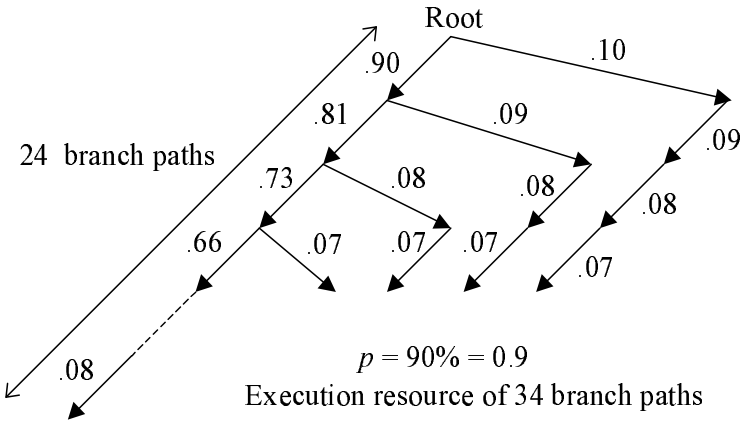



図 5.4: Static Tree Heauristic によるパス管理の例

分岐予測の確信度を 90%，利用できる分岐パスが 34 という設定におけるパス展開の例を図 5.4に示す．図 5.4における矢印は分岐パスを表し，それぞれの分岐パスには，そこに至る累積確信度が付加されている．利用できる分岐パスの数が 34 の時には，累積確信度が 70%以上の分岐パスがフェッチされることがわかる．

### 5.1.2 Selective Eager Execution

確率モデルとは異なる複数パス実行の方式として Selective Eager Execution(SEE)[KPG98, HS96] が提案されている．SEE では，分岐予測の確信度が高い/低いという 1 ビット出力の確信度評価器を用いて，確信度が低いと評価された分岐命令でのみパスの複製をおこなっていく．

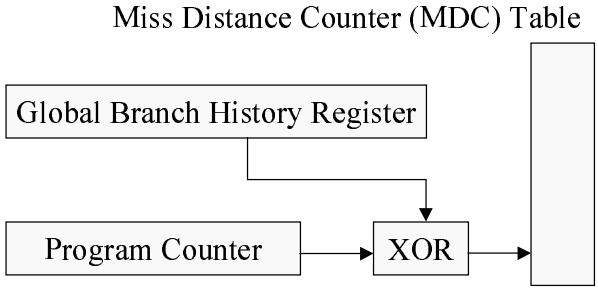



図 5.5: JRS 確信度評価器

SEE で重要な役割を果たす 1 ビット出力の確信度評価器として，Jacobsen-Rotenberg-Smith (JRS) 確信度評価器 [GKMP98] が提案されている．JRS 確信度評価器の構成を図 5.5に示す．Miss Distance Counter テーブルは，最近の分岐予測ミスからの距離，言い換えれば，連続して分岐予測が成功した回数を保存するカウンタを複数持つテーブルである．分岐命令のアドレスとグローバルな分岐履歴レジスタの排他的論理和によりテーブルのインデックスを生成し，該当するカウンタの値を読み出す．この値があるスレッショルドより大きい場合には，確信度が高い（分岐予測が成功する），そうでない場合には確信度が低い（分岐予測が失敗する）と評価する．分岐予測が正しかった場合に該当カウンタの値をインクリメントする．一方，分岐予測が間違っていた場合にカウンタの値を 0 で初期化する．

図 5.6に JRS 確信度予測器を実装し，評価した結果を示す．確信度が高いと判断するまでのスレッショルドを 1〜15 まで変化させて評価した．横軸は分岐予測にミスした分岐命令の中で確信度が低いと評価された命令の割合で，100%に近いほど多くの分岐ミスを排除できることになる．縦軸は，実行された分岐命令の中で確信度が低いと予測されたものの割合で，100%は Eager Execution を，0%は Single Path Execution を意味する．スレッショルド 15 における compress の評価結果を例にとると，全分岐命令の 29%でパスを複製することで，分岐予測ミスの 96%を削減できることがわかる．図 5.6の様に結果を表現した場合

には，グラフの右下にプロットされた構成が良い確信度評価器となる．図 5.6の結果より go の性能が特に悪いことがわかる．go はハイブリッド分岐予測のミス率が 9.1%と最も悪く，かつ確信度評価の性能が悪いという結果は深刻な問題である．それ以外のプログラムでは，30%程度の分岐命令でパスを複製することで，7 割以上の分岐予測ミスを削減できる．

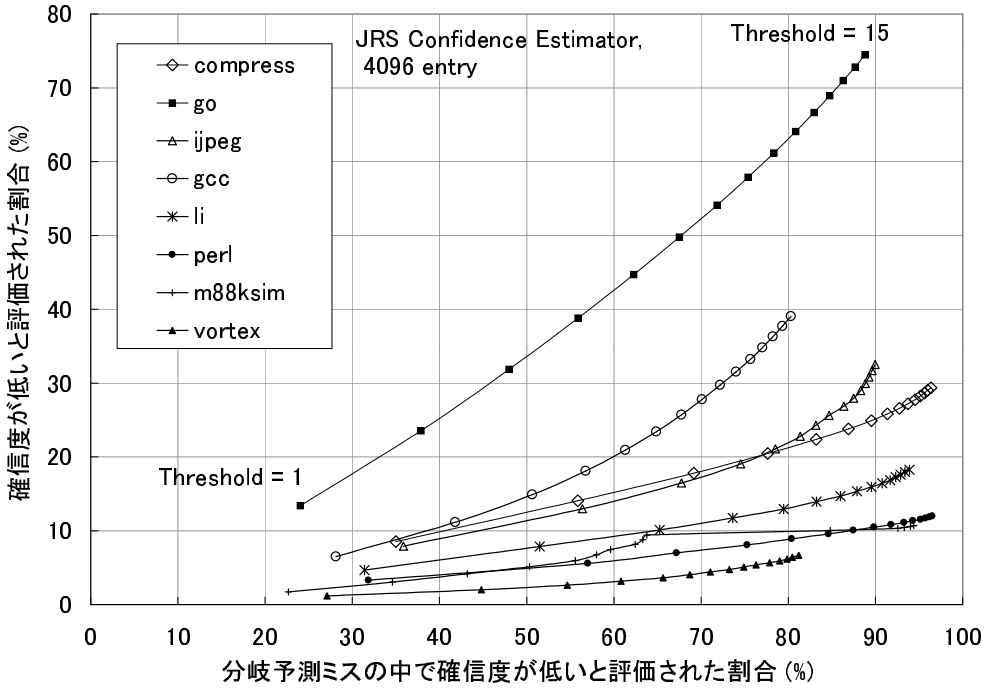



図 5.6: JRS 確信度予測器の評価結果

### Selective Dual Path Execution

Selective Eager Execution において利用できるパスの数を 2 つに制限した Selective Dual Path Execution(SDPE) について文献 [HS96] で議論されている．文献 [HS96] では，SDPE におけるパス管理方式として，Canceled Path Policy, First Delayed Policy, Last Delayed Policy の 3 つの戦略を提案し，その性能を評価している．

3 つの戦略の概要を図 5.7に示す．Canceled Path Policy は，パスが複製され 2 つのパスで処理が進んでいる間にフェッチされた分岐命令ではパスの複製をおこなわない．図 5.7(a) では分岐命令 1 の結果が計算されて，パスが開放されるまでの間にフェッチされた分岐命令 2, 3, 4 においてパスは複製されない．パスが開放された後にフェッチされた分岐命令 5

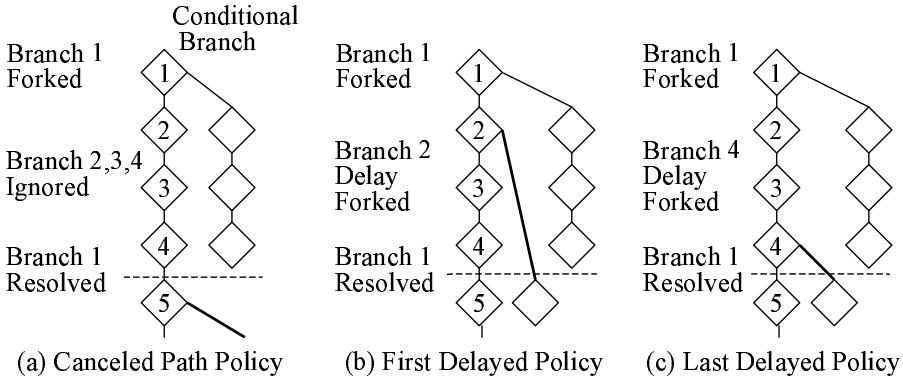



図 5.7: Selective Dual Path Execution における 3 つのパス割付け戦略

においてパスが複製される.

First Delay Policy では，パスが複製された次の分岐命令が複製の候補となる．図 5.7(b) の例では，分岐命令 2 が次の複製の候補となり，分岐命令 1 の結果が計算され，パスが開放されると同時に，分岐命令 2 においてパスを複製する.

Last Delay Policy では，分岐命令 1 の結果が計算され，パスが開放された時点で，最も最後にフェッチされた分岐命令を複製の候補とする．図 (c) の例では，分岐命令 4 においてパスを複製する.

図 5.7では，全ての分岐命令がパス複製の候補となるとして描いているが，文献 [HS96] では JRS 確信度評価器 [GKMP98] を用いて分岐予測の確信度が低いと評価された分岐命令のみを複製の候補としている．文献 [HS96] の評価によると Canceled Path Policy の性能が最も悪く，最も性能の良かった戦略が Last Delayed Policy だったという結果が出ている.

### 5.1.3 パス割付け戦略とアーキテクチャ

5.1.1節で述べた 3 つの確率モデルの違いは，(1) 分岐予測に対する確信度の計算，(2) フェッチ・ポイントに至る累積確信度の計算，(3) 最も高い累積確信度を持つ基本ブロックの選択，といった処理を動的におこなう度合いの違いといえる．これら動的処理の度合いは，後述するように，システム性能を大きく変える．また，アーキテクチャの複雑さに大きな影響を与える．このことは SEE についても同様である．このため，パス割付け戦略とプロセッサ性能の関係を明確にすることは重要な意味を持つ.

| Program | Executed Inst. | Hybrid | Path-based BTB | RAS | IPC |
|---|---|---|---|---|---|
| go | 138.9 mil. | 9.19% (97.4%) | 27.8% (2.64%) | 0.00% (0.00%) | 3.64 |
| m88ksim | 127.4 mil. | 1.50% (80.8%) | 40.8% (19.1%) | 0.00% (0.00%) | 6.67 |
| gcc | 151.9 mil. | 3.87% (85.8%) | 17.0% (14.1%) | 0.00% (0.00%) | 4.87 |
| compress | 142.0 mil. | 8.21% (99.9%) | 21.5% (0.01%) | 0.00% (0.00%) | 4.10 |
| li | 208.1 mil. | 4.24% (84.7%) | 14.8% (12.7%) | 0.56% (2.50%) | 5.04 |
| ijpeg | 172.3 mil. | 7.16% (99.8%) | 2.50% (0.18%) | 0.00% (0.00%) | 7.32 |
| perl | 153.9 mil. | 2.01% (90.2%) | 5.89% (8.63%) | 0.30% (1.15%) | 5.05 |
| vortex | 185.1 mil. | 0.77% (95.4%) | 8.94% (4.58%) | 0.00% (0.00%) | 9.12 |
| average | 160.0 mil. | 4.62% (91.7%) | 17.4% (7.74%) | 0.10% (0.45%) | 5.28 |

表 5.2: ベンチマークプログラムの実行命令数と分岐予測の失敗率

## 5.2 分岐予測ミスによる性能低下の検討

複数パス実行の可能性を調査するために，分岐予測ミスがプロセッサの性能に与える影響を議論する．測定には 4節で定義したベースラインプロセッサを利用する．

分岐予測のミス率を測定した結果を表 5.2 の 3 列から 5 列にまとめる．括弧の中には，ハイブリッド分岐予測，パスベース BTB，RAS の予測ミスが全分岐予測ミスに占める割合を示した．ハイブリッド分岐予測のミス率は全てのプログラムで 10%以下に抑えられており，平均のミス率は 4.6%となった．また，条件分岐の予測ミスが全分岐予測ミスの 91.7%と大部分を占めることがわかる．以上の結果より，条件分岐のみを複数パス実行の候補とする．

なお，複数パス実行に適したレジスタ間接分岐のための分岐予測は文献 [中村 97a] で議論しているが，その利用に関しては今後の課題とする．

本章における以降の文章では，条件分岐のことを分岐，条件分岐のための分岐予測を単に分岐予測と呼ぶことにする．

分岐予測のヒット率 95.4%は高い精度に見えるが，残りの 4.6%の予測ミスが並列性の抽出に対する大きな制約となる．確率的に分岐予測ミスを削除した場合の IPC を測定することで，分岐予測ミスを削減することによる性能向上率を調査する．この調査では，予測された分岐先アドレスとあらかじめ収集した正しい分岐先アドレスとを比較し，分岐予測が間違っていた場合には，乱数を用いて指定された確率で分岐予測を正しい結果に変更した．

測定結果を図 5.8に示す．図の横軸は分岐予測ミスの削減率で，図 5.8左端の削減率 0%はベースラインプロセッサにおける並列性，右端の削減率 100%は分岐予測ミスを完全に削

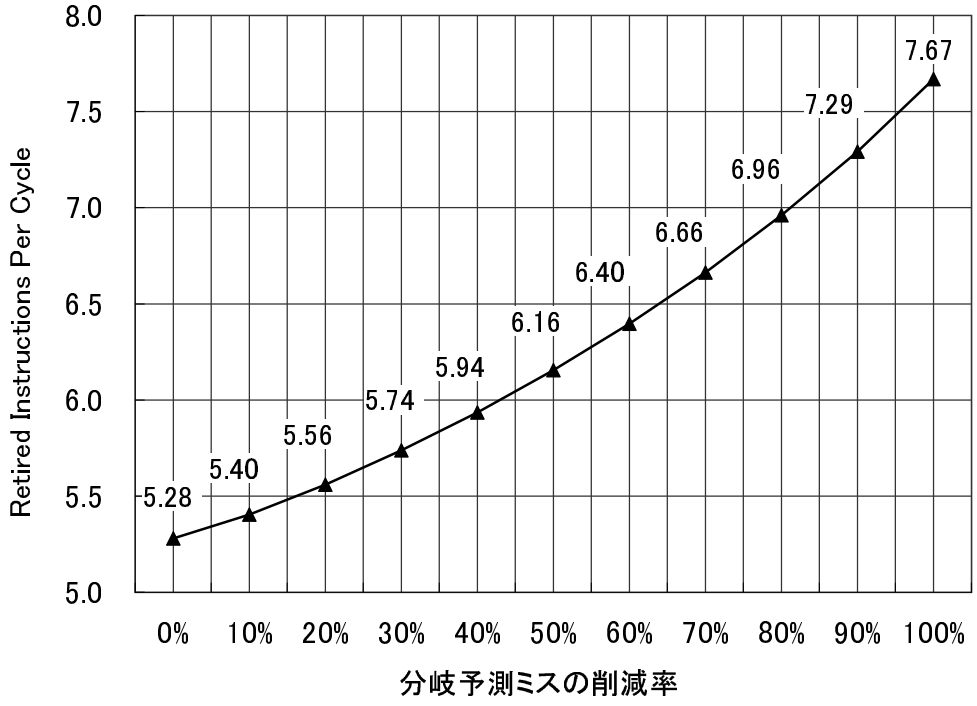



図 5.8: 分岐予測ミスを削減した場合の並列性

減した場合の並列性を示している．ただし，レジスタ間接分岐の予測ミスは削減していない．ベースラインプロセッサにおける IPC 5.28 に対して，分岐予測ミスを削減することで IPC が 7.67 に向上する．このように，4.62%の分岐予測ミスを削減することで最大 45%の IPC の向上を達成できることがわかる．

複数の分岐予測ミスが互いに影響を及ぼさないとすれば，分岐予測ミスの削減率を上げることで，線形に並列性が向上することを期待できる．しかし，図 5.8に示した測定結果を見ると，並列性の向上は線形ではなく曲線を描いていることがわかる．これは，予測ミスの削減率と予測ミスペナルティの削減率の間にずれが発生していることを意味している．分岐予測の精度が低く頻繁に分岐予測ミスが発生する場合や，分岐予測ミスが時間的に集中して発生する場合には，集中して発生した分岐予測ミスの全てを削減しなければ，分岐予測のミスペナルティを効率的に削減することはできない．これらの原因により，予測ミスの削減率と予測ミスペナルティの削減率の間のずれが発生していると考えられる．以上により，予測ミスの削減率ではなく，予測ミスペナルティの削減率あるいは IPC を用いて複数パス実行の性能を評価しなければならない．

## 5.3 JRS-PM 方式の提案

### 5.3.1 パス割付け戦略

複数パス実行の複雑さを軽減するために，パスを割り当てる分岐命令を選択するパス割付け**戦略**と，それぞれのパスからフェッチする命令数を制御する**資源割付け戦略**に分けてパスの管理方式を議論する．

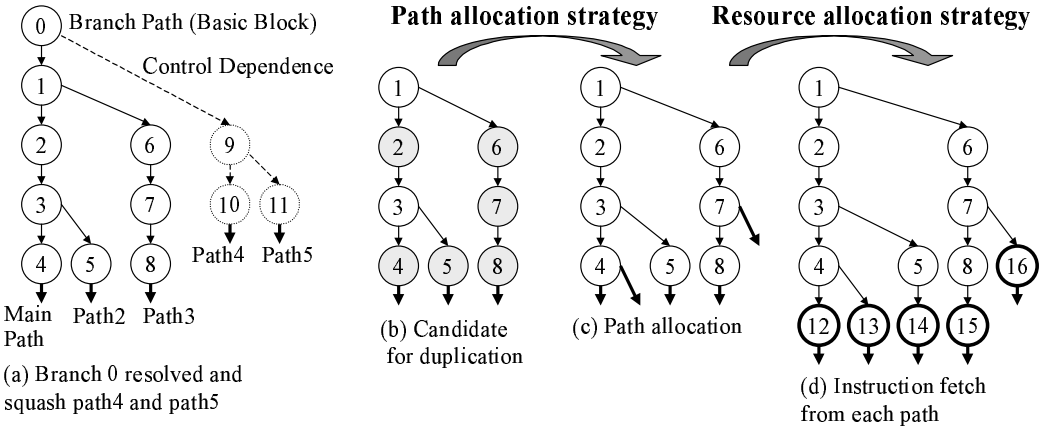



図 5.9: パス割付け戦略と資源割付け戦略

2 つの戦略の役割を図 5.9の例を用いて説明する．図 5.9は，5 つのパスを利用できる場合の例で，(a) は，分岐 0 の結果が判明し，基本ブロック 9,10,11 を含む 2 つのパスが開放される様子を表している．(b) は，2 つのパスが開放された状態で，開放されたパスを割り当てる候補となる基本ブロック（分岐命令）が 6 つ存在することを表している．分岐 1,3 では既にパスが複製されているので，パス複製の候補とする必要はない．(c) は，なんらかの戦略によって，6 つの分岐の中から分岐 4,7 が選択され，これらの分岐命令にパスが割り当てられたことを表している．ここではサイクル当たり 2 つのパスの割付けが可能とした．この時，パス複製の候補の中から，パスを割り当てる分岐命令を選択する戦略をパス割付け戦略と呼ぶことにする．(d) は，(c) からプロセッサの状態が 1 サイクル進み，それぞれのパスから命令がフェッチされた状態を表している．ここでは全てのパスから基本ブロックを 1 つフェッチできるとした．このように，各パスからフェッチする命令数を制御する戦略を資源割付け戦略と呼ぶことにする．

本章では，それぞれのパスからサイクル当たり 16 命令をフェッチ可能とする理想的な資

源割付け戦略を想定し，パス割付け戦略の性能を議論する．

### 5.3.2 確率モデルと SEE の方式比較

確信度を評価するアルゴリズムから確率モデルと SEE を比較する．確率モデルの利点はプロセッサが処理している分岐命令の中から，パスを複製する分岐命令を選択できる点にある．このため，グローバルな確信度の挙動を捕らえることができる．一方，JRS 確信度評価器を利用する SEE では，グローバルな確信度を利用せずに，当該分岐命令のローカルな確信度の履歴を用いて予測をおこなう．

ハードウェアコストの点から 2 つの方式を比較する．確率モデルでは，プロセッサが処理している分岐命令をバッファに格納し，その中から最大の累積確信度を持つ分岐命令を選択する必要がある．このため，プロセッサ内に存在する分岐命令の数が増加するに従ってハードウェアコストが増大する．一方，SEE では処理している分岐命令の情報を保存しておく必要はないため，上に述べたハードウェアが不要になるという利点がある．

### 5.3.3 JRS-PM 方式の提案

JRS 確信度評価器と確率モデル (PM) を同時に利用する JRS-PM 方式を提案する．先に述べたように，2 つの方式は異なるアルゴリズムを用いて確信度を計算する．このため，この 2 つを利用することで確信度評価の精度を向上させ，不必要なパス複製を削減できる可能性がある．

また，JRS 確信度評価器を用いて確信度が高い (分岐予測の成功する確率が高い) と評価された分岐命令をパス複製の候補から除外することで，確率モデルで発生するパス複製のための候補数の増加によるハードウェアコスト増大の問題を緩和できる可能性がある．

提案する JRS-PM 方式は，(1) 命令フェッチ時に JRS 確信度評価器を用いて，分岐予測の確信度が低いと評価された分岐命令をパス複製の候補とする．(2)JRS 確信度評価器で確信度が低いと評価された分岐命令の中から確率モデルを用いた優先度付けによりパスを複製する分岐命令を選択する．

## 5.4 評価環境

複数パス実行の評価には，スカラプロセッサにおいて実行される命令の他に，分岐命令の結果が得られた時点でフラッシュされる本来必要でない命令の実行による影響を考慮す

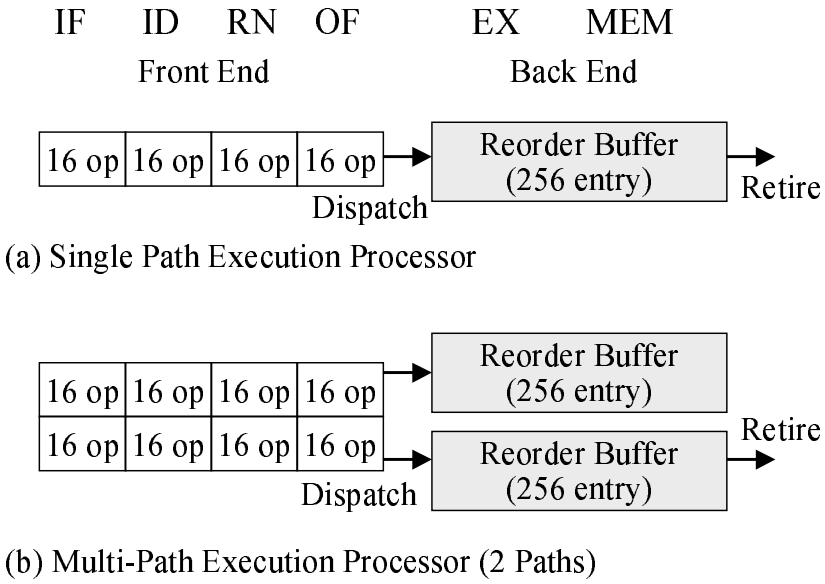



図 5.10: 複数パス実行プロセッサとベースラインプロセッサの比較

る必要がある．従来用いられてきたトレース駆動のシミュレーションでは，この必要性を満たすことはできない．このため，複数パス実行をおこなうプロセッサの状態をサイクル・レベルで更新する実行ベースのシミュレータを新たに開発し，これを用いて複数パス実行プロセッサの性能を評価する．

### 5.4.1 複数パス実行プロセッサ

複数パス実行プロセッサのパイプライン・ステージ構成や，データフォワーディング，分岐予測などはベースラインプロセッサと変わらない．すなわち，利用できるパスの数を 1 に設定した複数パス実行プロセッサの挙動はベースラインプロセッサと同じである．

図 5.10にベースラインプロセッサと複数パス実行プロセッサとの比較を示す．パスの数を 2 以上に設定した複数パス実行プロセッサは，パスの数だけベースラインプロセッサを多重化することで実現する．ただし，全てのパスが独立に動作する訳ではない．プロセッサのパイプライン構成はインオーダに処理が進むフロントエンドと，アウトオブオーダに処理が進むバックエンドに分けられる．複数パス実行プロセッサでは，それぞれのパス毎に独立した 256 エントリのリオーダバッファを想定するが，どれか一つのパスでリオーダバッファが溢れる場合にはプロセッサのフロントエンド全体をストールさせる．

単純にベースラインプロセッサを多重化した場合には，パスの数に比例して必要となるハードウェア資源が増加する．しかし，全てのパスにおいて最大のトラフィックで命令が処理される訳ではないし，プロセッサのバックエンドでは物理レジスタや演算器などを全てのパスで共有することが可能である．これらの要因により，ハードウェア資源を削減できる可能性がある．一方，データフォワーディングの複雑さは，発火を待つ命令数の二乗に比例して増加する．複数パス実行は，発火を待つ命令数を増加させるため，データフォワーディングがネックとなるとすれば，パスの数の二乗に比例してハードウェア量が増加する可能性もある．複数パス実行プロセッサで必要となるハードウェア資源の見積もりに関しては詳しい検討が必要となる．

### 5.4.2 パス割付け戦略

提案手法の JRS-PM 方式に加え，次に示す Selective Eager Execution と，2 つの確率モデルを利用した場合の複数パス実行プロセッサを評価する．

**SEE** 分岐命令をフェッチする際に，JRS 確信度評価器により確信度が高いと評価された分岐命令をパス複製の候補とし，利用できるパスの資源が余っている場合に，その候補でパスを複製する．ただし，パス複製の候補が利用できるパス数より多い場合には，メインパスの候補を優先し，それ以外の優先度はランダムとして，複製する分岐命令を決定する．

**PM** 浮動小数点形式で確信度と累積確信度を表現する．分岐予測の確信度は予測をおこなう分岐命令の過去の全ての実行において，分岐予測が正しかった回数を分岐予測をおこなった回数で割ることで求める．それぞれのフェッチ・ポイントにおける累積確信度は，制御依存グラフのルートからフェッチ・ポイントに至る分岐予測の確信度の積により計算する．利用できるパスの資源が余っている場合に，最も高い累積確信度を持つ分岐命令でパスを複製する．

**PM-static-tree** 全ての分岐予測の確信度として，表 5.2の 4 列目の平均に示した確信度 (95.8%) を利用する．累積確信度は，制御依存グラフのルートからフェッチ・ポイントに至る分岐予測の確信度の積により計算する．利用できるパスの資源が余っている場合に，最も高い累積確信度を持つ分岐命令でパスを複製する．この構成は Static Tree Heuristic の近似となる．

パスの複製には，レジスタマップの複製や，確率モデルを用いた場合には最も高い累積確信度を持つ分岐命令の選択が必要となる．後者は，図 5.2に示した構成のハードウェアを用いて実現できるが，サイクル当たりに複数回のパス複製をおこなうことは困難である．この事を考慮し，サイクル当たりのパス複製回数を 1 回に制限する．

## 5.5 評価結果

命令レベル並列性 (IPC) に加えて，各パイプライン・ステージで処理される命令トラフィックと，データキャッシュのヒット率の測定結果を示す．従来迄に，複数パス実行における命令トラフィックとデータキャッシュのヒット率を評価している研究はみられない．

### 5.5.1 IPC の変化

利用できるパスの数を 2, 4, 8, 16 に設定して IPC を測定した結果を図 5.11に示す．それぞれのパスの設定について，右から JRS-PM, PM, PM-static-tree, SEE, 比較のためにシングルパス実行 (Baseline) の IPC をまとめた．全てのパス設定において PM-static-tree の IPC が最も悪く，PM と JRS-PM が高い IPC を示した．最も高い IPC を示した JRS-PM における分岐ミスペナルティの削減率では，パスの数 2, 4, 8, 16 それぞれの場合にそれぞれ 21%, 40%, 52%, 59%となった．

パス割付け戦略と利用できるパスの数を変化させ，サイクル当たりのフェッチ命令数を測定した結果を図 5.12にまとめる．IPC では，PM と JRS-PM の間に著しい差はみられなかったが，フェッチ命令数で比較した場合にその差が明確となる．パスの数 16 の時，PM における 73.3 命令に対し，JRS-PM では 43.4 命令と 41%の命令フェッチを削減している．本評価では，機能ユニットや物理レジスタといったハードウェア資源を理想化したが，一定のハードウェア資源を仮定した場合には，IPC においても JRS-PM の有利性が明らかになると考えている．

### 5.5.2 各ステージで処理される命令数

最も性能の高い JRS-PM において，それぞれのパイプラインステージで処理される命令数を図 5.13にまとめる．図の横軸は各パイプラインステージを示し，縦軸はそれぞれのステージで処理された平均命令数を表している．分岐結果が得られた時点で不必要となるパスが削減され，パイプラインステージが進むに従って処理される命令数が減っていく．

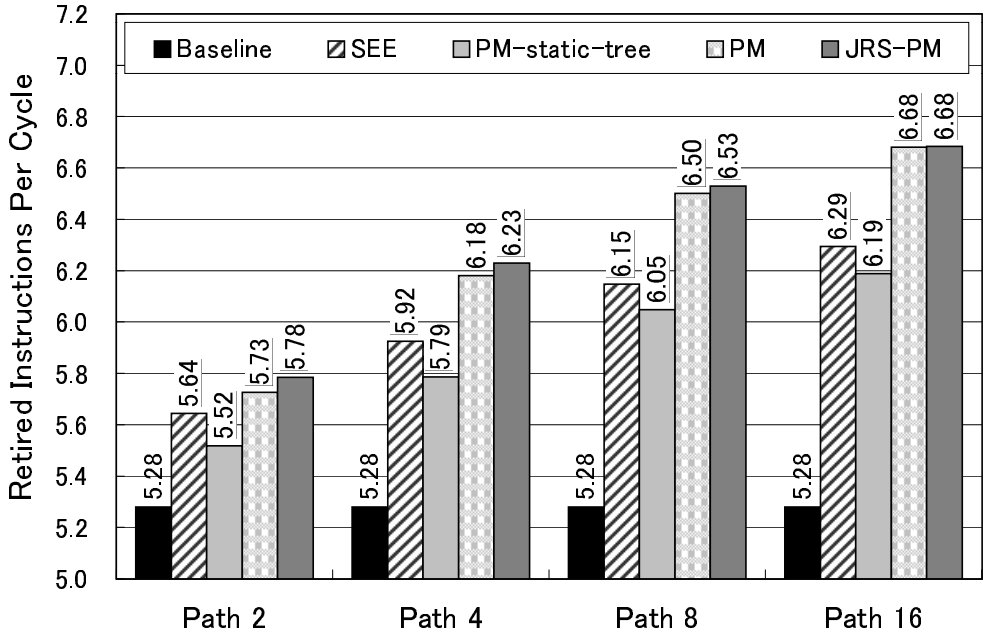



図 5.11: パス割付け戦略とパス数を変化させた時の IPC

図 5.13の結果から，プロセッサのフロントエンドでは処理される命令数が一定の割合で減少することがわかる．オペランド・フェッチと実行ステージの間ではデータの待ち合わせによる待ちサイクルが挿入されるため，実行される処理量は更に減少する．パス数が 16 の時には，フェッチした命令数 43.4 と比較し，実行した命令は 20.7 命令となった．実行機構では，最低でもこれ以上の数の実行ユニットを実装する必要がある．ただし，図 5.13に示した値は平均処理命令数であり，ピーク時には数倍の命令が処理される．ハードウェア資源を制限した場合の検討は今後の課題である．

### 5.5.3 データキャッシュのヒット率

複数パス実行によるデータキャッシュへの影響を議論する．ベースラインプロセッサ (Base) と，最も性能の高かった JRS-PM に関してパス数を変化させた場合のデータキャッシュのヒット率を表 5.3にまとめる．ここに示すヒット率は，リタイアしたロード命令を対象として計算した．表 5.3の結果より，パスの数を増やすに従ってヒット率が僅かに向上することがわかる．これは，実行に必要ないパスのメモリアクセスがプリフェッチとしての役割を果たているためと考えられる．

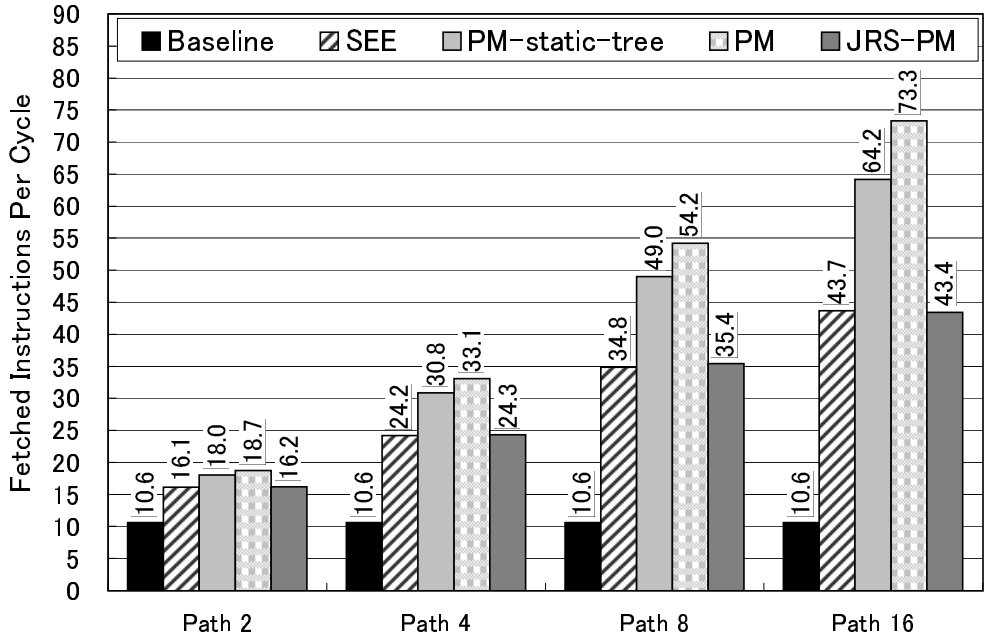



図 5.12: サイクル当たりの平均フェッチ命令数

複数パス実行はヒット率に対して良い影響を与えたが，パスの数を増やすに従ってリード・ポート数に対する要求は厳しくなっていく.(ただし，パスの数によりリタイアする命令数は変化しないので，ライト・ポート数に対する要求は深刻にならない.) JRS-PMでパスの数を 2, 4, 8, 16 に設定した場合には，サイクル当たり 2.3, 3.2, 4.3, 5.0 回のキャッシュリードが必要になるというデータを得た．平均値ではなく，ピーク時のアクセスを考えた場合にはさらに多くのリード・ポートが必要となる．ポート数の増加を目指した研究成果 [RTDA98] の利用を含め，ポート数と性能の検討は今後の課題である．

## 5.6 今後の課題

### 5.6.1 資源割付け戦略の理想化

複数パス実行は命令フェッチのトラフィックを倍増させる．しかし，命令領域は上書きされることがない，命令領域へのアクセスの局所性は非常に高い，という性質により，命令キャッシュを複製することで対処可能であると考えている．また，利用できるパスの数だけ命令キャッシュのポート数を確保できない場合には，文献 [HS96] で議論されているよう

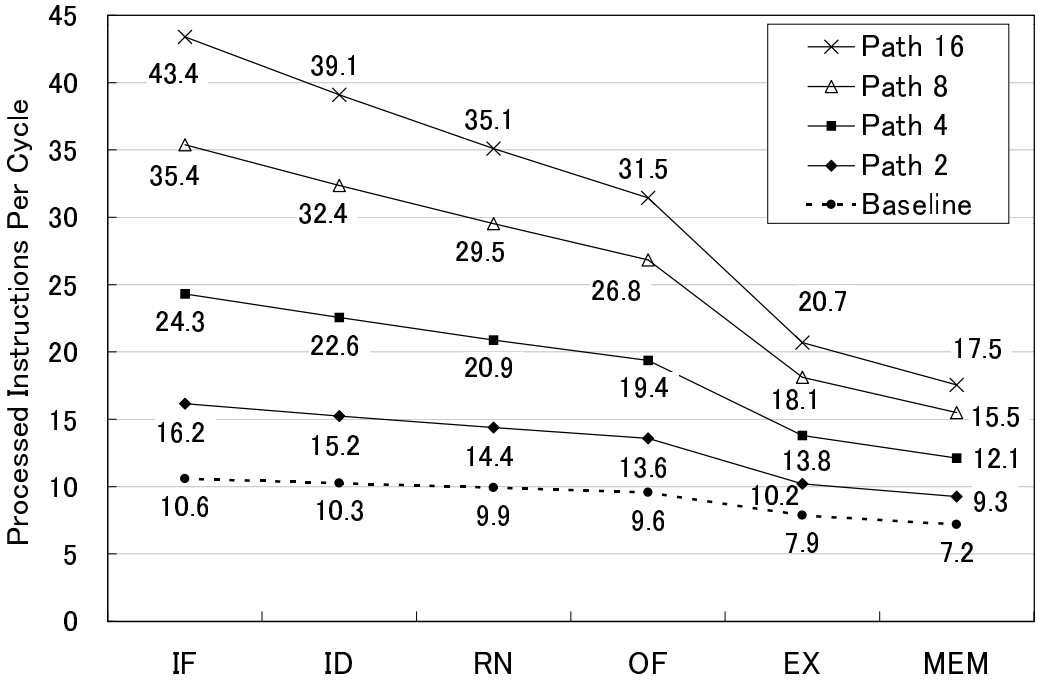



図 5.13: 各パイプラインステージの平均処理命令数

に，フェッチ資源を時分割してパスに割り付ける，パスの実行確率によってそれぞれのパスからフェッチする命令数を制御するといった手法の利用が考えられる．本章の評価で理想化した資源割付け戦略に関する検討は今後の課題である．

### 5.6.2 パス割付け戦略の検討

本章では，分岐予測ミスの削減を目指したパス割付け戦略を検討した．ここでは，別の方針に従ったパス割付け戦略の可能性を議論する．

文献 [森敦 97] で定義されている分岐予測の直行性という概念を用いれば，単体の予測性能が高く，かつ高い直行性をもつ複数の分岐予測を用いることで精度の高いハイブリッド分岐予測を構成できる．ただし，この場合，複数の分岐予測の結果から，ハイブリッドとしての予測結果を選択するセレクタ部の選択ミスが問題となる．複数の分岐予測の結果が異なる分岐命令で，パスを複製するパス割付け戦略により，ハイブリッド分岐予測の選択ミスを削減できる．

| Program | Base | Path 2 | Path 4 | Path 8 | Path 16 |
|---|---|---|---|---|---|
| go | 99.66 | 99.71 | 99.77 | 99.82 | 99.84 |
| m88ksim | 99.30 | 99.40 | 99.50 | 99.60 | 99.60 |
| gcc | 98.50 | 98.68 | 98.88 | 99.06 | 99.15 |
| compress | 98.04 | 98.13 | 98.03 | 98.05 | 98.07 |
| li | 98.13 | 98.42 | 98.64 | 98.82 | 98.90 |
| ijpeg | 99.73 | 99.75 | 99.77 | 99.78 | 99.79 |
| perl | 98.00 | 98.25 | 98.57 | 98.85 | 99.03 |
| vortex | 98.95 | 98.91 | 98.99 | 99.06 | 99.91 |
| 平均 | 98.78 | 98.90 | 99.02 | 99.13 | 99.28 |

表 5.3: JRS-PMにおけるデータキャッシュのヒット率 (%)

値予測 [WF97] などの予測機構を用いた場合には，予測ミスが深刻な性能低下を引き起こす場合がある．複数パス実行により予測する/しないという 2 つのパスを実行することで予測ミスペナルティを回避できる．また，値予測が複数の予測結果を出す場合においてそれぞれの予測値を用いてパスを複製することも考えられる．これらの戦略を用いたパス割付け戦略の検討は今後の課題である．

### 5.6.3 複数の予測によるカスケード接続

値予測や，曖昧なメモリ依存関係を解消するメモリ依存予測 [CE98] によるプロセッサ性能の向上が確認されている．これらの依存関係を解消した場合に，分岐ペナルティ削減の重要性がますます増加すると考えている．また，文献 [佐藤 99] では，値予測とメモリアドレス予測の協調型予測による有効性を議論している．パス割付け戦略に関しても複数予測の協調による性能向上の可能性がある．これらの検討は今後の課題である．

## 5.7 本章のまとめ

分岐成立と不成立の両方のパスにおいて投機的に処理を進めておくことで，分岐予測ミスのペナルティを削減する複数パス実行の可能性を検討した．分岐予測ミスを確率的に削減して分岐予測ミスと IPC の関係を調査した結果，分岐予測ミスを削減することで最大 45%の IPC 向上が可能となることを示した．また，予測ミスの削減率と予測ミスペナルティ

の間にずれが発生し，複数パス実行による性能向上を評価する場合には，予測ミスペナルティの削減率またはIPCを用いて評価することの必要性を述べた．

複数パス実行で重要な役割を果たすパス割付け戦略としてSelective Eager Executionと確率モデルを比較検討した上で，各手法の利点を組み合わせるJRS-PM方式を提案し，その有効性を確認した．

分岐ミスによりフラッシュされる本来必要ない命令の影響を考慮した実行ベースのシミュレータを開発し，これを用いて，命令レベル並列性，各パイプライン・ステージで処理される命令トラフィック，データキャッシュのヒット率を測定した．測定結果から，確率モデル，Selective Eager Executionと比較して，JRS-PMを用いた場合の性能が最も高く，パスの数を16とした場合に59%の分岐ミスペナルティを削減できることを示した．この時，サイクル当たりのフェッチ命令数は43，実行命令数は21というデータを得た．複数パス実行によりデータキャッシュのヒット率が僅かに向上するが，リード・ポート数に対する要求が増大することを示した．

複数パス実行を現実的な手法として確立するためには，資源割付け戦略の検討を始めとして，実行機構の構成や，データキャッシュのポート数の問題などの課題をクリアする必要がある．

# 第 6 章

# メモリ依存予測による曖昧なメモリ依存関係の解消

ロード命令とストア命令が参照するメモリ・アドレスは，パイプライン・ステージの後半（通常は実行ステージ）で確定する．このため，レジスタリネーミングのような機構を用いて，命令が発行される前にロード命令とストア命令間のデータ依存関係を解析することはできない．アウトオブオーダ実行をおこなうプロセッサでは，ロード・ストア間のデータ依存関係を検出するために，ロード命令と，それに先行する全てのストア命令のアドレスが計算されているという条件を満たさなければならず，先行するストア命令の中にメモリ参照アドレスが確定していないストア命令が存在する場合，プロセッサはロード命令の実行を遅らせる必要がある．このように，ストア命令のメモリ参照アドレスが計算されていない場合に発生する依存関係を**曖昧なメモリ依存関係 (ambiguous memory depenedence)** と呼ぶ．

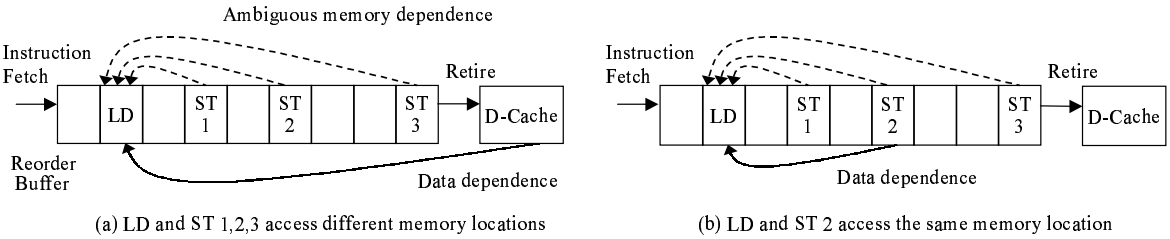



図 6.1: 曖昧なメモリ参照により生じる依存関係

曖昧なメモリ依存関係の例を図 6.1に示す．この例では，ロード命令に先行してフェッチ

された 3 つのストア命令 (ST1, ST2, ST3) がリオーダ・バッファ中に存在しているとする．この時，ロード命令は，先行する全てのストア命令に対して曖昧なメモリ依存関係を持つ．図 6.1では，この依存関係を破線矢印で示している．

ロード・ストア命令のメモリ参照アドレスが計算されて曖昧なメモリ依存関係が解消できれば，ロード・ストア間のデータ依存関係を解析できる．図 6.1(a) は，先行するストア命令とロード命令のメモリ参照アドレスが一致しなかった場合の例で，この時，ロード命令が必要とするデータはデータキャッシュから供給されることになる．図 6.1(b) は，ロード命令と ST2 のメモリ参照アドレスが一致した場合の例で，この時，ロードされるデータには ST2 がストアした値を用いなければならない．このように，曖昧なメモリ依存関係を解消した後には，実線矢印で示したデータ依存関係を満たすように処理を進めればよい．

本章では，曖昧なメモリ依存関係を解消することの重要性を検証するとともに，メモリ参照アドレスが計算される前にデータ依存関係の有無を予測することで曖昧なメモリ依存関係を解消するメモリ依存予測 (**memory dependence prediction**) を議論する．

## 6.1 関連研究

曖昧なメモリ依存関係を解消する手法としてアドレス予測の利用が検討されている [Sat97]．これは，ストア命令のメモリ参照アドレスを予測することで，曖昧なメモリ参照を明確なメモリ参照に変える手法である．文献 [Sat97] では，ストア命令の前回の実行におけるメモリ参照アドレスと，過去 2 回のメモリ参照アドレスの差分から，今回のメモリ参照アドレスを予測する動的なアドレス予測 (Reference Prediction Table を用いたアドレス予測) を評価しており，SPECint92 を用いた評価において，プログラムにより 84%〜93%の精度でストア命令のメモリ参照アドレスを正しく予測できることを示している．

しかし，大規模な投機処理を想定した場合に 93%の予測精度は十分ではない．大規模な投機処理を考えた場合，数十個のストア命令が実行中となることは珍しくない．例えば，あるロード命令に先行して 10 個のストア命令が実行中であるとすると，それぞれのアドレス予測の成功率が 93%としても，これらの全てのアドレス予測が成功する確率は $0.93^{10} = 0.45$ に低下してしまう．64 ビットのメモリ参照アドレスの予測（64 ビット出力の予測）が必要となる点がアドレス予測の精度向上を困難なものにしている．一方，曖昧なメモリ依存関係を解消するためには 64 ビット出力のアドレス予測が必要という訳ではない．例えば，図 6.1(a) に示した様に，実行中のストア命令に対してデータ依存関係を持たないという 1 ビット出力の予測により曖昧なメモリ依存関係を解消できる場合がある．本章では，アドレス

予測を利用することなく，曖昧なメモリ依存関係を解消するメモリ依存予測を検討する．

### 6.1.1 Load/Store Wait Tables

Alpha21264[Kes99] で用いられている曖昧なメモリ依存関係の解消手法を図 6.2の例を用いて説明する．同じアドレスを参照するロード命令がストア命令に続いて実行されるとする．これらの命令を最初に実行する際には，できる限り早いタイミングでロード命令を実行する．この時，ストア命令に先行してロード命令が実行され，ロード命令は間違った値をデータキャッシュからロードしたとする．その後，ロード命令が正しくない値を利用した事を検出し，プロセッサはロード命令以降の全ての命令をフラッシュすることで予測ミスから回復する．この時，これらの命令の次回の実行において再び予測ミスが発生することを回避するために，ロード命令のプログラムカウンタをインデックスとして load wait table のビットをセットする．また，ストア命令のプログラムカウンタをインデックスとして store wait table のビットをセットする．

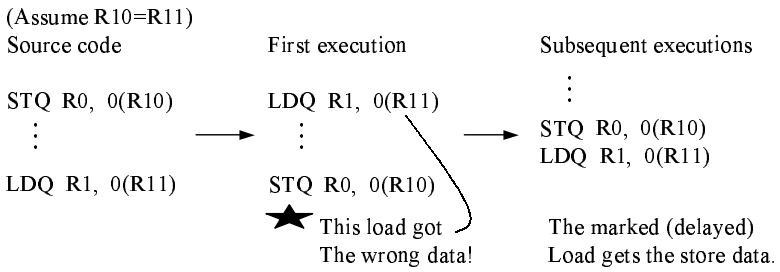



図 6.2: An Example of the Alpha21264 Memory Dependence Prediction

次回の実行からは，ロード命令をフェッチする際に load wait table を参照することで，以前に予測ミスを引き起こしたロード命令である事を検出する．さらに，store wait table を用いて，以前に予測ミスを引き起こしたストア命令が実行中であるかを検出する．過去に予測ミスを起こしたストア命令が実行中の場合に，過去に予測ミスを起こしたロード命令のストア命令の追い越しを禁止することで，予測ミスを回避する．ロード命令の不必要な待ちを削減するために，store wait table のすべてのビットは一定の時間間隔でリセットされる．

### 6.1.2 ストアセット

Alpha21264 における load/store wait tables を用いた予測では，過去に予測ミスを起こした任意のロード命令と，過去に予測ミスを起こした任意のストア命令がデータ依存関係を持つ候補となる．これは，過去に予測ミスを起こしたという 1 ビットの情報を用いて予測をおこなっており，予測ミスを引き起こしたロード命令とストア命令を結び付ける手段を用いていないためである．この点を改良するために，ストアセットという概念を用いた予測手法が提案されている．

あるロード命令の過去の実行においてデータ依存関係を持ったストア命令の集合を，当該ロード命令のストアセットと呼ぶ [CE98]．このストアセットを用いて，ロード命令を実行する際に，データ依存関係を持つ可能性のあるストア命令を予測できる．すべてのロード命令のストアセットは，プログラムの開始時に空集合で初期化されている．ロード命令を実行する際に，そのロード命令のストアセットに登録されているストア命令がプロセッサ内で実行中であるかをチェックする．ストアセットに登録されているストア命令が実行中でない場合には，先行するストアにデータ依存を持たないとして投機的にロード命令の処理を開始する．もし，ロード命令とストア命令が間違った順番で実行された場合には，ストア命令の PC をストアセットに追加する．

ストアセットの例を図 6.3に示す．図における A,B,C,D は，64 ビットのメモリ参照アドレスを表しているとする．ロード命令は複数のストア命令にデータ依存関係を持つことがある．図 6.3の例では，PC 36 のロード命令は，PC 0 と PC 12 のストアにデータ依存関係を持つ．また，複数のロードが同じストア命令にデータ依存関係を持つことがある．図 6.3 の例では，PC 28 と PC 40 のロード命令が PC 8 のストア命令にデータ依存関係を持つ．

| PC | | PC | |
|---|---|---|---|
| 0 | store C | 28 | load B, store set { PC 8 } |
| 4 | store A | 32 | load D, store set { (null) } |
| 8 | store B | 36 | load C, store set { PC 0, PC 12} |
| 12 | store C | 40 | load B, store set { PC 8 } |

図 6.3: ストアセットの例

ストアセットを用いた予測を素直に実装する場合には，予測の対象となるロード命令に先行して実行されているストア命令の PC とストアセット内の PC との比較が必要となり，これを処理するために多くの比較器からなる複雑なハードウェアが必要となる．この問題

を回避するために，文献 [CE98] ではストアセットを用いた現実的な予測機構を提案している．この構成を図 6.4に示す．SSIT は，ロード命令とストア命令で共通に利用するテーブルで，共通のタグを用いてストアセットを保持する．LFST は，ストアセット内のストア命令の中で最も最近にフェッチされたストア命令を保存しておくテーブルである．プログラムの開始時には，SSIT の全てのエントリを無効にセットしておく．この状態では，全てのロード命令は全てのストア命令にデータ依存関係を持たないと予測される．もし，ロード命令が間違った順番で実行され，ロード命令がストアセットを持っていない場合には，新しいストアセットを生成する．このストアセットを別のストアセットと区別するために，識別子 (SSID: Store Set ID) を利用する．予測ミスを引き起こした原因となるロード命令，ストア命令の PC から SSIT のインデックスを生成し，SSIT の 2 つのエントリに SSID を格納する．

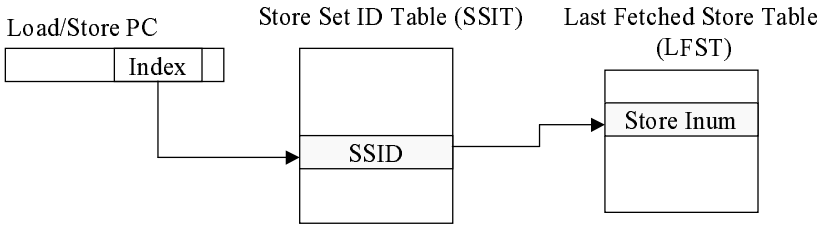



図 6.4: Implementation of Store Sets Memory Dependence Prediction

ストア命令をフェッチした際に，ストア命令の PC からインデックスを生成して SSIT にアクセスする．SSIT のエントリに SSID が格納されている場合には，当該ストア命令は過去に予測ミスを引き起こし，ストアセットの要素となっている．SSID から LFST のインデックスを作成し，LFST のエントリに当該ストア命令の識別子 (Store Inum) を格納する．当該ストア命令がリタイアした際，または，分岐予測ミスなどでフラッシュされた際に，当該ストアが作成したエントリを LFST から削除する．

ロード命令をフェッチした際に，ロード命令の PC からインデックスを生成して SSIT にアクセスする．当該ロード命令が過去に予測ミスを引き起こしストアセットを持つ場合には，SSIT のエントリに有効な SSID が格納されている．SSID から LFST のインデックスを作成し，LFST をアクセスする．ストアセットに含まれるストア命令が実行中の場合には，LFST のエントリには Store Inum が格納されている．そうでない場合には，ストアセット内のストア命令は実行中でないとして，当該ロード命令はできるだけ早いタイミングで実行を開始する．

ストアセットに含まれるストア命令が実行中の時のロード命令の発行タイミングは利用するプロセッサモデルに依存する．図 6.1(b) の場合で，LFST のエントリに ST2 を示すタグが格納されている場合を例に考える．この場合，ST1, ST2 が共に発行されてから，ロード命令を発行する必要がある．ただし，ST3 の発行を待つ必要はない．なぜなら，もし ST3 がストアセットの要素となっている場合には ST3 がフェッチされた時点で LFST のエントリは ST3 のタグで上書きされるからである．ストア命令間で発行をインオーダにおこなうプロセッサでは，ST2 が発行された時点で ST1 が発行されていることを保証できるので，ST3 の発行を待つことなくロード命令の実行が可能となる．一方，4節で定義したベースラインプロセッサのように，アウトオブオーダにストア命令を実行するプロセッサの場合には，ST2 が発行されたとしても ST1 の発行を保証できないので，先行する全てのストア命令 (ST1, ST2, ST3) が発行されたことを確認してロード命令の実行を開始する必要がある．

### 6.1.3 メモリ・バイパシング

曖昧なメモリ依存関係を解消する手法ではないが，曖昧なメモリ依存関係を解消しさらにデータ依存関係を持つストア命令とロード命令を特定することで生じる利点としてメモリ・バイパシング **(speculative memory bypassing)** という手法がある．

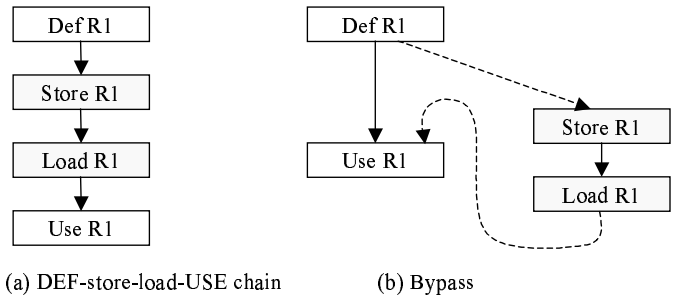



図 6.5: Speculative Memory Bypassing

ロード・ストア命令は，入力オペランドに対して演算をおこなっているわけではなく，ある命令が生成したデータを消費者となる他の命令に渡すために用いられる．曖昧なメモリ依存関係を解消するとともに，データ依存関係を持つストア命令とロード命令のペアを特定することで DEF-store-load-USE というデータ依存関係を DEF-USE という形に変えることができる．この変形の様子を図 6.5に示す．このようなメモリ依存予測を用いた依存関係の変形はメモリ・バイパシングと呼ばれ，文献 [MS97] において検討されている．

## 6.2 曖昧なメモリ依存関係の削除による性能向上

曖昧なメモリ依存関係を解消することによる性能向上の上限を得るために，曖昧なメモリ参照を 100%解消できると想定した場合の IPC を測定する．評価では，ストア命令のメモリ参照アドレスはフェッチする際に正しく予測できると仮定する．これは，あらかじめ収集しておいたストア命令の参照アドレスの履歴を用いて実現する．この変更により，ロード命令のメモリ参照アドレスが計算された時点で，データ依存関係を持つストア命令の解析が可能となり，曖昧なメモリ依存関係を排除できる．

測定結果を図 6.6に示す．図 6.6の実線 (Baseline) は 4節で定義したベースラインプロセッサのシミュレータを用いて測定した IPC であり，破線 (Memory Disambiguation) は曖昧なメモリ依存関係を解消した場合の IPC を示している．複数パス実行により条件分岐の予測ミスを削減する場合を考慮し，条件分岐の分岐予測ミスを確率的に削除して IPC を測定した．図 6.6の横軸は削減した条件分岐の分岐予測ミスの割合で，0%〜100%まで 10%単位で変化させた．分岐予測の際に，予測した結果とあらかじめ収集しておいた分岐結果とを比較し，もし，分岐予測が間違っていた場合には，乱数を用いて 0%〜100%の確率で分岐予測を正しい結果に変更した．左端の削減率 0%は通常の分岐予測を用いた設定，右端の削減率 100%は分岐予測ミスを完全に削減した設定を意味している．

図 6.6の結果より，曖昧なメモリ依存関係を排除することで，0.73〜1.37 の IPC の向上を達成できる．また，分岐予測ミスの削減率が大きい場合に IPC の向上が大きいことがわかる．図 6.6の結果にはメモリ・バイパシングによる性能向上は考慮されていない．データ依存関係を持つストア命令を特定できれば，メモリ・バイパシングを用いて更なる IPC の向上を期待できる．

## 6.3 曖昧なメモリ依存関係を解消する予測機構

曖昧なメモリ依存関係を解消する予測機構を検討する前に，ロード命令が実行中のストア命令にデータ依存関係を持つ頻度を調査する．もし，ロード命令が，実行中のストア命令にデータ依存関係をもたなければ，ロード命令は常にデータキャッシュからデータをロードすればよい．実行中のストア命令にデータ依存関係を持つロード命令の実行頻度を図 6.7にまとめる．図 6.7には，4節で定義したベースラインプロセッサにおいて，曖昧なメモリ依存関係を 100%解消し，分岐予測ミスを 50%の確率で削減した場合のデータをまとめた．このデータから，平均で 32%のロード命令が実行中のストア命令にデータ依存関係を持つ

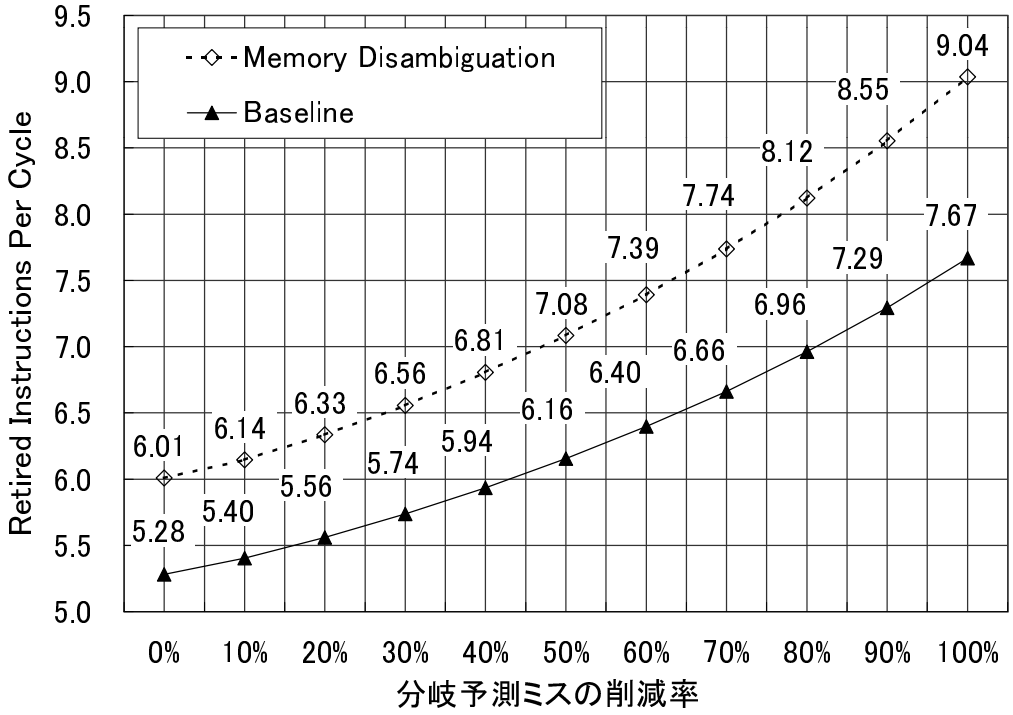



図 6.6: 曖昧なメモリ参照を排除することによる IPC の向上

ことがわかる．すなわち，常に実行中のストア命令にデータ依存関係が無いと予測し，データキャッシュにアクセスする場合には 32%の予測ミスが発生する．32%の予測ミスは許容できる範囲ではなく，曖昧なメモリ依存関係を解消するメモリ依存予測が必要となる．

実行中のストア命令にデータ依存関係を持たないロード命令を予測することを**依存存在予測**と呼ぶことにする．依存存在予測により，実行中のストア命令にデータ依存関係が無いと予測されたロード命令は，図 6.1(a) の場合のように，メモリ参照アドレスが計算された時点でデータキャッシュからデータをロードすればよく，先行するストア命令のアドレス計算を待つ必要はない．すなわち，曖昧なメモリ依存関係を解消できる．先に述べた Load/Store Wait Tables を用いた手法と，ストアセットを用いた手法は依存存在予測の一つである．依存存在予測は 1 ビット出力の予測である．

図 6.7に示した様に，依存存在予測を用いて依存関係が存在しないことを正しく予測できる割合はたかだか全ロード命令の 68%であり，残りの 32%のロード命令は実行中のストア命令にデータ依存関係を持つ．この場合には，依存存在予測のみでは曖昧なメモリ依存関係を解消できない．図 6.1(b) の場合を考えると，ロード命令が実行中のストア命令にデー

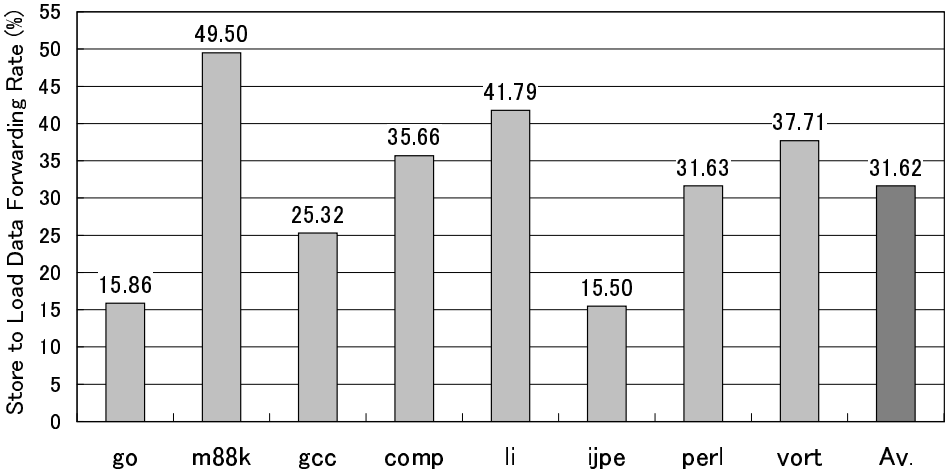



図 6.7: 実行中のストア命令にデータ依存関係を持つロード命令の実行頻度

タ依存関係を持つという情報だけでは，先行してフェッチされた全てのストア (ST1, ST2, ST3) のアドレス計算を待つ必要性が生じてしまう．もし，ST2にデータ依存関係を持つことを特定できれば，その他のストア命令のアドレス計算を待つ必要がなくなり，曖昧なメモリ依存関係を解消できる．このように，データ依存関係を持つストア命令を特定する予測をストア特定予測と呼ぶことにする．ストア特定予測により，データ依存関係のあるストア命令を予測した場合には，このストア命令のストア・データが用意できた時点で，ロード命令にデータを供給すればよい．また，ストア特定予測が有効となる場合にはメモリ・バイパシングを利用した最適化を期待できる．

### 6.3.1 ストア特定予測の提案

レジスタ数の不足が原因で生じるロード・ストア命令のように，データ依存関係をもつロード命令とストア命令が動的に変化しない場合が存在する．このような組を動的に検出し，データ依存関係を持つストア命令を特定するストア特定予測を提案する．

ストア特定予測の構成を図 6.8に示す．予測は Valid, Tag, State, StorePC という 4 つのフィールドを持つエントリから構成されるテーブルを利用する．ただし，図が複雑になることを避けるために，エントリが有効であることを示す Valid フィールドは省略した．State フィールドには，予測するロード命令の過去の実行において，連続して，同じストア命令にデータ依存関係が存在した回数を保持する．State フィールドは値 1 で初期化しておくと

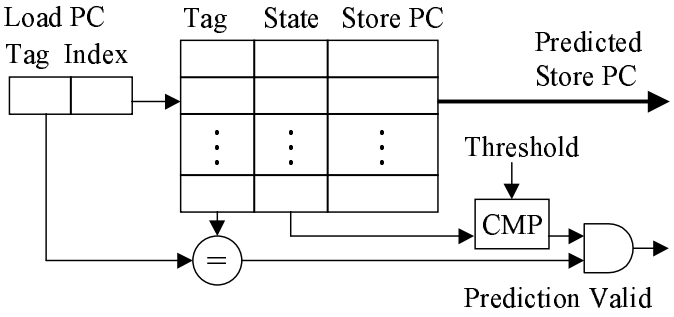



図 6.8: ストア特定予測のブロック図

する．State フィールドの値がある閾値より大きい場合に，ロード命令は StorePC フィールドに格納されているアドレスのストア命令にデータ依存関係を持つと予測する．

テーブルの更新は，ロード命令がリタイアする際におこなう．ロードしたデータがストア命令から供給された場合には，ストア命令の PC を StorePC フィールドに保存する．ただし，同一 PC のストア命令が複数実行されている場合には，ロード命令に最も近いストア命令にデータ依存関係がある場合のみテーブルに保存する．以降，同じロード命令がリタイアした際，StorePC フィールドに保存されている PC のストア命令に再度データ依存関係を持った場合にカウンタの値をインクリメントする．

予測ミスが発生した場合には，予測をおこなわないことを示す特別な値 0 を State フィールドにセットする．予備評価により，一度予測ミスを引き起こした命令は，再びミスを引き起こす可能性が高いことが確認されている．このため，予測ミスの削減を目指し，一度予測ミスを起こしたロード命令の以降の予測を禁止するために，State フィールドの値が 0 であるエントリは値 0 から変化しないとする．

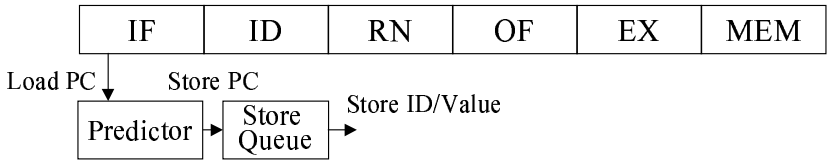



図 6.9: ロード命令の命令パイプラインとストア特定予測

ロード命令の命令パイプラインにおいて，ストア特定予測を利用するタイミングを図 6.9

に示す．命令フェッチ・ステージでロード命令の PC を用いてストア特定予測を利用する．予測が有効な場合にはデータ依存関係を持つストア命令の PC が出力される．その後，命令デコード・ステージにおいて，ストア命令 PC を用いてストアキューを検索することで，該当するストア命令が先行して実行中であるかを判定する．もし，該当するストア命令が実行中でない場合には，ロード命令は実行中の全てのストア命令にデータ依存関係がないものとして処理を進める．一方，該当するストア命令が実行中である場合には，そのストア命令とロード命令を何らかのタグを用いて結びつける．この場合，ロード命令は結び付けられたストア命令の実行が終了したことを検出して処理を進めることができる．ただし，ストアキューを検索した時点でデータ依存関係を持つストア命令のストアデータが用意できている場合には，タグを用いて関連付ける必要はなく，ストアデータをロード命令にフォワーディングする．

## 6.4 評価環境

### 6.4.1 ストアキュー

アウトオブオーダ実行をおこなうプロセッサは，キャッシュの更新をインオーダに処理するためにストアキューを利用する．このような目的でストアキューを実装した場合には，キャッシュにデータを書き出したエントリはストアキューから開放してかまわない．一方，キャッシュにデータを書きこんだ後にも，エントリを開放せずストアキューからデータを供給するように設計することもできる．この場合，新しくエントリが必要となった時点で，最も古く割り当てられたエントリを開放する．エントリが開放されるまで，ストアキューは各エントリのデータを供給できるバッファとなる．ベースラインプロセッサではストアキューを両方の目的で利用する．

ストアキューのエントリ数を 16 から 512 まで変化させて，ストアキューからデータを受け取るロード命令の実行頻度を測定した．スカラプロセッサ環境を用いて測定したため，$n$ エントリのストアキューからデータを受け取る割合は，ロードするデータが先行する $n$ 個のストア命令において生成された値である確率と解釈できる．測定結果を図 6.10に示す．この結果から，先行する 256 個のストア命令にデータ依存関係を持つロード命令の実行頻度は，平均で 51%となることがわかる．以降の評価では，256 エントリのストアキューを利用する．ベースラインプロセッサではリオーダバッファのエントリ数を 256 エントリに設定しているので，ストアキューのエントリを 256 に設定することで，ストアキューのエ

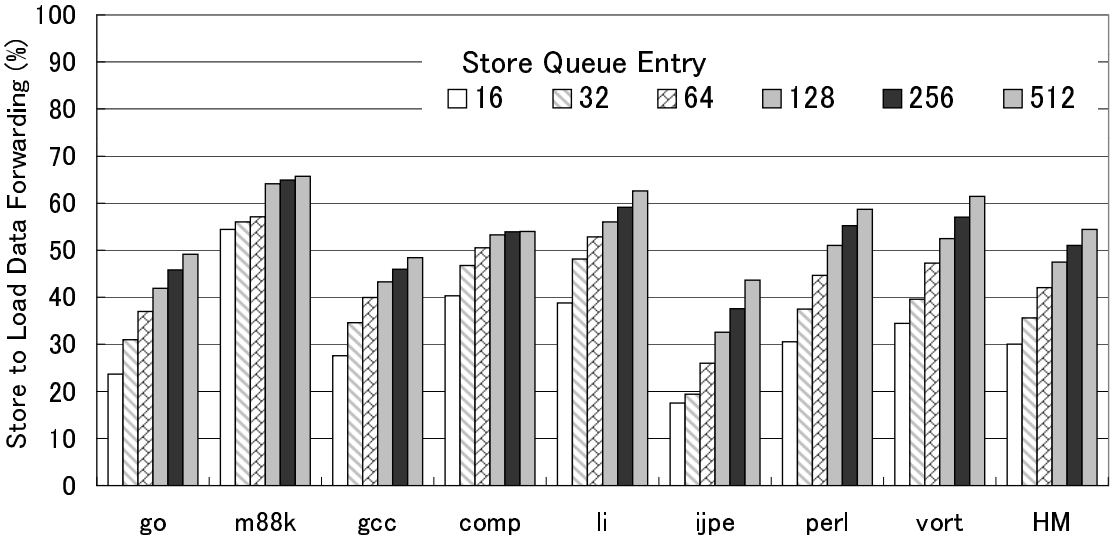



図 6.10: ストアキューからデータを受け取るロード命令の実行頻度

ントリ不足により生じるプロセッサのストールを排除できる．

### 6.4.2 予測ミスの検出機構

予測ミスの検出には次の 2 つの方法が考えられる．一つは，ロード命令とストア命令のメモリ・アドレスを比較して，データ依存関係の有無を検出する方法で，これをアドレス比較によるミス検出と呼ぶことにする．別の方法は，予測により得られたロード・データと正しいデータを比較するもので，これをデータ値比較によるミス検出と呼ぶことにする．予測を用いて曖昧なメモリ依存関係を解消した場合においても，ロード命令が参照するメモリ・アドレスは正しい値が計算されている．このため，ロード命令のリタイア時にデータキャッシュにアクセスすることで正しい値をロードできる．この値と予測値を比較することで予測ミスを検出できる．

それぞれの検出方法は利点と欠点を持つ．次に述べる利点により，後者の，データ値比較によるミス検出を用いて評価をおこなう．

データ値比較でミスを検出する場合には，それぞれのロード命令で必要となる比較は，正しいデータ値と投機的に利用したデータ値の比較 1 回で済む．一方，アドレス比較によるミス検出では，ロード命令のメモリ参照アドレスと，ロード命令に先行されてフェッチさ

れ実行中の $n$ 個のストア命令のメモリ参照アドレスに対して $n$ 回の比較が必要となる．サイクル当たり複数のロード命令が処理されることを考えると，アドレス比較によるミス検出では多数の比較器を実装する必要がある．この点から，データ値比較による検出が有利となる．

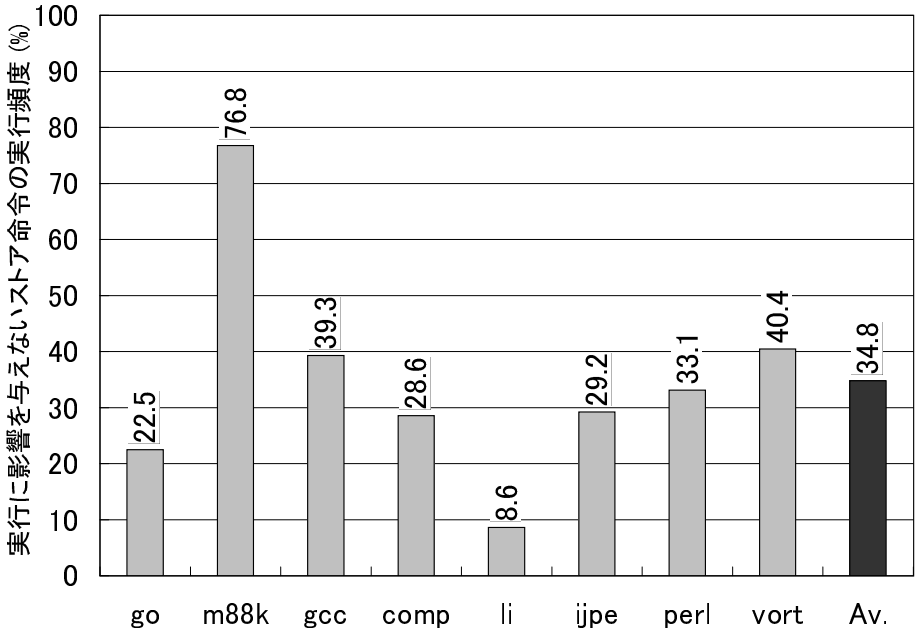



図 6.11: 実行に影響を及ぼさないストア命令の実行頻度

ストア命令の中には，ストア・データとストアする直前のメモリにおけるデータが同じ値で，その実行がプロセッサの状態に影響を及ぼさないストア命令 (no-effect store) が存在する．このようなストア命令にデータ依存関係を持つロード命令は，データ依存関係を持つストア命令からデータを受け取っても，データキャッシュからデータを受け取ってもよい．アドレス比較によるミス検出では，このような場合においても正しい順序で実行されることを強要するが，データ値比較によるミス検出を利用した場合には，ロードした値が正しい値である限り実行を継続できる．つまり，実行中のストア命令にデータ依存関係があると予測しても，データ依存関係がないと予測しても予測成功となる．実行に影響を及ぼさないストア命令の実行頻度を測定した結果を図 6.11に示すが，実行された全ストア命令の 34%が実行に影響を与えないストア命令であることがわかる．データ値比較によるミス検出を用いて，これらのストア命令にデータ依存関係を持つロード命令の予測ミスを

削減できる利点は大きい.

### 6.4.3 予測ミスからの回復機構

予測ミスからの回復に関しても幾つかの方式が考えられる. 予測ミスを起こしたロード命令の後続命令をフラッシュし, 再度, 命令フェッチから再実行することで予測ミスから回復できる. この方式は, 分岐予測で利用されているハードウェアを利用できるため, 回復のために投入するハードウェア量を抑えることができる. 一方, 間違った値を利用した可能性のある命令を全て再実行するために, 予測ミスのペナルティは大きいものとなる. 特に, 予測ミスの検出を命令のリタイア時におこない, 後続命令のフラッシュを用いて予測ミスから回復する場合には, フラッシュすべき命令はプロセッサ内の全命令となる. リオーダバッファのエントリ数 256 で, サイクル当たり 16 命令をフェッチするベースラインプロセッサでは, 最悪の場合で, プロセッサのバックエンドに存在する 255 命令と, プロセッサのフロントエンドの 4 つのステージに存在する命令の和 $255 + (16 \times 4) = 319$ 命令をフラッシュしなければならない.

別の選択として, 予測ミスを起こしたロード命令にデータ依存関係を持つ命令のみを再実行する手法 [佐藤 98] が考えられる. ただし, 命令を発行した後も命令ウィンドウのエントリを開放できない, データ依存関係を持つ命令の検出のためのハードウェアが必要となるという欠点がある.

実装の容易さから, 前者の後続命令のフラッシュによる回復を用いて評価をおこなう. 再実行による回復を用いた場合の性能は今後の課題である.

## 6.5 評価結果

4節で定義したベースラインプロセッサのシミュレータを用いて曖昧なメモリ依存関係を解消するメモリ依存予測を評価する. 6.5.1節から 6.5.3節の評価では, 確率的に分岐予測ミスの 50%を削減してメモリ依存予測による性能向上を評価する. 分岐予測ミスの削減率を変化させた場合の性能向上率は 6.5.4節で議論する.

### 6.5.1 依存存在予測

依存存在予測として Load/Store Wait Tables とストアセットを用いた予測機構を評価する. それぞれの予測はロード命令のリネーミング・ステージにおこなうとした. テーブル

エントリの競合による性能低下を抑えるために，予測に用いるテーブル (Load Wait Table, Store Wait Table, SSIT, LFST) のエントリ数を 64K エントリと非常に大きく設定した．

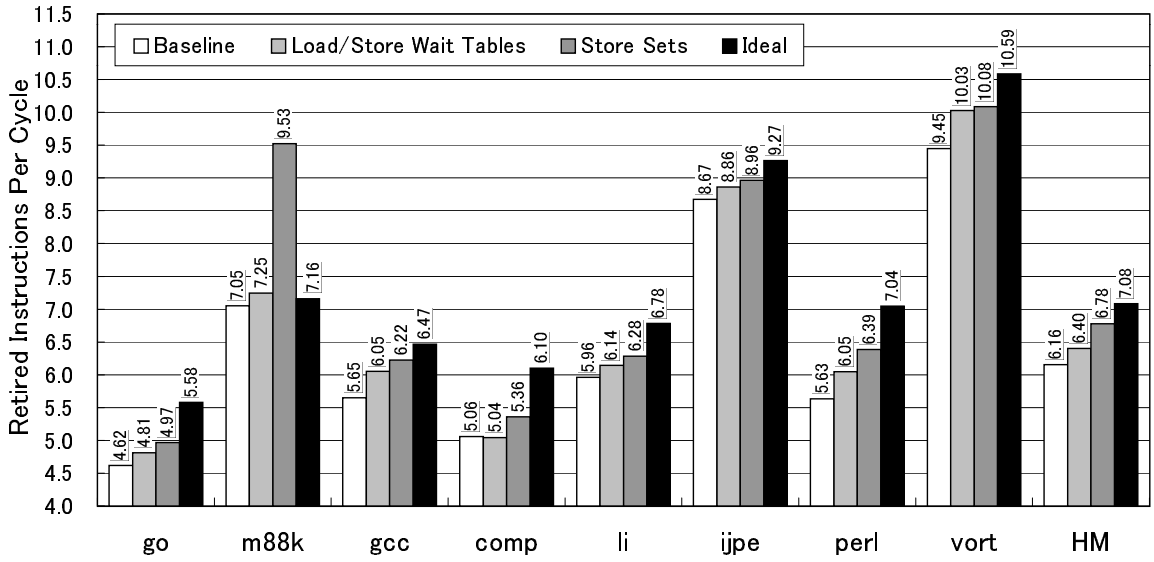



図 6.12: 依存存在予測 (Load/Store Wait Tables, Store Sets) による IPC の変化

依存存在予測による IPC の変化を図 6.12に示す．Load/Store Wait Tables より，ストアセットの方が良い結果を示した．m88ksim の場合には，ストアセットを利用することで，曖昧なメモリ依存関係を完全に解消した場合 (Ideal) より高い IPC を達成している．図 6.11に示したように，m88ksim では実行に影響を与えないストア命令の実行頻度が 77%と非常に高い．データ値比較による予測ミス検出を用いることで，本来満たすべきデータ依存関係を解消できることが，この IPC 向上につながっていると考えられる．ベースラインの IPC 6.16 と比較して，ストアセットを用いた場合には 0.62 高い IPC 6.78 を達成する．

Load/Store Wait Tables, ストアセットにおいて，全ロード命令に対して実行中のストア命令にデータ依存関係がないと正しく予測できた割合（予測精度）と予測ミスの回数を表 6.1の 2 列から 5 列にまとめる．ストアセットを用いた場合には，平均で 66%のロード命令について実行中のストア命令にデータ依存関係がないと正しく予測できることがわかる．ストアセットを用いた場合の予測ミスはプログラム当たり 13247 回発生しており，Load/Store Wait Tables の 5.6 倍となっている．

| Program | Wait Tables | | ストアセット | | ストア特定予測 | |
|---|---|---|---|---|---|---|
| | Accuracy | Miss | Accuracy | Miss | Accuracy | Miss |
| go | 50.4 % | 3562 | 76.5 % | 11777 | 19.7 % | 264 |
| m88ksim | 56.4 % | 615 | 67.6 % | 993 | 51.4 % | 64 |
| gcc | 48.1 % | 7745 | 67.5 % | 17058 | 17.1 % | 1236 |
| compress | 34.5 % | 223 | 57.4 % | 38591 | 25.3 % | 47 |
| li | 36.2 % | 640 | 49.7 % | 18164 | 31.9 % | 64 |
| ijpeg | 61.1 % | 1395 | 82.4 % | 4154 | 22.0 % | 94 |
| perl | 50.2 % | 1105 | 65.6 % | 7975 | 30.8 % | 120 |
| vortex | 48.2 % | 3586 | 57.7 % | 7267 | 34.3 % | 300 |
| Average | 48.1 % | 2358 | 65.6 % | 13247 | 29.0 % | 273 |

表 6.1: 依存存在予測とストア特定予測の予測精度と予測ミスの回数

### 6.5.2 ストア特定予測

ストア特定予測で用いるテーブルはダイレクトマップ構成とし，エントリを 64K と大きく設定して測定する．ストア特定予測が有効となる閾値を 8 に設定する．

ストア特定予測による予測精度と予測ミスの回数を表 6.1の 6 列と 7 列にまとめる．全ロード命令の 29%のロード命令について，ストア特定予測を利用して正しくデータ依存関係を持つストア命令を特定できることがわかる．予測ミスの回数は，プログラム当たり 273 回であり，依存存在予測と比較して非常に少ないことがわかる．

ストア特定予測を利用した場合の IPC を図 6.13に示す．ベースラインプロセッサと比較して，ストア特定予測を利用することで平均 0.21 の IPC 向上を確認した．

### 6.5.3 依存存在予測とストア特定予測のハイブリッド

Load/Store Wait Tables とストアセットを用いた手法の比較より，依存存在予測としてはストアセットを用いた手法が高い性能を示すことが判明した．また，ストア特定予測の評価より，実行中のストア命令にデータ依存関係がある場合に関しても，データ依存関係のあるストア命令を特定できることを示した．

依存存在予測とストア特定予測はそれぞれの予測が有効となる場合に曖昧なメモリ依存関係を解消できる．また，2 つの予測は異なった領域を予測し，それぞれの予測できない領域を補間し合う関係にある．そこで，ストアセットを用いた依存存在予測と，ストア特

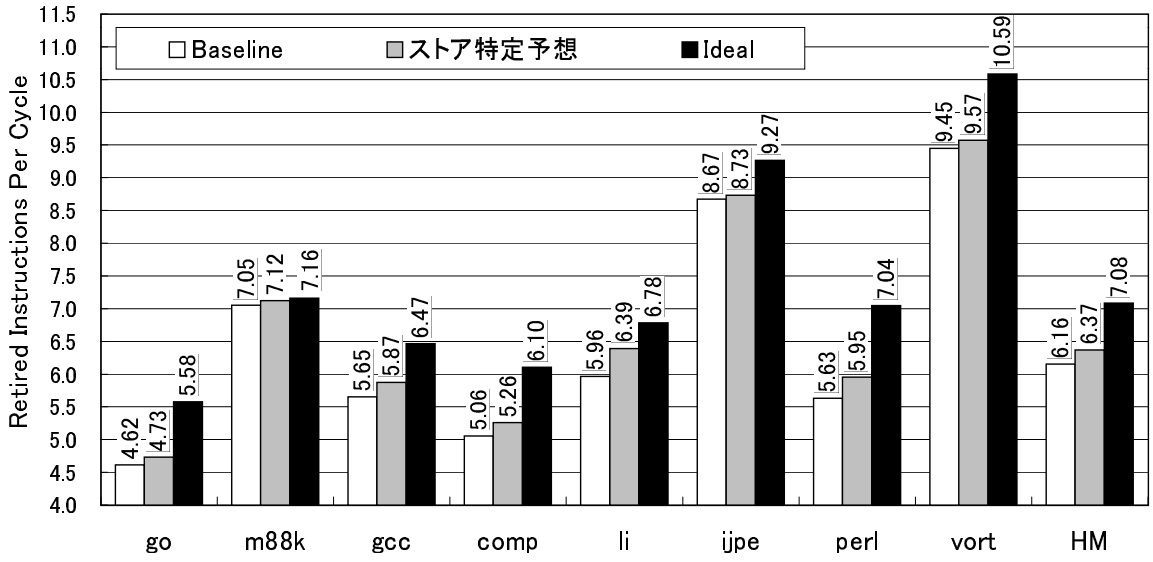



図 6.13: ストア特定予測による IPC の変化

定予測のハイブリッド予測を用いて更なる IPC の向上を目指す．

2 つの予測が共に有効となった場合には，どちらかの予測を優先し，優先度の低い予測結果を無効とする．この優先度付けにより 2 種類のハイブリッドが考えられる．

優先度を変えた 2 種類のハイブリッド予測を用いた場合の IPC を図 6.14にまとめた．それぞれのプログラムの結果には，左から，ベースラインプロセッサ，ストアセット，ストアセットを優先したハイブリッド，ストア特定予測を優先したハイブリッドという 4 つのデータを示した．図 6.14の結果から，ストア特定予測の優先度を高くした構成がより高い IPC をもたらすことがわかる．この時，ストアセットを利用した場合の IPC 6.78 と比較すると，ストアセットとストア特定予測のハイブリッドにより 0.27 の IPC 向上を確認した．

表 6.2にストアセットとストア特定予測のハイブリッドの予測精度と予測ミスの回数をまとめる．表の 2 列から 5 列にはストアセットを優先したハイブリッドの結果，6 列から 9 列にはストア特定予測を優先した場合の結果をまとめた．この結果から，優先度の違いにより，正しく予測できる割合はほとんど変化しないが，予測ミスの回数が変化することがわかる．ストア特定予測を優先した設定では，ストアセットを優先した場合と比較して，予測ミスの回数が約 30%に低下しており，これが性能向上につながっていると考えられる．ストア特定予測を優先したハイブリッド予測における予測精度は $59.2 + 26.4 = 85.6\%$ であり，これを用いて曖昧なメモリ依存関係の 86%を排除できることが明らかとなった．

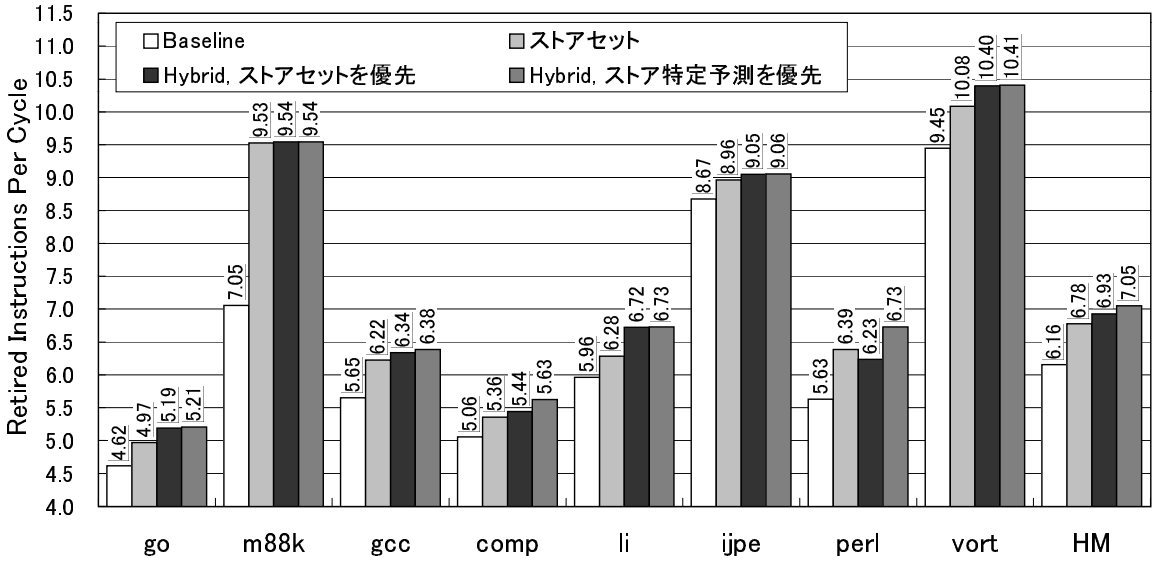



図 6.14: 依存存在予測とストア特定予測のハイブリッドによる IPC の向上

| Program | ストアセット優先 | | | | ストア特定予測優先 | | | |
|---|---|---|---|---|---|---|---|---|
| | ストアセット | | ストア特定予測 | | ストアセット | | ストア特定予測 | |
| | Accuracy | Miss | Accuracy | Miss | Accuracy | Miss | Accuracy | Miss |
| go | 70.3 % | 11606 | 9.6 % | 93 | 68.0 % | 10301 | 19.6 % | 292 |
| m88ksim | 67.7 % | 1630 | 27.9 % | 21 | 62.4 % | 963 | 33.2 % | 50 |
| gcc | 66.6 % | 25799 | 12.4 % | 869 | 63.4 % | 15384 | 16.7 % | 1348 |
| compress | 58.1 % | 66030 | 17.2 % | 25 | 49.6 % | 32411 | 24.2 % | 43 |
| li | 48.8 % | 12684 | 29.9 % | 34 | 47.9 % | 10258 | 31.7 % | 68 |
| ijpeg | 82.5 % | 4414 | 8.5 % | 54 | 69.3 % | 4155 | 21.6 % | 97 |
| perl | 65.8 % | 136560 | 23.5 % | 40 | 58.4 % | 5376 | 30.2 % | 96 |
| vortex | 61.5 % | 7257 | 27.8 % | 136 | 55.1 % | 6320 | 34.5 % | 289 |
| Average | 65.1 % | 33247 | 19.6 % | 159 | 59.2 % | 10646 | 26.4 % | 285 |

表 6.2: ストアセットとストア特定予測のハイブリッドの予測精度と予測ミスの回数

### 6.5.4 分岐予測ミスの削減率と曖昧なメモリ依存関係

これまでの評価では，確率的に分岐予測ミスの 50%を削減した環境において，メモリ依存予測の性能を評価してきた．本節では，分岐予測ミスの削減率を 0%から 100%まで変化させ，ストア特定予測を優先したストアセットとストア特定予測のハイブリッド予測の性能を評価する．

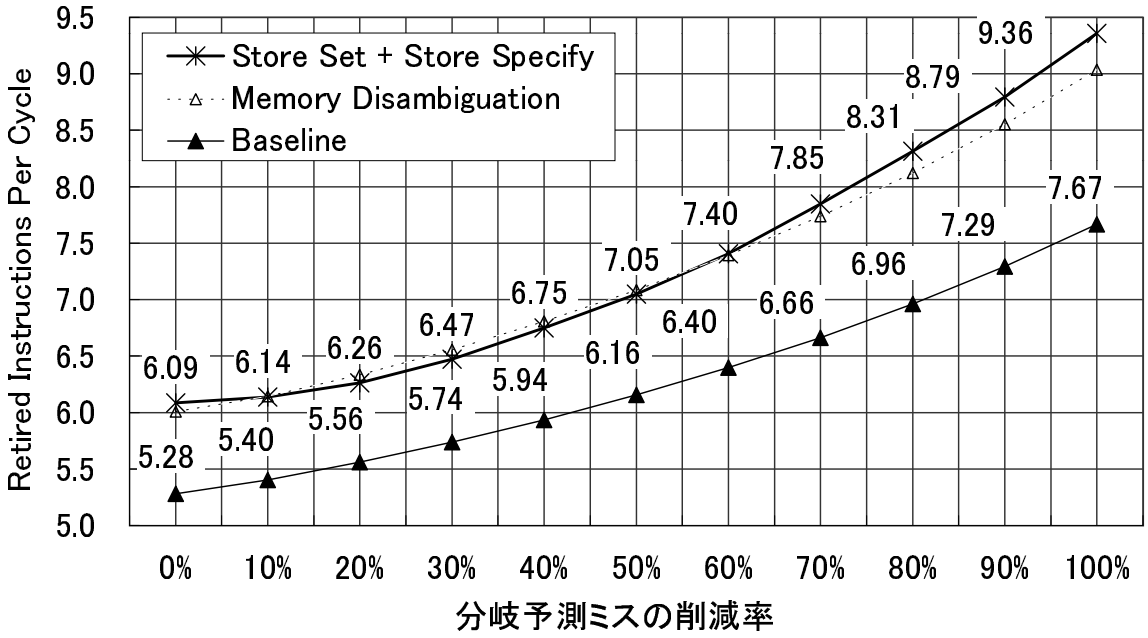



図 6.15: 分岐ミスを削除した場合のメモリ依存関係予測による IPC の向上

測定結果を図 6.15に示す．図 6.15の横軸は条件分岐の分岐予測ミスの削減率で，左端の削減率 0%はベースラインプロセッサ，右端しの 100%は条件分岐の分岐予測ミスを完全に削減した環境を意味している．今回用いたハイブリッドのメモリ依存予測 (Store Set + Store Specify) では，値予測による効果と，メモリ．バイパシングの効果を利用しているため，曖昧なメモリ依存関係を 100%解消した場合の性能とは単純に比較できないが，比較のために，図 6.15では破線として，曖昧なメモリ依存関係を 100%削減した場合の性能 (Memory Disambiguation) を加えてある．ハイブリッドのメモリ依存予測とベースラインプロセッサ (Baseline) との比較より，分岐予測ミスの削減率によって 0.7 から 1.7 の IPC 向上を達成できることがわかる．分岐予測ミスの削減率を向上させた場合に，ハイブリッドのメモリ依存予測による著しい IPC の向上を確認した．

## 6.6 今後の課題

### 6.6.1 データ値比較によるミス検出の欠点

アドレス比較のミス検出と比較して，データ値比較によるミス検出の利点を 6.4.2節で議論した．しかしながら，データ値比較によるミス検出には，ロード命令のリタイア時にデータキャッシュを参照する必要があり，データキャッシュのポート数に対する負荷を増加させるという欠点がある．

高い IPC を利用するプロセッサや，複数パス実行プロセッサではサイクル当たりに多くのロード命令を処理する必要があり，データキャッシュのポート数の増加は深刻な問題となっている．このような状況では，メモリ依存予測によるデータキャッシュ利用の増加は避けなければならない．

データ値比較のよるミス検出によるデータキャッシュ利用の増加は，リタイアしたストア命令のストア・データを格納する専用のキャッシュを用いることで解決できる．このキャッシュには，ストアキューのエントリ数と等しい数の最近リタイアしたストア命令のストア・データを格納する．このキャッシュはメモリ参照アドレスを用いて参照し，同じメモリ参照アドレスにアクセスするストア命令が多数リタイアした場合には，最も最近リタイアしたストア命令のデータで上書きする．ロード命令がリタイアする時点で，当該ロード命令のメモリ参照アドレスを用いて，このキャッシュを参照する．もし，キャッシュが有効なエントリを持たない場合には，先行して実行されたストア命令により当該ロード命令のロードすべきデータが更新されていないことを保証できる．この場合には，先行するストアを追い越して実行した当該ロードは正しい値を利用したことになる．一方，キャッシュが有効なエントリを持つ場合には，このキャッシュの内容が当該メモリ参照アドレスの最新のデータとなる．この場合は，格納されているストア・データと当該ロード命令が利用した予測値を比較することで，予測ミスを検出できる．

以上のキャッシュを追加することで，データ値比較によるミス検出の欠点であるデータキャッシュ利用の増加を解消できると考えている．具体的な機構の提案と評価は今後の課題である．

## 6.7 本章のまとめ

本章では，曖昧なメモリ依存関係を解消するメモリ依存予測を議論した．

まず，曖昧なメモリ依存関係がプロセッサ性能に与える影響を測定し，曖昧なメモリ依存関係を排除することで，最大 18%の性能向上を達成できることを示した．

これまでに提案されているストアセットを用いたメモリ依存予測は，実行中のストア命令にデータ依存関係を持たないロード命令を予測する．しかし，32 %のロード命令は実行中のストア命令にデータ依存関係を持ち，ストアセットだけでは曖昧なメモリ依存関係の 68 %しか解消することはできない．このため，ロード命令が実行中のストア命令にデータ依存関係を持つ場合に，複数のストア命令から，データ依存関係を持つストア命令を特定するストア特定予測を提案した．

シミュレーションによる評価より，以下の結果を得た．ストアセットとストア特定予測のハイブリッドでは，2 つの予測が同時に有効になったときの優先度により 2 つの構成が考えられる．これらの構成を評価した結果より，ストア特定予測を優先することで高い性能向上を達成できることが明らかになった．ストアセットとストア特定予測のハイブリッドにより，86 %の曖昧なメモリ依存関係を解消できることが判明した．この時，分岐予測ミスの削減率により，0.7 から 1.7 の IPC 向上を達成できることが判明した．分岐予測ミスの削減率を向上させた場合に，著しい IPC の向上を確認した．

# 第 7 章

# 値予測によるデータ依存関係の解消

レジスタを介したデータ依存関係は，レジスタ数の不足が原因で生じる出力依存関係，逆依存関係と，ハードウェア資源に依存しない真の依存関係に分類できる [HP95]．この中で，出力依存関係と逆依存関係はレジスタ名前替えで解決できる．残る真の依存関係を解消することは困難と考えられてきたが，近年，真の依存関係を解消する技術として，値予測を用いた投機処理の手法が提案された [LWS96, LS97a, WF97, SS97]．これは，実際に計算をおこなってデータ値を得る代わりに，生成されるデータの値を予測することで処理を進めておくという投機処理技術である．この値予測により，命令レベル並列性の上限を引き上げることが可能となる [GG98]．

予測により正しい値を得ることができれば真の依存関係を解消できるが，予測に失敗すれば誤った値を利用した命令の再処理が必要となり，プロセッサ性能にペナルティを与えてしまう．幾つかの研究 [GG98, LS97a, WF97] において，ミス率の低減を考慮した値予測機構が提案されているが，これらの値予測機構を用いても，全実行命令に対して 2%〜5%程度の頻度で予測ミスが発生する．値予測による性能向上を維持するためにはミス率の低減が重要な課題となっている．

本章では，値予測をおこなう命令までの制御流の変化を考慮することで，ストライド値予測機構の欠点を補う 2 レベル・ストライド値予測機構を提案する．提案手法では，ストライド値予測で予測ミスが発生した際に，予測ミスを引き起こした命令の演算結果とその命令に至るグローバルな分岐履歴をテーブルに保存し，この情報を用いてストライド値予測で予測ミスとなる領域を正しく予測することを試みる．このため，予測ミスを削減できるとともにヒット率の向上を期待できる．2 レベル・ストライド値予測機構の評価として，幾つかの重要なパラメタを変化させ，ヒット率とミス率を議論する．また，スーパースカラ・プロセッサのシミュレータを用いた評価より，プロセッサ性能に与える影響を議論する．

## 7.1 関連研究

従来の研究は，実行レイテンシの長いロード命令に焦点を絞り，ロードするデータ値を予測する値予測機構 [LWS96, 佐藤 98] と，演算結果を生成する全ての命令の結果の値を予測する値予測機構 [GG98, LS97a, WF97, SS97, NGS99, CW99] とに分類できる．本研究は後者の分類に入り，ストア，分岐，特殊命令（例えば，DEC Alpha の PAL 命令）を除く全ての命令の演算結果を予測の対象とする．このように多くの命令を予測対象とした値予測機構には次のものがある．

### 7.1.1 Last-Value 予測機構

Last-Value 予測機構 [LWS96, LS97a] は，予測する命令の前回の演算結果を今回の予測値として利用する．文献 [LS97a] では，予測ミスの回数を削減するために，動的に変化するカウンタからなる分類テーブル（Classification Table）を用いて予測精度の高い命令のみ予測する Last-Value 予測機構を提案している．

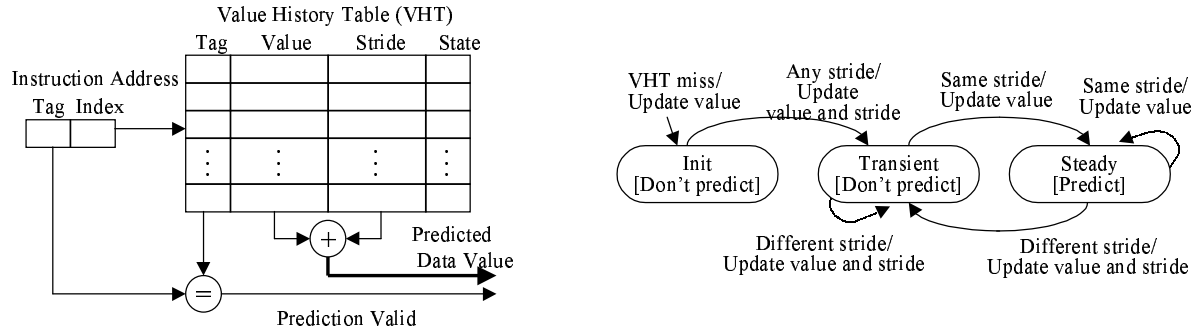
### 7.1.2 ストライド値予測機構

ストライド・ベースの値予測機構は，予測する命令の過去 2 回の演算結果の差分 $Stride$ と，最も近い過去に得られた値 $Value$ から，$Value + Stride$ により予測値を計算する．ストライド・ベースの値予測機構を図 7.1(a) に示す．値履歴テーブル（Value History Table, VHT）は前回の演算結果やストライド等を格納するテーブルである．値履歴テーブルの更新の仕方に選択の余地があるが，3 つの状態を用いて予測ミスを削減する文献 [WF97] のアルゴリズムを説明する．

VHT のエントリは図 7.1に示す Tag，Value，Stride，State（状態）の 4 つのフィールドを持つ．状態は，Init，Transient，Steady のどれかの値を持つ．

状態の変化と予測アルゴリズムを図 7.1(b) に示す．最初に値を生成する命令に出会った時には予測をおこなわない．値が生成された時に VHT のエントリが割り当てられ，(i) 演算結果を Value フィールドに格納し，(ii) 状態を Init に変更する．

状態が Init だった場合には予測をしない．しかし，その命令が値（D1）を生成した場合にはストライドの開始となった可能性があるので，(i) ストライドを $S1 = D1 - Value$ より計算し，(ii)D1 を Value フィールドに，S1 を Stride フィールドに格納し，(iii) 状態を Transient に変更する．




(a) Block diagram of stride value predictor

(b) State transition of stride value predictor

図 7.1: ストライド値予測機構のブロック図と状態遷移図

状態が Transient の時に命令の別のインスタンスが実行されたときも予測をおこなわない．そのインスタンスが値（D2）を生成した場合には，(i) ストライドを $S2 = D2 - Value$ で計算し，(ii) D2 をエントリの Value フィールドに格納し，(iii) もし S2 の値が前のストライドと等しかった場合には，状態を Steady に変更する．ストライドが等しくなければ S2 を Stride フィールドに格納する．

状態が Steady の時には Stride と Value を用いて値を予測する．計算したストライドが前回のストライドと等しい場合には，Value フィールドを更新する．もし異なったストライドが得られた時には，状態を Transient に変更するとともに Value，Stride の値を更新する．

### 7.1.3 グローバル・コンテキストを利用した値予測機構

文献 [NGS99, CW99] では演算結果を予測する命令以外の命令の挙動（グローバル・コンテキスト）を利用する値予測機構を議論している．グローバル・コンテキストとして，グローバルな分岐履歴を利用する値予測機構を次にまとめる．

Per-path stride per-path value(PS-PLV) 予測機構 [NGS99] はストライド・ベースの値予測機構で，命令アドレスと分岐履歴レジスタを用いて値履歴テーブルのインデックスを生成する．これにより，実行パス毎に異なった Last-Value とストライドを利用した予測が可能となる．PS-PLV 予測機構のブロック図を図 7.2に示す．ただし，テーブル更新のための幾つかのフィールドを値履歴テーブルから省略した．

Per-path stride(PS) 予測機構 [NGS99] はストライド・ベースの値予測機構で，Last-Value とストライドを別々のテーブルに保存し，ストライドを選択するために分岐履歴を利用す

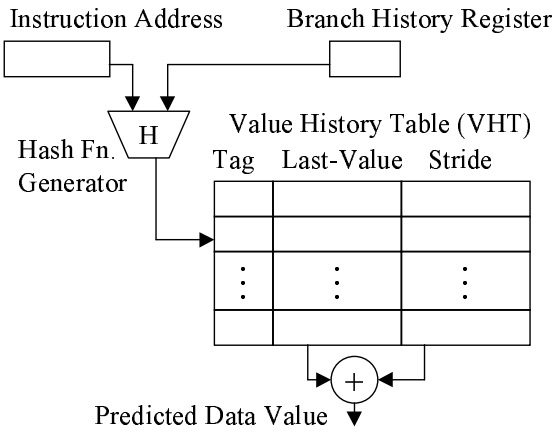



図 7.2: Per-path stride per-path value(PS-PLV) 予測機構のブロック図

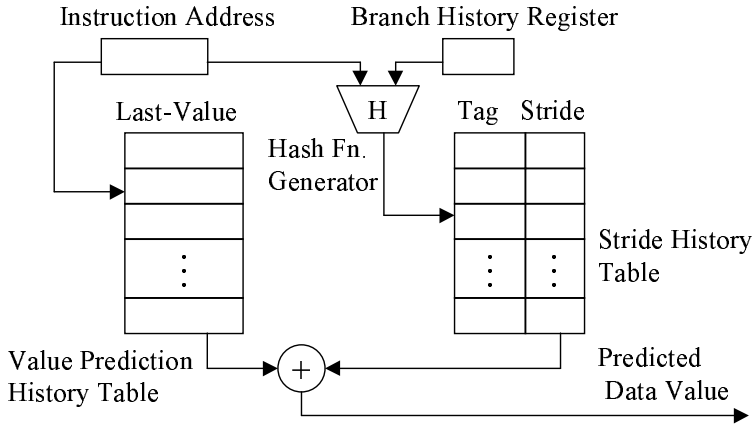



図 7.3: Per-path stride（PS) 予測機構のブロック図

る．これにより，実行パス毎に異なったストライドを利用した予測が可能となる．図 7.3にPS 予測機構のブロック図を示す．ただし，テーブル更新のための幾つかのフィールドを図から省略した．

文献 [NGS99] では，PS-PLV 予測機構と PS 予測機構の予測成功率を評価しており，ストライド値予測機構と比較してそれぞれ 6.9%と 8.4%の予測成功率の向上を確認している．

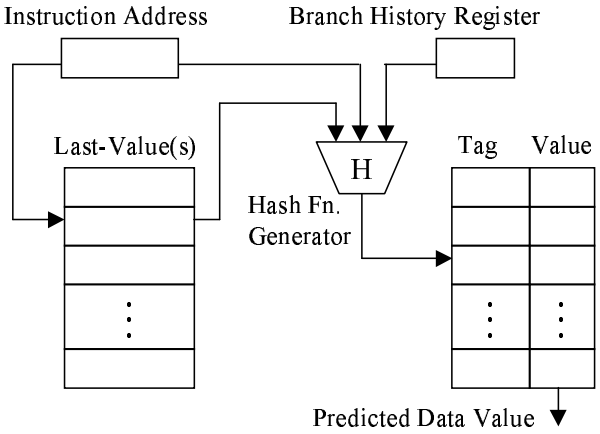



図 7.4: Last-Value と分岐履歴を利用する 2 レベル値予測機構

文献 [CW99] では，命令アドレスと分岐履歴レジスタに加えて，予測をおこなう命令の過去の実行結果 (Last-Value) を利用してテーブルのインデックスを作成する 2 レベルの値予測機構を検討している．この値予測機構のブロック図を図 7.4に示す．

### 7.1.4 その他の値予測機構

文献 [WF97] では，動的に生成される値の多くは最近のユニークな 4 個以内の値であるという結果に注目し，パターン・ベースの 2 レベル値予測機構を提案している．この 2 レベル値予測機構では，VHT に 4 つの異なった値を格納しておき，次の 2 段階で値を予測する．最初のレベルでは，予測するインスタンスの命令アドレスを用いて VHT のエントリを引き，そこに格納されている 4 つの値を出力する．2 レベル目で，4 つの中から 1 つの値を選択することで予測値を得る．2 レベル目の選択は，過去の $p$ 回の予測結果とカウンタを用いて決定するが詳細は省略する．

文献 [WF97] では，Last-Value 予測とストライド値予測のハイブリッド値予測機構，ストライド値予測と 2 レベル値予測のハイブリッド値予測機構を提案評価しており，複数の

予測アルゴリズムを利用することで予測範囲が広がることを確認している．

文献 [SS97] では，過去の連続した有限個の値の履歴（コンテキスト）と過去の実行履歴を比較し，高い確率で実行されるパターンに基づいて予測をおこなうコンテキスト・ベースの値予測機構を提案している．この値予測機構では，次のように $a\ a\ a\ b\ c\ a\ a\ a\ b\ c\ a\ a\ a$ と値が変化した時 ($a, b, c$ は 32 ビットまたは 64 ビットの値を表すシンボル)，実行確率の高いパターン $a\ a\ a\ b$ とコンテキスト $a\ a\ a$ の比較により値 $b$ を予測する．文献 [SS97] ではコンテキスト・ベースの値予測機構がストライド値予測機構や Last-Value 予測機構より高い予測成功率を達成する可能性があるとしているが，ハードウェア量や実装に関する議論は今後の課題としている．

### 7.1.5 予測機構のカスケード接続

文献 [RC99, 佐藤 99] において，値予測とその他の予測機構との組み合わせが議論されている．文献 [佐藤 99] では，値予測が利用できなかったロード命令に関してはデータアドレスの予測を用いて投機的に処理を進める協調型予測器を議論している．文献 [RC99] では，固定された順序により，値予測，メモリリネーミング，データアドレス予測，依存関係予測の 4 つの投機技術を利用する協調型予測器を議論している．本稿で議論するグローバルな分岐履歴を利用する値予測機構は，複数の予測機構の利用という意味で協調型予測器と言えるが，例えば，分岐予測の確信度により値予測の動作を変化させるといったより密接な協調は今後の課題といえる．

### 7.1.6 値予測機構とプロセッサの性能向上

本稿で用いる値予測のヒット率とミス率という言葉を定義する．値予測機構が正しく値を予測した回数を実行命令数で割った値としてヒット率を定義する．値予測機構が間違った値を予測した回数を実行命令数で割った値としてミス率を定義する．ヒット率だけでなくミス率を示す理由は，予測をおこなわない命令が存在するためであり，これらのヒット率とミス率を足し合わせた割合が全実行命令に対して値予測をおこなった割合となる。

分類表を用いた Last-Value 予測（LV+CT），ストライド値予測（Stride），2 レベル値予測（Two-level），ストライドと 2 レベルのハイブリッド値予測 (Hybrid) のヒット率とミス率を表 7.1にまとめる．これらの値は文献 [GG98] のデータを用いて計算した．表 7.1から，複雑な値予測機構を用いることでヒット率とミス率が共に増加していくことがわかる．ハイブリッド値予測機構を用いた場合には，全実行命令の約 40%の演算結果を正しく予測

| | LV+CT | Stride | Two-level | Hybrid |
|---|---|---|---|---|
| Hit | 21.7% | 29.8% | 29.4% | 39.9% |
| Miss | 1.7% | 2.9% | 3.9% | 5.9% |

表 7.1: 全実行命令に対する予測ヒット率とミス率

できる反面，平均で 17 命令実行するたびに 1 回の値予測ミス（ミス率 5.9%）が発生することになる．

表 7.1に示したヒット率とミス率は値予測機構を評価する際の重要な情報となるが，値予測機構はハードウェア・コストとプロセッサ性能の向上率を用いて評価しなければならない．プロセッサの性能向上は，予測のヒット率，ミス率，ヒットした場合の利得，ミスした時のペナルティに依存するので，値予測を用いてプロセッサの性能を向上させるためには，性能向上につながる命令の演算結果を正しく予測するとともに，予測ミスの回数を削減し，予測ミスのペナルティを削減することが重要となる．

## 7.2 2レベル・ストライド値予測の提案

本節では，ストライド値予測機構の予測ミスと，予測する命令までの条件分岐結果（分岐履歴レジスタ）の関係を示した後に，分岐履歴レジスタを用いてストライド値予測機構を拡張する 2 レベル・ストライド値予測機構を提案する．

従来から，分岐予測において分岐履歴レジスタを利用した 2 レベルの予測機構が提案されている [McF93b]．また，近年，分岐履歴レジスタを用いた値予測が文献 [小池 99, NGS99, CW99, 吉瀬 99] において検討されている．本節で提案する 2 レベル・ストライド値予測は文献 [吉瀬 99] を基本としている．7.1.3節で述べた文献 [NGS99, CW99] の値予測と本節で提案する 2 レベル・ストライド値予測は，独立に提案された，分岐履歴レジスタを利用するという点が共通する方式の値予測であるが，本方式は文献 [NGS99, CW99] にない以下の特徴を持つ．文献 [NGS99, CW99] で検討されている値予測機構がヒット率の向上を目指しているのに対して，提案手法ではヒット率の向上とともにミス率の削減を目的とする．手法として，ストライド値予測で予測ミスが発生した際に，予測ミスを引き起こした命令の演算結果とその命令に至るグローバルな分岐履歴をテーブルに保存し，この情報を用いてストライド値予測で予測ミスとなる領域を正しく予測する方式を提案する．

```
Value Sequence: 1   2   3   4   5   1   2   3   4   5   1   ...
                NP  NP  NP  Hit Hit Miss NP  NP  Hit Hit Miss ...
```

図 7.5: 定期的にストライドが変化する場合の予測結果

### 7.2.1 値予測ミスと分岐履歴レジスタ

ストライド・ベースの値予測では，演算結果が一定間隔で変化を続ける場合に予測が成功する．しかし，典型的なループ構造（for 文）に見られるように，実際のプログラムでは，演算結果が一定の間隔で変化を続けるわけではなく，なんらかの条件により，値が初期化されることが多いと考えられる．7.1.2節で示したストライド値予測機構を用いて 1〜5 の値を繰り返す例を予測した結果を図 7.5に示す．図の下段は予測結果 NP（No Predict）, Hit, Miss を表している．この例の場合, 値が 1 に初期化される際に予測ミスが発生する．また，値が初期化されるまでの間隔が短くなるに従って正しく予測できる割合が減少する．もし，この初期化のタイミングと初期値を正しく予測できたとすれば，1〜5 の値を繰り返す例の様に，一定間隔でストライドが変化する場合に，100%に近い予測成功率を達成できる．

我々は，初期化のタイミングを予測するために，予測する命令に至るまでのグローバルな条件分岐命令の履歴に注目した．すなわち，予測値が一定間隔で変化している時と，初期化される時で異なった制御の流れをとり，この制御流の変化を検出することで初期化のタイミングを予測できるという仮定により予測をおこなうことを提案する．この仮定を図 7.6の例を用いて説明する．

図 7.6上のソースコードにおいて，インナーループの誘導変数 j の値予測を考える．SPARC アーキテクチャをターゲットとして，ソースコードをコンパイルした結果を図 7.6下に示す．%o0, %o1, %o2 はレジスタを表し，それぞれの命令の最後のレジスタがデスティネーションのレジスタを示している．ただし，分岐命令で利用するフラグを設定する (6)cmp と分岐命令にはデスティネーションのレジスタは存在しない．命令 (5) がソースの j++ に対応し，この命令が生成する値の予測を検討する．命令 (5) は図 7.5に示した様に 1，2，3，4，5 という値を繰り返し生成する．また，このプログラムを実行した時の条件分岐命令（命令 7 と 10）の結果は分岐成立を $T$，不成立を $N$ で表現すると，$TTTTTNT\ \ TTTTTNT...$ というビット列となり，予測する値が 1 に初期化される時点までの分岐結果は $...NT$ となっていることがわかる．このコードの場合には値予測の際に最近の条件分岐の結果 2 ビットからなるビット・パターン（2 ビットの分岐履歴レジスタ）が $NT$ だった場合に値 1 を予

```
    for(i=0; i<10; i++)
       for(j=0; j<5; j++) sum += j;

(1)   mov 0,%o1        ; sum=0
(2)   mov 9,%o2        ; i=9
(3)   mov 0,%o0        ; j=0
   .LL13:
(4)   add %o1,%o0,%o1  ; sum+=j
   .LL12:
(5)   add %o0,1,%o0    ; j++, Predict this!
(6)   cmp %o0,4        ; j<4 ?
(7)   ble,a .LL12      ; conditional branch
(8)   add %o1,%o0,%o1  ; sum+=j,delayed slot

(9)   addcc %o2,-1,%o2 ; i--
(10)  bpos,a .LL13     ; conditional branch
(11)  mov 0,%o0        ; delayed slot, j=0
```

図 7.6: 分岐履歴を用いることで予測ミスを改善できる例

測することで予測ミスを回避できる．このように，制御流の変化を検出することで初期化のタイミングを予測できるという仮定により2レベル・ストライド値予測機構を提案する．

### 7.2.2 2レベル・ストライド値予測機構

2レベル・ストライド値予測機構は，図7.7に示す様に，ストライド値生成部，初期値生成部，セレクタ部の3つの要素から成る．ストライド値生成部は，図7.1に示したストライド値予測機構を拡張したもので，ストライドを用いて予測値を生成する．初期値生成部は，分岐履歴レジスタ（BHR）とプログラム・カウンタを用いて初期値を予測する．セレクタ部は，ストライド値生成部と初期値生成部の出力から，適切な予測値を選択するとともに，予測の確信度を評価する．この確信度が高いと評価された場合のみ予測を有効にすることで，予測ミスを削減する．

初期値生成部は，命令アドレスと分岐履歴レジスタを用いて初期値を出力する初期値テーブルから成る（図7.8）．初期値を得るために，まず，命令アドレスからインデックスを計

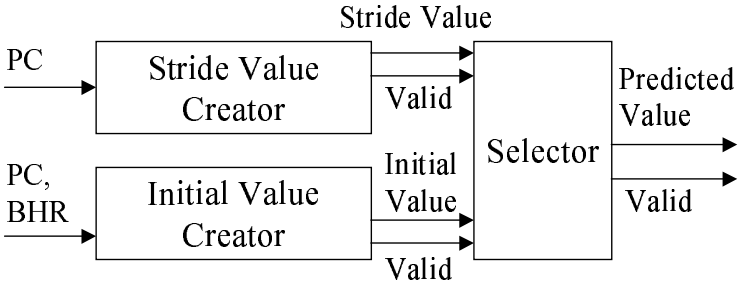



図 7.7: 2 レベル・ストライド値予測機構のブロック図

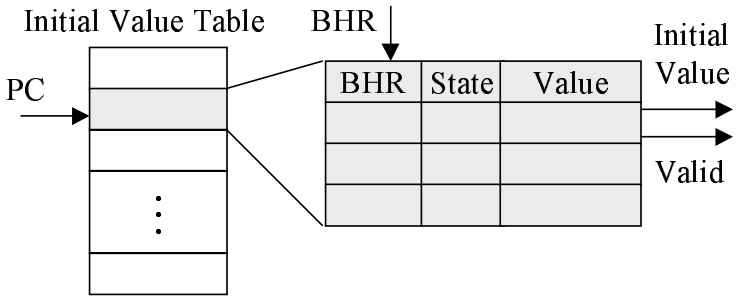



図 7.8: 4 ウェイの初期値テーブル

算し，テーブル中のエントリを選択する．それぞれのエントリは，BHR， State，初期値，というフィールドを $n$ 組持っている（$n$ ウェイの初期値テーブルと呼ぶ）．次に，これらの $n$ 組の中から，分岐履歴レジスタが一致するウェイを検索し，一致した場合に初期値を出力する．ただし，初期値の予測成功率を上げるために，それぞれのウェイには前回の予測成功・失敗を保存する State というフィールドを用意し，前回の予測が失敗した場合には，予測を無効にする．

セレクタ部は，まず，ストライド値生成部，初期値生成部が生成した値から適切な値を選択する．もし，ストライド値生成部のみが値を生成した場合には，ストライド値を予測値とする．もし，両方が値を生成した場合には，初期値を予測値とする．それ以外の場合には，予測を無効とする（初期値生成部のみ値を生成した場合には予測が無効となる点に注意）．次に，セレクタ部は予測の確信度評価を用いた絞込みをおこなう．予測ミスを抑えるために，予測する命令の過去の履歴が，ある値（閾値）以上連続して成功している場合にのみ予測を有効とする．

次に，状態の更新アルゴリズムを説明する．これらの状態更新は全て命令のリタイア時におこなわれる．

ストライド値生成部は，次の変更点を除いて，7.1.2節で説明したアルゴリズムによりテーブルの更新と予測をおこなう．ストライド値生成部の予測がミスした場合でも，初期値生成部がおこなった予測がヒットした場合には状態を Transient に変更せず，Steady のままで予測を継続するように変更する．

初期値生成部は，ストライド値生成部が予測をおこない，かつストライド値生成部の予測がミスした場合に，計算により得られた結果を初期値として初期値テーブルに登録する．この時，初期値テーブルの全てのウェイが有効だった場合には，LRU でウェイを入れ替える．また，初期値生成部の予測が有効だった場合には，演算結果と初期値生成部の予測値とを比較し，初期値テーブルの State フィールドを更新する．

セレクタ部で確信度の評価を利用する場合には，予測値と演算結果を比較し，それぞれの命令ごとに，連続して予測が成功した回数を専用のテーブルに保存する．

## 7.3 評価環境

### 7.3.1 プロセッサ・モデル

予測する時刻とテーブルを更新する時刻とのサイクル数のずれにより，予測機構のヒット率とミス率が変化する．このため，評価の際には用いるプロセッサ・モデルに注意する必要がある．7.4節の評価では，この問題を回避するために，スカラ・プロセッサのシミュレータを用いて 2 レベル・ストライド値予測機構のヒット率とミス率を評価する．7.5節において，プロセッサ性能を測定する際には，4節で定義したベースラインプロセッサを利用する．

## 7.4 ヒット率とミス率の評価

スカラ・プロセッサのシミュレータを用いて，2 レベル・ストライド値予測機構のヒット率とミス率を測定する．ここでは，初期値テーブルのウェイ数，初期値生成部における分岐履歴レジスタのビット数，セレクタ部において確信度を評価する際の閾値（確信度評価の閾値と呼ぶ），という 3 つのパラメタを変化させ，2 レベル・ストライド値予測機構のヒット率とミス率を評価する．測定結果で示すヒット率とミス率は，8 つのベンチマーク

| | 1 way | 2 way | 4 way | 8 way |
|---|---|---|---|---|
| 8bit | 38.9%,2.48% | 39.7%,2.40% | 40.4%,2.37% | 41.1%,2.31% |
| 16bit | 38.9%,2.39% | 39.7%,2.30% | 40.6%,2.17% | 41.6%,2.04% |
| 24bit | 38.9%,2.36% | 39.5%,2.26% | 40.4%,2.13% | 41.4%,2.00% |
| 32bit | 38.9%,2.32% | 39.4%,2.23% | 40.2%,2.12% | 41.2%,1.95% |

| | 16 way | 32 way | 64 way |
|---|---|---|---|
| 8bit | 41.5%,2.29% | 41.6%, 2.28% | 41.6%,2.28% |
| 16bit | 42.4%,1.93% | 43.0%, 1.89% | 43.4%,1.88% |
| 24bit | 42.3%,1.84% | 43.0%, 1.77% | 43.5%,1.74% |
| 32bit | 42.1%,1.77% | 42.8%, 1.67% | 43.4%,1.63% |

表 7.2: 2 レベル・ストライド値予測機構のヒット率とミス率

プログラムの調和平均を用いる．以降の評価では，予測する命令間の競合を避けるために，初期値テーブルのエントリ数，ストライド値生成部における値履歴テーブルのエントリ数を非常に大きな値（64K）に設定して評価する．

初期値生成部における分岐履歴レジスタのビット数を 8，16，24，32 ビットに，初期値テーブルのウェイ数を 1，2，4，8，16，32，64 に設定した場合のヒット率とミス率を表 7.2 に示す．ここでは，セレクタ部における確信度評価を利用していない．

表 7.2の結果から，初期値テーブルのウェイ数を増やすことでヒット率が向上することを確認できる．また，分岐履歴レジスタのビット数を増やすことで，ミス率が低下することを確認できる．初期値テーブルのウェイ数を固定させた場合，分岐履歴レジスタのビット数を大きくすることでヒット率が僅かに低下する．これは，分岐履歴レジスタのビット数を大きくすることにより，ウェイの競合が深刻になるためと考えられる．表 7.2の評価においては，初期値テーブルのウェイ数 64，分岐履歴レジスタのビット数 32 とした設定が，ヒット率，ミス率ともに良い結果を出した．以降，この設定を用いて評価を続けていく．

3 つ目のパラメタ，確信度評価の閾値を 0，1，2，4，8 に変化させた場合のヒット率とミス率を図 7.9にまとめる．確信度評価の閾値 0 という設定は，確信度評価を用いないことを意味している．初期値テーブルのウェイ数 64, 分岐履歴レジスタ 32 ビットに設定した 2 レベル・ストライド値予測機構について評価した．比較のために，同じ条件で確信度評価を用いた場合のストライド値予測機構の評価結果を加えてある．

値予測機構は，確信度評価の閾値で指定した回数以上連続して予測が成功している命令

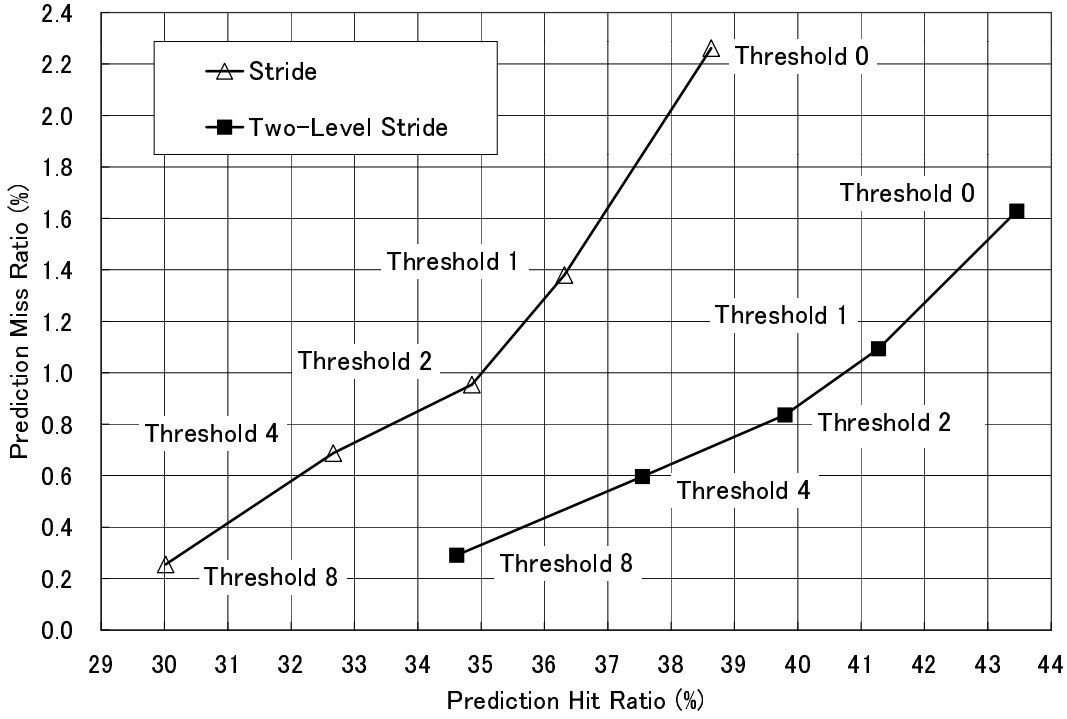



図 7.9: 確信度評価の閾値を変化させた時のヒット率とミス率

の予測を有効とする．このため，確信度評価の閾値を大きくすることは，値予測をおこなう割合の低下につながり，図 7.9 に示したヒット率とミス率の低下を引き起こす．図 7.9 の結果から，同じ閾値を用いた場合には，2 レベル・ストライド値予測機構の方が 4.5%～4.9%ヒット率が高いことがわかる．また，閾値 8 のとき微妙にミス率が増加している点を除いて，2 レベル・ストライド値予測機構の方がミス率に関しても良い結果を出している．閾値が小さい時に，2 レベル・ストライドによるミス率改善が顕著になり，閾値 0 の時には 0.6%の予測ミスを削減できている．表 7.1に示した様に，複雑な値予測機構を用いた場合にはヒット率と共にミス率が増加する傾向にあるが，2 レベル・ストライド値予測機構はヒット率，ミス率ともに改善できるという特徴がある．

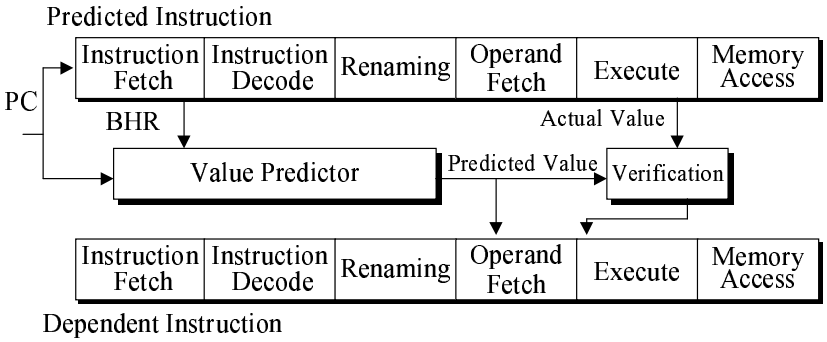



図 7.10: 必要としている命令への予測値の供給

## 7.5 プロセッサ性能に与える影響

2 レベル・ストライド値予測機構をスーパースカラ・アーキテクチャに組み込む場合には，ストライド値生成部の構成を一部変更する必要がある．予測値を履歴テーブルから得られる $Value + Stride$ で計算する代わりに，予測する命令と同一の命令がリオーダバッファ内に存在する場合には，リオーダバッファから $Value$（演算結果または予測値）を出力するように変更する．この拡張をおこなわない場合には，履歴テーブルが更新される前に，同じアドレスの命令が複数フェッチされた場合に，それらの予測値が同じ値となり，予測のヒット率が著しく低下する．

### 7.5.1 プロセッサモデル

値予測に失敗した場合には，間違った値を利用した命令の再実行が必要となる．本評価では，値予測にミスした場合，正しいオペランドを用いて予測ミスした命令の後続命令を実行ステージから再処理することで予測ミスから回復する．このために，命令が発行された後にも，リタイアするまでは，リザベーション・ステーションのエントリを解放しない．予測ミスの場合にはリザベーション・ステーションに格納されている情報を用いて，命令の再実行が可能となる．文献 [LS97a] で議論されている様に，予測ミスした命令にデータ依存関係を持つ命令だけを実行ステージから再処理することで予測ミスから回復することができるが，本実装では，予測ミスした命令の後続命令を全て再実行するとした．

図 7.10にシミュレータのパイプライン構成と，値生成と結果比較のタイミングを示す．予測値を生成する命令は，命令フェッチからオペランド・フェッチ・ステージまでの 4 サイクルの間に予測値を生成すればよい．値生成のための処理をパイプライン化することで，プ

ロセッサのサイクル時間に大きな影響を与えることなく予測値を生成できると考えている．本評価では，予測値と演算結果の比較を実行ステージ（ロード命令の場合にはメモリアクセス・ステージ）で行なうとした．

### 7.5.2 スーパースカラにおける値予測機構のヒット率

プロセッサの性能を見る前に，スーパースカラ・プロセッサにおける値予測機構のヒット率とミス率を確認しておく必要がある．4節で定義したベースラインプロセッサを用いて測定したヒット率とミス率を，図7.11の破線に示す．ストライド値予測機構，2レベル・ストライド値予測機構ともに，スカラ・プロセッサにおける評価結果からの性能低下を確認できる．特に，閾値が0の時に性能低下が大きく，ストライド値予測の場合でヒット率が4.5%，2レベル・ストライド値予測の場合でヒット率が5.0%低下する．本評価では，命令がリタイアする時に，値予測で必要となる履歴テーブルの更新をおこなっている．このために，予測をおこなう時刻とテーブルを更新する時刻とのずれが生じ，この性能低下を引き起こしている．テーブル更新のタイミングによる性能低下は，テーブル更新の一部を投機的に更新するといった改良で削減できる可能性がある．これらの改良は，今後の課題である．次に示す並列性は，図7.11の破線で示したヒット率，ミス率における並列性である．

### 7.5.3 値予測による命令レベル並列性の変化

値予測機構による命令レベル並列性の変化を図7.12に示す．図7.12では，8本のプログラムとそれらの調和平均について，それぞれ11個の値を示している．それらは，左から，値予測を用いなかった場合の並列性（ベースライン），次の5本がストライド値予測機構を用いた場合の並列性で，確信度評価の閾値を0，1，2，4，8と変化させた時の並列性をそれぞれ示している．最後の5本が2レベル・ストライド値予測機構を用いて確信度評価の閾値を0，1，2，4，8と変化させた時の並列性である．

図7.12の結果を検討する．まず，m88ksim, vortexにおいて値予測により大幅な性能向上が得られることがわかる．ただし，これらのプログラムは，値予測を用いなかった場合にも高い並列性を示しており，これらの並列性の更なる向上が調和平均ではそれほど大きな性能向上につながらない点に注意する必要がある．m88ksim, vortex以外のプログラムでは，確信度評価の閾値により性能が低下する場合がある．特に，閾値0で確信度評価を利用しない場合には，ストライド値予測機構，2レベル・ストライド値予測機構ともに幾つかのプログラムで性能低下が起こっている．平均で見た場合にも，ストライド値予測機

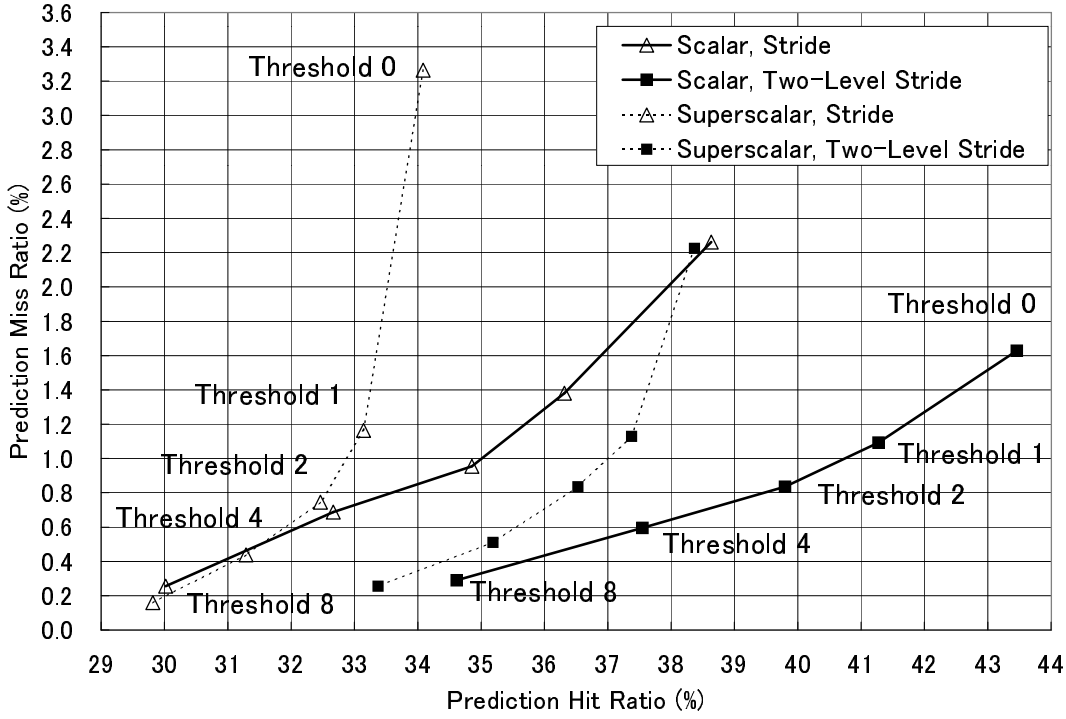



図 7.11: 確信度評価の閾値を変化させた時のヒット率とミス率

構で閾値 0 の場合にはベースラインより性能が低下する．

値予測による性能向上が著しいプログラムには，値予測のヒット率と分岐予測の成功率が高いという特徴がある．2 レベル・ストライド値予測で確信度評価の閾値 8 の設定におけるデータを用いて議論するが，最も著しい並列性向上を示した m88ksim の場合には，値予測のヒット率は 67.7%と平均の 34.6%と比較して非常に高い．また，分岐予測に関しても成功率は 98.4%と平均成功率 95.2%に対して非常に高い．m88ksim に次いで並列性向上の大きい vortex の場合には，値予測のヒット率は 39.7%と平均より 5%高く，分岐予測に関しても成功率 99.2%と非常に高い．これらは，値予測により著しい性能向上をもたらすプログラムの傾向である．より詳細な分岐予測と値予測の相関関係の検討は今後の課題である．

プロセッサ性能と確信度評価の閾値の関係を見てみると，m88ksim の一部を除いて，確信度評価の閾値を大きくした方が性能が高いことがわかる．この傾向は，ストライド値予測機構，2 レベル・ストライド値予測機構ともに確認できる．この結果は，ヒット率の増加

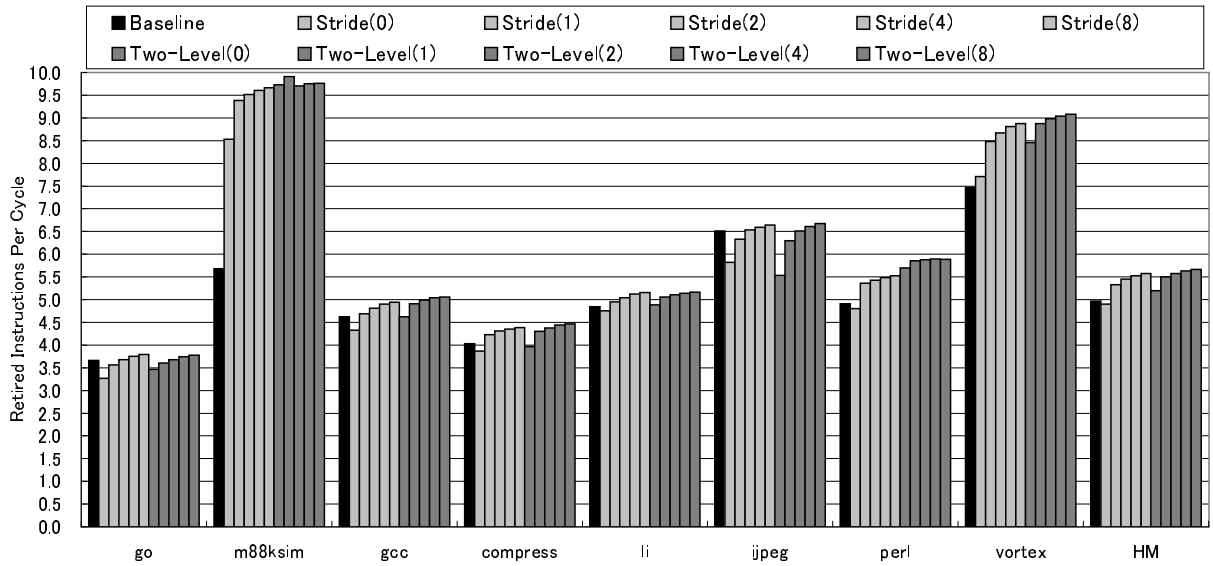



図 7.12: ストライド値予測機構と 2 レベル・ストライド値予測機構による命令レベル並列性の変化

以上に，ミス率の改善が性能向上につながることを意味している．ただし，確信度評価の最適な閾値は，予測のヒット率，ミス率，ヒットした場合の利得，ミスしたときのペナルティにより決まるので，予測ミスによるペナルティを小さくすることができれば，閾値を小さくした場合でも，高い性能を達成できる可能性がある．

最後に，ストライド値予測機構と，2 レベル・ストライド値予測機構とを比較する．調和平均で見た場合には，全ての閾値で，2 レベル・ストライド値予測機構を用いた方が性能が高い．しかし，ストライド値予測機構との性能差は，命令レベル並列性で見て，確信度評価の閾値 0 の時 0.41，閾値 8 の時 0.11 とそれほど高くない．閾値が 8 の時に最も高い並列性を得ているが，この時のベースラインからの向上率で比較すると，ストライド値予測機構による性能向上率が 12%，2 レベル・ストライド値予測機構による性能向上率が 14%と，その差は 2%という結果になった．図 7.9に示した様に，2 レベル・ストライド値予測機構によるヒット率とミス率の改善は，確信度評価の閾値を小さくした場合に大きく現れる．さらに，スーパースカラ・モデルを用いることで生じるテーブルの更新タイミングによる低下が，値予測による利得を削減している．予測ミスからの回復機構と履歴テーブルの更新手法の改良により，2 レベル・ストライド値予測機構による性能向上率を更に引き上げることができると考えている．

## 7.6 本章のまとめと今後の課題

予測する命令までの制御流の変化を利用する 2 レベル・ストライド値予測機構を提案し，その可能性を議論した．2 レベル・ストライド値予測機構における 3 つのパラメタ，初期値テーブルのウェイ数，初期値生成部における分岐履歴レジスタのビット数，セレクタ部における確信度評価の閾値を変化させ予測ヒット率とミス率を測定した結果，確信度評価の閾値 0 の時に最も高いヒット率 43.4%を達成した．これはストライド値予測機構より 4.8%高いヒット率となる．また，閾値が 0 の時，ヒット率だけでなくミス率に関しても 0.6%の改善を確認した．

スーパースカラ・プロセッサのシミュレータを用いた評価より，2 レベル・ストライド値予測機構がプロセッサの性能に与える影響を検討した．並列性の向上率では，値予測機構における確信度評価の閾値を 8 とした場合に最も高い性能向上を示し，ストライド値予測機構を用いることで 12%，2 レベル・ストライド値予測機構を用いることで 14%の性能向上を確認した．本評価で用いたスーパースカラ・プロセッサでは，2 レベル・ストライド値予測機構の有効性を示すには至らなかった．ただし，2 レベル・ストライド値予測機構によるヒット率とミス率の改善は，確信度評価の閾値を小さくした場合に大きく現れる．更なる予測ミスペナルティの削減と最適な確信度評価の閾値の検討は今後の課題である．

本評価では，予測する命令間の競合を避けるために初期値テーブルのエントリ数，ストライド値生成部における値履歴テーブルのエントリ数を 64K と非常に大きな値に設定した．また，初期値テーブルのウェイ数に関しても 64 ウェイと大きな値を想定して，2 レベル・ストライド値予測機構の可能性を検討した．ストライドや初期値を格納するフィールドのビット数，テーブルのエントリ数とテーブルの構成方式を含め，必要となるハードウェア量と性能の関係を議論する必要がある．ハードウェア量に関しては，予測ミスした場合の回復機構で必要となるハードウェア量も同時に検討する必要がある．これらの検討は今後の課題である．

文献 [RC99, 佐藤 99] で議論されている様に，値予測と分岐予測に加えて様々な予測機構がプロセッサ上に実装されていく可能性がある．これらの複数予測機構によるカスケード方式の有効性に関する検討は今後の課題である．

# 第 8 章

# 高レベル投機技術を用いた複数パス実行プロセッサ

プロセッサの性能を向上させるためには，命令供給，メモリを介したデータ処理，レジスタを介したデータ処理というそれぞれの領域において処理の効率化を図る必要がある．これら 3 つの領域を図 8.1に示す．

命令供給を妨げる原因は主に分岐命令により生じる制御依存関係であり，これを解消するために，分岐予測 [McF93a] や，分岐成立と不成立の両方のパスにおいて投機的に処理を進める**複数パス実行** [US95, Che98, KPG98] が提案されている．本研究においても，確率モデルと JRS 確信度評価の利点を融合する JRS-PM 方式を提案し，その性能を 5章で議論した．

データキャッシュのミスの他に，メモリを介したデータ処理を妨げる原因には曖昧なメモリ依存関係がある．これは，先行するストア命令のメモリ参照アドレスが計算されていない場合にはデータ依存関係の有無を解析できないため，ロード命令を発行することができないという制約から生じる依存関係である．メモリ参照アドレスが計算される前にデータ依存関係の有無を予測し，曖昧なメモリ依存関係を解消する手法を**メモリ依存予測**と呼び，ストアセットを用いた予測機構 [CE98] が提案されている．6章では，ストアセットを用いて曖昧なメモリ依存関係を解消できない領域において，曖昧なメモリ依存関係を解消するストア特定予測を提案し，ストアセットとストア特定予測のハイブリッドを用いて 85%の曖昧なメモリ依存関係を解消できることを示した．

データフォワーディングにより，レジスタを介したデータ処理の効率を向上できる．しかし，データフォワーディングはデータを生成した ALU から利用する ALU にデータを直接転送する仕組みであり，真のデータ依存関係を解消している訳ではない．従来，真のデー

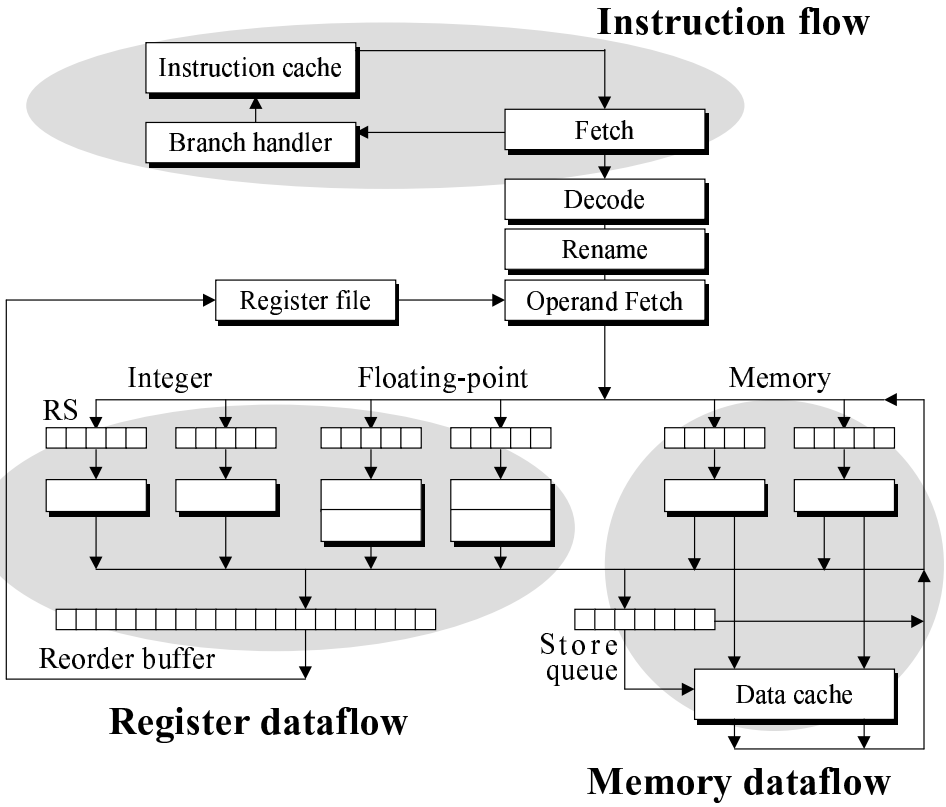



図 8.1: 高レベル投機技術のターゲット

タ依存関係の解消は困難と考えられてきたが，近年，これを解消する**値予測** [LS97a] が提案された．これは，実際に計算をおこなってデータを得る代わりに，生成されるデータの値を予測することで処理を進めておくという投機技術である．7章では，グローバルな分岐履歴を利用してストライド値予測を拡張する 2 レベル・ストライド値予測を提案し，その性能を評価した．

複数パス実行，メモリ依存予測，値予測という 3 つの投機技術は，その確立を目指して研究が進められており，各手法による性能向上が報告されている．しかしながら，これらの投機技術を同時に利用した場合の性能向上は議論されていなかった．本章では，複数パス実行，メモリ依存予測，値予測という **3 つの高レベル投機技術の融合手法**を議論する．複数パス実行プロセッサのシミュレータを用いた評価より，これらの融合手法による性能向上率を測定し，その有効性を明らかにする．

## 8.1 高レベル投機技術の融合の可能性

本節では，実行トレースを用いて依存関係を削減し，制御依存関係，曖昧なメモリ依存関係，真のデータ依存関係を解消することによる性能向上の可能性を議論する．それぞれの依存関係の解消による性能向上を明確にするために，次に示す構成の組合せを評価する．

- 条件分岐命令の分岐予測ミスを 0%から 100%まで 10%単位で確率的に削減する 11 構成
- 曖昧なメモリ依存関係を 100%解消する構成と，曖昧なメモリ依存関係を解消しない構成の 2 構成
- Last-Value 予測を用いた構成と，値予測を用いない構成の 2 構成

値予測に関しては，上限に相当する構成を設定する事は難しい．このため，現実的な手法として Last-Value 予測を利用する．Last-Value 予測では，予測ミスを削減するために 8.3.1 節で述べる 10 ビットカウンタを採用する．また，値履歴テーブルの競合を避けるためにエントリ数を非常に大きく設定した．本節の評価では，ベースラインプロセッサのシミュレータを利用して，以上の組み合わせ $11 \times 2 \times 2 = 44$ 構成における性能向上率を測定する．

図 8.2に測定結果をまとめる．図の横軸は，分岐予測ミスの削減率を表しており，図 8.2 左端の削減率 0% はベースラインプロセッサで定義した分岐予測を用いた場合の構成，右端の削減率 100%は，複数パス実行を用い，完全に条件分岐命令の分岐予測ミスを削減できるとした場合の構成である．その間の 10%から 90%の構成は，乱数を用いて確率的に分岐予測ミスを削減した場合の構成である．グラフには，分岐予測ミスだけを確率的に削減する構成 (Multi-Path)，Last-Value 予測を利用するとともに分岐予測ミスを削減する構成 (Multi-Path, Last-Value)，曖昧なメモリ依存関係を排除し分岐予測ミスを削減する構成 (Multi-Path, Memory Disambiguation)，曖昧なメモリ依存関係を排除し，Last-Value 予測を利用し，分岐予測ミスを削減する構成 (Multi-Path, Last-Value, Memory Disambiguation)，という 4 つの曲線を示した．グラフの縦軸は IPC で 5.28 というベースラインプロセッサに対する性能向上率である．

図 8.2の結果を議論する．左端に示した分岐予測ミスの削減率 0%における評価結果は複数パス実行を利用しない場合の性能向上率を示している．この時，Last-Value 予測とメモリ依存予測の同時利用による性能向上率は 22%，この時の IPC は 6.5 となった．一方，分岐予測ミスを解消できる場合の性能向上率は最大で 98%，この時の IPC は 10.4 であり，分岐予測ミスを削減することの重要性がわかる．

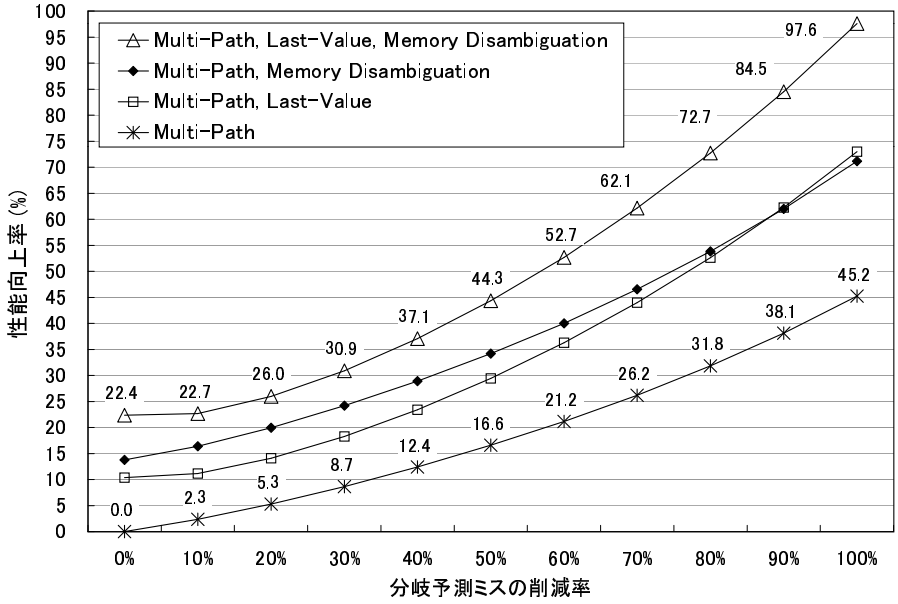



図 8.2: 高レベル投機技術による性能向上率

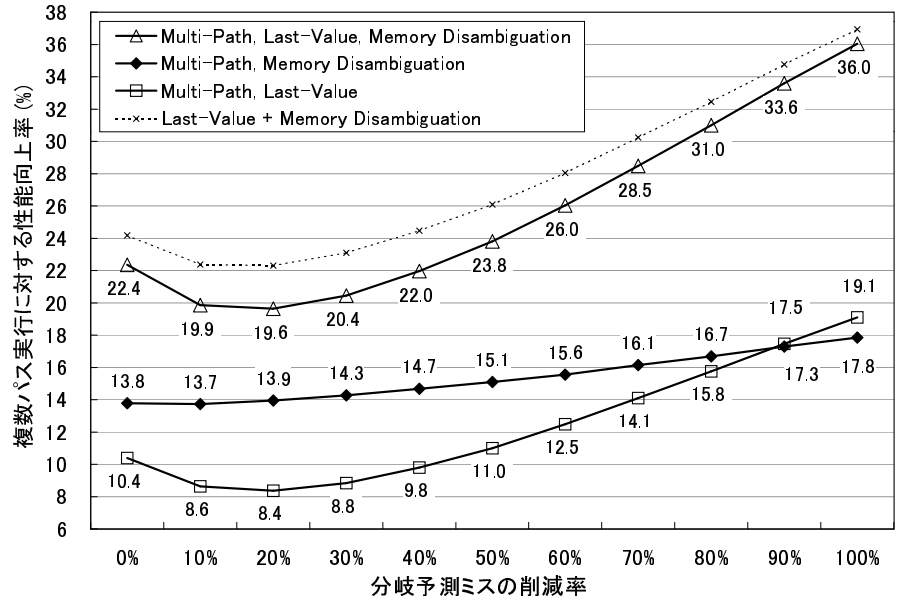



図 8.3: 複数パス実行の性能で正規化した性能向上率

メモリ依存予測と Last-Value 予測の影響を明確にするために，図 8.2に示した性能向上率を複数パス実行の性能で正規化した性能向上率を図 8.3 にまとめる．曖昧なメモリ依存関係による性能向上率は 14%から 18%であり，分岐予測ミスを削減するに従って性能向上率が緩やかに向上することがわかる．この結果は，複数パス実行を用いて分岐予測ミスを削減するに従って曖昧なメモリ依存関係を解消することの利得が向上することを意味している．Last-Value 予測の性能向上率は特徴的な傾向を示している．性能向上率は，分岐予測ミスの削減率により 8%から 17%まで変化する．ただし，性能向上率は単調に増加するわけではなく，分岐予測ミスの削減率が 0%から 20%の範囲で減少し，その後，急激に増加していく．分岐予測ミスの削減率が 90%を超えたところでは，Last-Value 予測による性能向上率が曖昧なメモリ依存関係の解消による性能向上率を上回る．20%までの部分で性能向上率が低下する理由は明らかになっていないが，分岐ミスの削減率を 100%に近づけることで Last-Value 予測の利得が向上することがわかる．

比較のために，曖昧なメモリ依存関係を排除した場合の性能向上率と Last-Value 予測の性能向上率を足し合わせた向上率を破線として図 8.3に加えた．この値と，メモリ依存予測と Last-Value 予測を同時に利用した場合の測定値の比較より，メモリ依存予測と Last-Value 予測の同時利用が，それぞれの性能向上率の和に近い性能向上率を達成することがわかる．この結果は，制御依存関係，曖昧なメモリ依存関係，真のデータ依存関係を同時に排除することがプロセッサの性能向上の有効な手段となることを意味している．

本節では，ベースラインプロセッサのシミュレータを用いて，制御依存関係，曖昧なメモリ依存関係，真のデータ依存関係を解消することによる性能向上率を測定し，3 つの依存関係を同時に削減することで高い性能向上を達成できることを確認した．以降，高レベル投機技術の融合における問題点と融合手法を議論するとともに，複数パス実行プロセッサのシミュレータを用いて，高レベル投機技術を融合した場合の性能を評価していく．

## 8.2 複数パス実行とメモリ依存予測の融合

本節では，複数パス実行とメモリ依存予測の融合を検討する．メモリ依存予測にはストアセットを用いた予測機構 [CE98] と，6.3.1節で提案したストア特定予測を利用する．複数パス実行におけるパス割り付け戦略には 5.3節で提案した JRS-PM 方式を採用する．5.4.1節で定義した複数パス実行プロセッサのシミュレータを用いて融合手法を評価する．

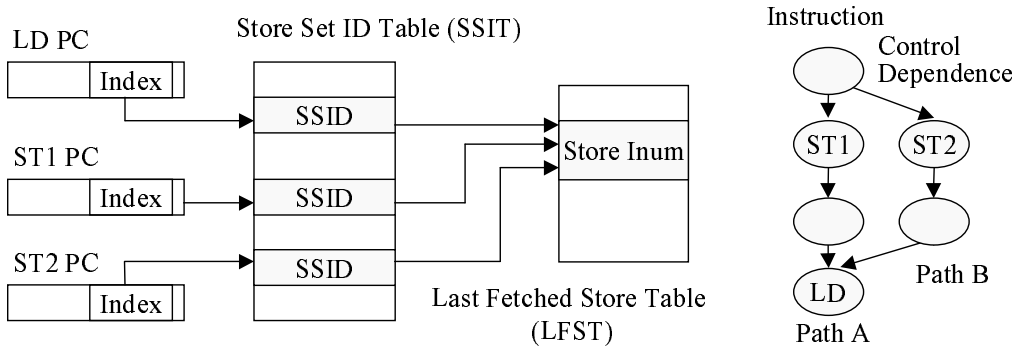



図 8.4: 複数パス実行により引き起こされるストアセットの予測ミス

### 8.2.1 複数パス実行におけるストアセットの検討

シングルパス実行では発生しないが，複数パス実行によりストアセットの予測ミスが発生する場合がある．この例を図 8.4に示す．図 8.4右に示した制御依存関係を持つプログラムの処理を考える．ここで，LD（ロード命令）, ST1（ストア命令）, ST2（ストア命令）のメモリ参照アドレスが等しく，ロード命令はこれらのストア命令にデータ依存関係を持つとする．ロード命令の過去の実行において ST1, ST2 とデータ依存関係を持つことが検出され，ロード命令のストアセットにはこれらのストア命令が登録されているとする．この時の SSIT と LFST の様子を図 8.4左に示す．ロード命令と 2 つのストア命令のプログラムカウンタをインデックスとしてアクセスする SSIT の中には，3 つのエントリに同じ SSID が格納されており，この SSID を用いて共通の LFST エントリにアクセスできるようになっている．

シングルパス実行でパス A が実行された場合には，ST1 がフェッチされた時点で，LFST には ST1 のタグが格納される．ロード命令は LFST を参照することでデータ依存関係を持つストア命令が実行中であることを検出し，ストア命令を追い越した投機処理を禁止する．これにより，ストア命令から正しい値を得ることができる．シングルパス実行でパス B が実行された場合にも，同様にしてデータ依存関係の存在を正しく予測できる．

一方，複数パス実行でパス A とパス B の両方を実行した場合には予測ミスが発生する可能性がある．これは ST1, ST2 の中で遅くフェッチされた命令が LFST のエントリを上書きし，パスが削除される時に，このエントリを無効化するためである．予測ミスが発生する様子を順番に説明する．

| Program | シングルパス | | 複数パス（パス数 8 ） | |
|---|---|---|---|---|
| | Accuracy | Miss | Accuracy | Miss |
| go | 82.1 % | 4,728 | 79.3 % | 248,301 |
| m88ksim | 69.2 % | 666 | 74.0 % | 81,890 |
| gcc | 71.5 % | 10,036 | 72.4 % | 281,014 |
| compress | 63.1 % | 212 | 56.7 % | 269,731 |
| li | 53.4 % | 750 | 53.6 % | 549,617 |
| ijpeg | 83.7 % | 1,440 | 82.9 % | 91,336 |
| perl | 68.1 % | 1,089 | 68.7 % | 260,060 |
| vortex | 59.3 % | 5,501 | 60.3 % | 158,023 |
| Average | 68.8 % | 3,053 | 68.5 % | 242,497 |

表 8.1: ストアセットの予測精度と予測ミスの回数

1. パス A の ST1 がフェッチされて，LFST のエントリに ST1 のタグが格納される．
2. パス B の ST2 がフェッチされて，LFST のエントリに ST2 のタグが上書きされる．
3. 分岐結果が判明してパス B がフラッシュされ，LFST から，ST2 のタグを持つエントリが削除される．
4. LD の依存関係が予測されるが，LFST のエントリが無効となっているために，LD がデータ依存関係を持つストア命令は実行中でないと予測され，ST1 を追い越して，LD の実行がおこなわれる．
5. LD が ST1 を追い越して実行されたことにより，予測ミスが発生する．

このように，複数パス実行プロセッサでは，LFST の状態を正しく保持できないことが原因となり予測ミスが発生する．このことを確認するために，シングルパス実行と同様に複数パス実行プロセッサにストアセットを実装し，予測精度と予測ミスの回数を測定した．ここで**予測精度**という言葉は，正しく予測できた回数と予測対象となる命令の実行回数との割合と定義する．予測の対象となる全ての命令の予測が有効となり，かつ全ての予測が正しかった場合に予測精度が 100%となる．予測精度だけではなく，予測ミスの回数を示す理由は，予測をおこなわない命令が存在するためである．

シングルパス実行と，パスの数を 8 に設定した複数パス実行プロセッサで測定したストアセットの予測精度と予測ミスの回数を表 8.1に示す．表 8.1の結果から，複数パス実行に

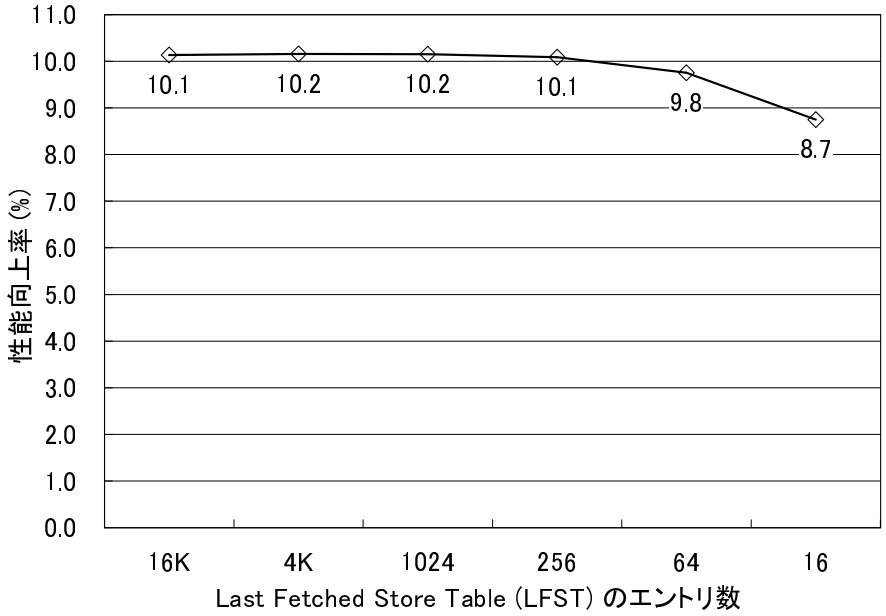



図 8.5: LFST のエントリ数を変化させた時の性能向上率

よる予測精度の低下は小さいが，予測ミスの平均回数が 80 倍程度に増加することがわかる．これらの予測ミスの増大は，ストアセットによる IPC 向上を打ち消してしまう．このため，予測ミスの増加を防ぐ何らかの手段を講じる必要がある．

### LFST の多重化

複数パス実行プロセッサで処理されているそれぞれのパスは，LFST に格納すべき異なったストア命令の集合を持つ．これらのストア命令の集合を一つの LFST で管理した点が，複数パス実行プロセッサにおいてストアセットの性能低下を引き起こす原因となっている．そこで，利用するパスの数に応じて LFST を多重化することを検討する．

まず，多重化のハードウェア・コストを見積もるために，LFST のエントリ数を変更して測定した性能向上率を図 8.5に示す．LFST のそれぞれのエントリにタグ領域は必要ない．それぞれのエントリには実行中のストアを特定できるタグ (Store Inum) を格納する．ほぼ 100%の性能を達成するために 256 エントリ，それぞれのエントリのサイズ 16 ビットを想定しても，LFST の総容量は 0.5KB となる．この値は 128KB のデータキャッシュなどと比較して非常に小さく，パス毎に LFST を用意することは現実的な選択といえる．

次に，多重化の手法を検討する．複数パスで利用するパスの数だけ LFST を複製してそ

れぞれのパスに，ひとつの LFST を割り当てる．それぞれのパスでストア命令がフェッチされた場合には，そのパスに割り当てられた LFST を更新する．不必要となったパスのフラッシュ等により，ストア命令がフラッシュされる際には，ストア命令のタグを利用して全ての LFST から当該ストアが生成したエントリを削除する．新しいパスを生成する場合には実行中のストア命令に応じて LFST を生成する．

以上の拡張により先に指摘した問題点は解決可能であり，複数パス実行プロセッサにおいても，シングルパス実行プロセッサと同様のミス率を維持できる．

### 8.2.2 予測ミスの検出

メモリ依存予測の予測ミス検出には 6.4.2節で検討した様に，アドレス比較によるミス検出と，データ値比較によるミス検出が考えられる．それぞれの検出方法は利点と欠点を持つが，次に述べる利点により，後者のデータ値比較によるミス検出を用いて評価する．

データ値比較でミスを検出する場合には，それぞれのロード命令で必要となる比較は，正しいデータ値と投機的に利用したデータ値との比較 1 回で済む．一方，アドレス比較によるミス検出では，ロード命令のメモリ参照アドレスと，ロード命令に先行されてフェッチされ実行中の $n$ 個のストア命令のメモリ参照アドレスに対して $n$ 回の比較が必要となる．サイクル当たり複数のロード命令が処理されることを考えると，アドレス比較のハードウェアが複雑化することは避けられない．この点から，特に複数パス実行プロセッサでは，データ値比較によるミス検出が適している．

### 8.2.3 予測ミスからの回復

予測ミスからの回復に関しても，6.4.3節で検討したように，後続命令のフラッシュによる回復と，後続命令の再実行による回復が考えられる．

ただし，複数パス実行プロセッサでは，複数パス実行がプロセッサのバックエンドの設計を複雑にする．メモリ依存予測によるハードウェアの複雑化を避けるために，前者の後続命令のフラッシュによる回復を採用する．再実行による回復を用いた場合の性能は今後の課題である．

### 8.2.4 評価結果

複数パス実行とストアセットを利用した場合の IPC を図 8.6にまとめた．利用できるパ

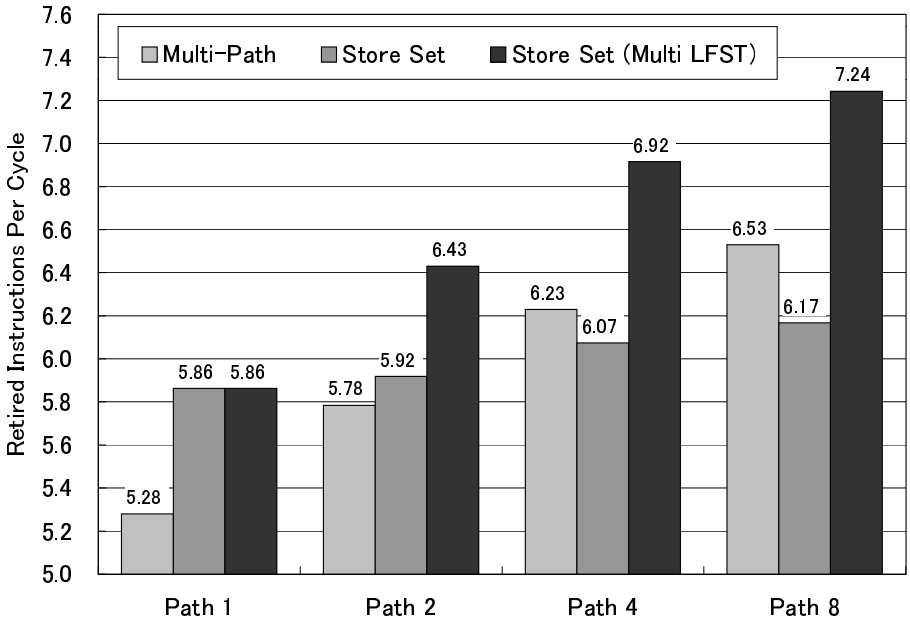



図 8.6: 複数パス実行におけるストアセットの利用

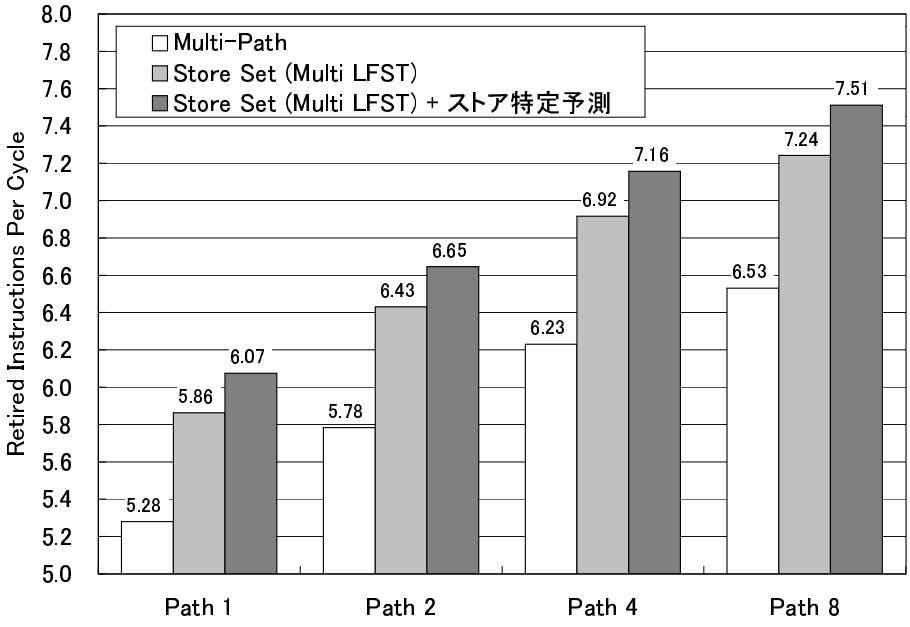



図 8.7: 複数パス実行におけるメモリ依存予測の利用

スの数を 1, 2, 4, 8 に変更して測定した．パスの数 1 はシングルパス実行を意味している．それぞれのパスには 3 つの値を示した．左から順番に，複数パス実行，複数パス実行とストアセットの融合，複数パス実行と LFST を多重化したストアセットの融合による IPC を示した．図 8.6の結果から，パスの数を 4 以上に設定した場合には，LFST を多重化しないストアセットにより性能が低下することがわかる．一方，LFST を多重化したストアセットを用いることで，0.58 から 0.71 の IPC 向上を確認できる．複数パス実行の性能を基準とした性能向上率は，パスの数にほとんど影響を受けず 11%となった．

図 8.7には，LFST を多重化したストアセットにストア特定予測を加えて測定した IPC を示す．ストア特定予測を加えることで，更に，0.21 から 0.27 の IPC 向上を達成できる．複数パス実行の性能を基準とした，ストアセットとストア特定予測のハイブリッドによる性能向上率はパスの数にほとんど影響を受けず 15%となった．この結果から，複数パス実行プロセッサにおいても，ストアセットとストア特定予測から構成されるハイブリッドのメモリ依存予測が高い成功向上率をもたらすことがわかる．

### 8.2.5 ストアセットのためのハードウェア資源

ストアセットの実装において必要となる主なハードウェアは SSIT, LFST という 2 つのテーブルである．エントリ間の競合を回避するためにエントリ数を 64K に設定した SSIT の記憶容量は 64KB，パス数 8 の時に必要となる多重化された LFST の記憶容量は 4KB である．これらのハードウェア量は，現実的な容量である．ただし，複数パス実行は，命令パイプラインの各ステージで処理される命令数を増加させる．このため，SSID, LFST の各テーブルには多くのポート数が必要となる．サイクル当たりに処理される命令数は 8.4.4 節で議論するが，ポート数を考慮したハードウェア量の見積もりは今後の課題である．

## 8.3 複数パス実行と値予測の融合

本節では，複数パス実行と値予測の融合を検討する．実装の容易さから，値予測には Last-Value 予測を利用する．複数パス実行におけるパス割り付け戦略には 5.3節で提案した JRS-PM 方式を採用する．5.4.1節で定義した複数パス実行プロセッサのシミュレータを用いて融合手法を評価する．

### 8.3.1 Last-Value 予測

Last-Value 予測の予測ミスを削減するために，次の仕組みを利用する．値履歴テーブルのそれぞれのエントリに，0 から 1023 までの値を持つ 10 ビットのカウンタを追加する．カウンタは 512 で初期化されており，前回の演算結果と今回の演算結果が等しかった場合に値をインクリメントする．演算結果が異なる場合にはカウンタの値から 64 を引く．Last-Value 予測を利用する時点で，このカウンタの値が $512 + 64 = 566$ より大きかった場合に予測を有効とする．以上のカウンタを用いることで，頻繁に予測ミスを引き起こす命令の予測を無効にすることができる．

#### 予測ミスの検出

実行ステージで正しい演算結果が得られるので，メモリアクセス・ステージで値予測のミスを検出することができる．しかし，以下の利点より，値予測ミスの検出は命令のリタイア時におこなうとする．

複数パス実行により処理される命令が増加しても，リタイアする命令数は変わらない．このため，リタイア時にミスを検出することで，検出されるミスの増加を抑えることができる．一方，メモリアクセス・ステージにおいて値予測のミスを検出する場合には，処理する命令数の増加により，検出される予測ミスの回数が増加する．また，この場合には，同一サイクルに複数のパスから値予測のミスが検出される場合があり，これを処理するハードウェアが複雑になる．本評価で採用した Last-Value 予測のミス率は非常に低いため，ミス検出をリタイア時まで遅らせることによる性能低下は非常に小さい．

#### 予測ミスからの回復

予測ミスによるペナルティを抑えるためには，間違った値を利用した命令のみを再実行すればよい [佐藤 98] ただし，この手法を用いた場合には，発行された命令に関しても再実行を可能とするために，命令ウィンドウに必要な情報を格納しておく必要があり，これを実現するためのハードウェアコストは大きい．このため，本評価では，後続命令のフラッシュにより値予測ミスから回復する．

本評価で用いる，リタイア時の予測ミス検出と後続命令のフラッシュによる予測ミスからの回復は，最もシンプルなミス検出とミスからの回復手法である．本節の目的は，このシンプルな手法においてさえ 10%を超える性能向上が可能となることを示すことにある．更なる性能改善の検討は今後の課題である．

| Program | シングルパス | | 複数パス (パス数 8) | |
|---|---|---|---|---|
| | Accuracy | Miss | Accuracy | Miss |
| go | 22.2 % | 13,247 | 22.2 % | 13,246 |
| m88ksim | 56.9 % | 5,930 | 56.9 % | 5,930 |
| gcc | 18.9 % | 35,874 | 18.9 % | 35,870 |
| compress | 24.6 % | 5,481 | 24.6 % | 5,481 |
| li | 24.1 % | 35,549 | 24.1 % | 35,549 |
| ijpeg | 11.5 % | 21,055 | 11.5 % | 21,055 |
| perl | 29.4 % | 2,322 | 29.4 % | 2,322 |
| vortex | 34.2 % | 21,961 | 34.2 % | 21,961 |
| Average | 27.7 % | 17,677 | 27.7 % | 17,676 |

表 8.2: Last-Value 予測の予測精度と予測ミスの回数

### 8.3.2 評価結果

パスの数を 1 と 8 に変更して測定した Last-Value 予測の予測精度と予測ミスの回数を表 8.2にまとめる．この結果から，Last-Value 予測は複数パス実行の影響をほとんど受けず，全実行命令の 28%の演算結果を正しく予測できることがわかる．

複数パス実行と Last-Value 予測を融合した場合の IPC を図 8.8に示す．パスの数を 1, 2, 4, 8 に変更して IPC を測定した．Last-Value 予測を利用することで 0.55 から 0.80 の IPC 向上を確認できる．複数パス実行の性能を基準とした性能向上率では，パスの数 1, 2, 4, 8 についてそれぞれ，10.6%, 11.4%, 11.9%, 12.3%と，パスの数を増加するに従って性能向上率が向上する．この結果は，複数パス実行が Last-Value 予測の効果をより効果的に引き出すことを意味している．

### 8.3.3 値予測を追加するためのハードウェア資源

命令のリタイア時に予測ミスを検出するために，Last-Value 予測のミスはサイクル当たり 1 回しか発生しない．更に，後続命令のフラッシュにより予測ミスから回復する場合には，予測ミスの際に実行中の全ての命令をフラッシュしてかまわない．これは，分岐ミスからの回復と同じ機構で実現できるため，予測ミスからの回復のためのハードウェアの追加を抑えることができる．

値予測を実現するためのハードウェアの中で最も深刻となるものは，予測で利用する履歴テーブルの容量とポート数である．履歴テーブルのエントリ数を変化させて予測精度を

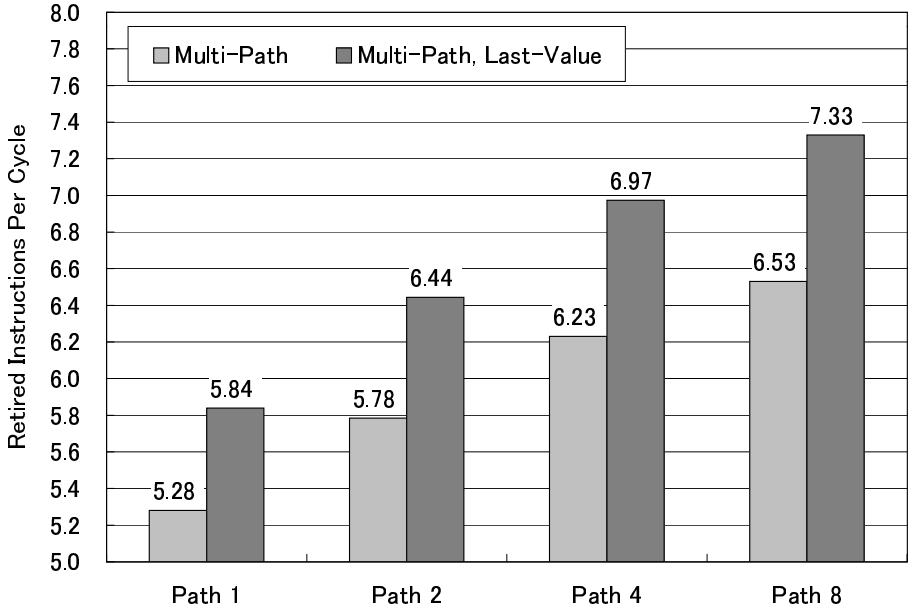



図 8.8: 複数パス実行と Last-Value 予測を同時に利用した際の IPC

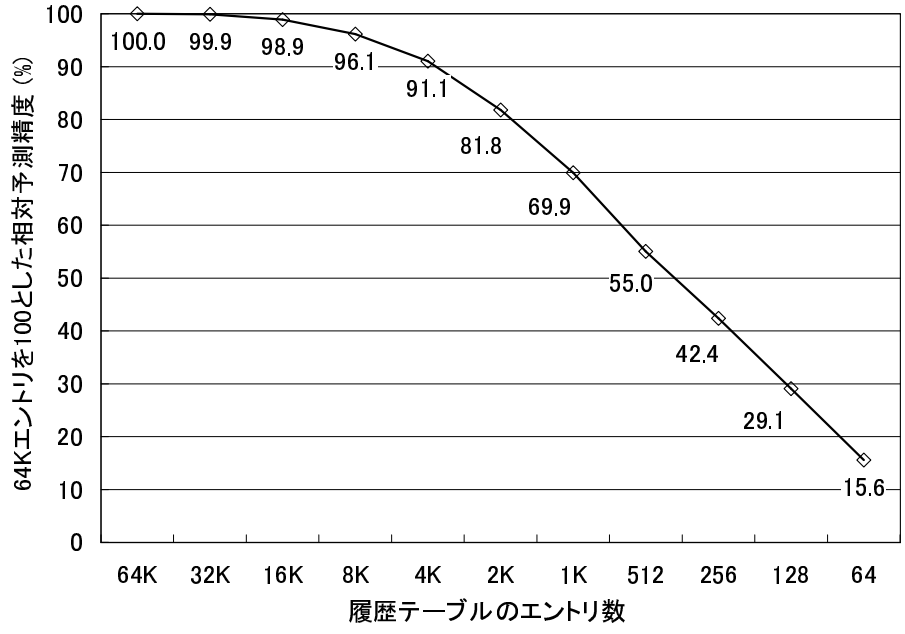



図 8.9: Last-Value 予測の履歴テーブルのエントリ数を変化させた際の予測精度

測定した結果を図8.9に示す．図では64Kエントリにおける予測精度を100%とした相対予測精度を示した．なお，64Kエントリではほとんどテーブルの競合は発生していないので，この性能はエントリ無限大の場合の性能と考えてよい．図8.9の結果から，8Kエントリの履歴テーブルを用いることで96%の予測精度を達成できることがわかる．2ウェイ・セットアソシアティブで，それぞれのエントリに20バイトを必要とすると，8Kエントリの記憶容量は，$8K$エントリ$\times 20B = 160KB$となり，データキャッシュと同様のハードウェア量で値履歴テーブルを実現できる．

Last-Value予測で必要となる履歴テーブルの容量は現実的であるが，複数パス実行プロセッサでは，サイクル当たりに処理する命令数の増加により，より多くのポート数を用意しなければならない．ポート数を含む検討は今後の課題である．

## 8.4 3つの高レベル投機技術の融合

### 8.4.1 高レベル投機技術の融合手法の提案

複数パス実行，メモリ依存予測，値予測という3つの高レベル投機技術の融合手法を次に定義する．

**JRS-PM方式の採用** 複数パス実行のパス割り付け戦略として確率モデル，Selective Eager Execution, JRS-PM方式を評価した5.5節の結果より，JRS-PM方式における性能が最も高いことが確認されている．本融合手法ではJRS-PM方式を利用する．

**LFSTを多重化するストアセットとストア特定予測の採用** メモリ依存予測の性能を評価した6.5.3節の結果より，LFSTを多重化するストアセットとストア特定予測のハイブリッド予測により高い性能向上を達成できることが明らかになっている．本融合手法では，このハイブリッドによるメモリ依存予測を利用する．

**10ビットのカウンタを利用するLast-Value予測の採用** 2レベル・ストライド値予測により，Last-Value予測より数%高い性能向上率を達成できることが判明している．しかしながら，7.5節で議論した様に，2レベル・ストライド値予測はリオーダバッファに拡張が必要となり，プロセッサのバックエンドの設計を複雑にする恐れがある．このため，本融合手法では，より現実的な手法としてLast-Value予測を利用する．Last-Value予測の予測ミスを削減するために，8.3.1節で説明した，10ビットのカウンタを用いた確信度評価を利用する．

**メモリ依存予測ではデータ値比較によるミス検出を採用** アドレス比較によるミス検出と比較して，検出するために必要となる比較の回数が少ないという利点と，予測値と正しい値が一致している限り処理を継続できるという利点から，本融合手法では，データ値比較によりメモリ依存予測の予測ミスを検出する．

**命令のリタイア時に予測ミスを検出** 命令のリタイア時に値予測とメモリ依存予測の予測ミスの検出をおこなう．これにより，フラッシュされる命令に対するミス検出を排除できると共に，検出されるミスの数をサイクル当たり 1 回に制限できる．メモリアクセス・ステージでミスを検出する場合には，実行されている複数のパスにおいて，同一サイクルに複数のミスが検出されることがあり，これらを処理するハードウェアが必要となる．

**後続命令のフラッシュによる予測ミスからの回復を採用** 間違った値を利用した命令のみを再実行することで，予測ミスのペナルティを削減できる．ただし，再実行による回復を実現するためには命令ウィンドウを拡張する必要があり，プロセッサのバックエンドの設計を複雑にする可能性がある．このため，本融合手法では，後続命令のフラッシュにより値予測とメモリ依存予測の予測ミスから回復する．

メモリ依存予測の対象はロード命令であるが，値予測の対象は演算結果を生成するすべての命令であり，この中にはロード命令も含まれる．このため，ロード命令に関して，値予測とメモリ依存予測の両方が利用できるという状況が発生する．この場合には以下の戦略の利用が考えられる．

**値予測を利用し，メモリ依存予測を無効化** この設定では，パイプラインにおけるロード命令の処理のタイミングは，予測をおこなわない場合と変わらない，すなわち，先行する全てのストア命令のアドレス計算が終了した後に，ロード命令は発行される．ただし，値予測により演算結果が予測されるので，ロード命令にデータ依存関係を持つ命令は，予測された値を用いて処理を開始できる．

**メモリ依存予測を利用し，値予測を無効化** メモリ依存予測を利用する場合には，先行するストア命令のアドレス計算を待たずにロード命令の処理を開始できる．しかし，ロード命令にデータ依存関係を持つ命令は，ロード命令がメモリアクセス・ステージを終了した際にデータを受けとる必要がある．

**値予測とメモリ依存予測の両方を利用** 値予測とメモリ依存予測を同時に利用することで，上に示したの 2 つの利点を利用できる．

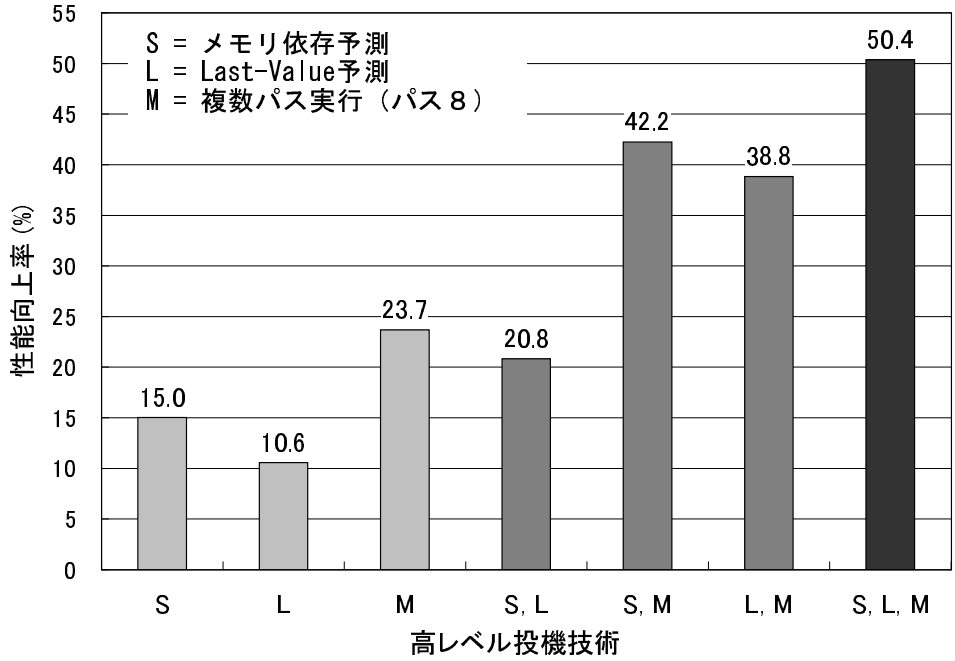



図 8.10: 投機技術の同時利用による性能向上

ロード命令に関して以上の選択の自由が発生するが，2つの予測を同時に利用することによる性能低下は発生しないと考え，3つ目の値予測とメモリ依存予測の両方を利用する設定を採用する．

### 8.4.2 性能向上率

複数パス実行 (M)，メモリ依存予測 (S)，Last-Value予測 (L) という3つの高レベル投機技術の組合せによる性能向上率を図 8.10にまとめる．本評価では，複数パス実行で利用できるパスの数を8に固定する．図 8.10の縦軸はベースラインプロセッサの IPC 5.28 に対する性能向上率を示している．図 8.10の左の3本のデータは，メモリ依存予測，Last-Value予測，複数パス実行をそれぞれ単独で利用した場合の性能向上率を示している．次の3本のデータは2つの高レベル投機技術の融合による性能向上率を示している．右端のデータは，3つの高レベル投機技術の融合による性能向上率を示している．

図 8.10の結果から，複数パス実行，メモリ依存予測，Last-Value予測それぞれの単体利用では 23.7%, 15.0%, 10.6%の性能向上率しか得られないが，3つの高レベル投機技術の融合により 50.4%という高い性能向上率を達成できることがわかる．3つの高レベル投機技

| Program | メモリ依存予測 | | Last-Value 予測 | |
|---|---|---|---|---|
| | 予測精度 | ヒット率 | 予測精度 | ヒット率 |
| go | 88.66 % | 99.98 % | 22.16 % | 99.96 % |
| m88ksim | 96.01 % | 99.99 % | 56.98 % | 99.99 % |
| gcc | 81.21 % | 99.96 % | 18.88 % | 99.88 % |
| compress | 71.64 % | 99.99 % | 24.64 % | 99.98 % |
| li | 78.78 % | 99.99 % | 24.13 % | 99.92 % |
| ijpeg | 90.49 % | 99.99 % | 11.53 % | 99.89 % |
| perl | 88.55 % | 99.99 % | 29.40 % | 99.99 % |
| vortex | 90.11 % | 99.98 % | 34.17 % | 99.96 % |
| Average | 85.68 % | 99.99 % | 27.73 % | 99.94 % |

表 8.3: 3 つの投機技術の融合時の予測精度とヒット率

術を同時に利用した場合の IPC は 7.9 となった．この時の性能向上率はそれぞれの投機技術による性能向上率の合計 $23.7 + 15.0 + 10.6 = 49.3\%$ を上回ることに注意して欲しい．この結果は，制御依存関係，曖昧なメモリ依存関係，真のデータ依存関係のそれぞれを同時に解消することの有効性を示している．

3 つの高レベル投機技術の融合時におけるメモリ依存予測と Last-Value 予測の予測精度とヒット率を表 8.3にまとめる．**予測精度**は，正しく予測できた回数 $(Hit)$ と予測対象となる命令の実行回数 $(Candidate)$ との割合 $Hit/Candidate$ で定義する．メモリ依存予測の予測精度は 85.7%，Last-Value 予測の予測精度は 27.7%という結果を得た．**ヒット率**は，予測に成功した回数 $(Hit)$ と予測に失敗した回数 $(Miss)$ を用いて $Hit/(Hit + Miss)$ で定義する．メモリ依存予測のヒット率は 99.99%，Last-Value 予測のヒット率は 99.94%となった．この様に，本融合手法で用いた予測手法の予測ヒット率は非常に高い．この高いヒット率が，融合時の性能向上を可能とする理由の一つとなっている．

### 8.4.3 融合手法におけるハードウエアの複雑さ

メモリ依存予測が有効となり，実行中のストア命令にデータ依存関係が無いと予測されるロード命令は，その実行においてストアキューを参照する必要はない．このため，ストアキューに要求されるポート数増加の問題を軽減できる可能性がある．

値予測を用いることで，プロセッサのフロントエンド部で供給できるオペランドの割合が増加する．このため，命令ウィンドウで待ち合わせるオペランドの数を削減できる．

以上は，高レベル投機技術の同時利用により発生する利点である．

さらに，提案した融合手法では，後続命令のフラッシュにより予測ミスから回復する．再実行による回復と異なり，この手法では，命令ウィンドウのフィールドの追加といった拡張を必要としない．また，リタイア時にミスを検出することで，フラッシュされる命令に対するミス検出を排除できると共に，検出されるミスの数をサイクル当たり1回に制限できる．メモリアクセス・ステージでミスを検出する場合には，実行されている複数のパスにおいて，同一サイクルに複数のミスが検出されることがあり，これらを処理するハードウェアが必要となる．

高い命令レベル並列性を利用するプロセッサでは，大規模化に伴い，命令発火機構とストアキューの複雑さが増加する．提案した融合手法において，後続命令のフラッシュにより予測ミスから回復する，リタイア時に予測ミスを検出する，メモリ依存予測の予測ミスをデータ値比較により検出する、という構成を用いたことで，値予測とメモリ依存予測の追加が命令発火機構とストアキューの複雑さを増加させないという点は重要である．高レベル投機技術の融合がプロセッサの複雑さに与える影響に関しては，今後，詳細な検討が必要である．

### 8.4.4 命令トラフィックの増加

複数パス実行は，各パイプライン・ステージで処理される命令数を増加させる．処理される命令数の増加は，予測のためのテーブル (STIT, LFST, 値履歴テーブル) で必要となるポート数を増加させる．

3つの高レベル投機技術の融合時において，それぞれのパイプラインステージで処理された命令数を図8.11にまとめる．図の横軸は各パイプラインステージを示し，縦軸はそれぞれのステージで処理される平均命令数を示している．

図8.11の結果から，プロセッサのフロントエンドでは処理される命令数が一定の割合で減少することがわかる．オペランド・フェッチと実行ステージの間ではデータの待ち合わせによる待ちサイクルが挿入されるため，実行される処理量は更に減少する．フェッチした命令数42と比較し，実行した命令は55%の23命令となった．図8.11に示した結果は，処理される平均命令数であるため，実行機構では，23を超える数の実行ユニットを実装する必要がある．

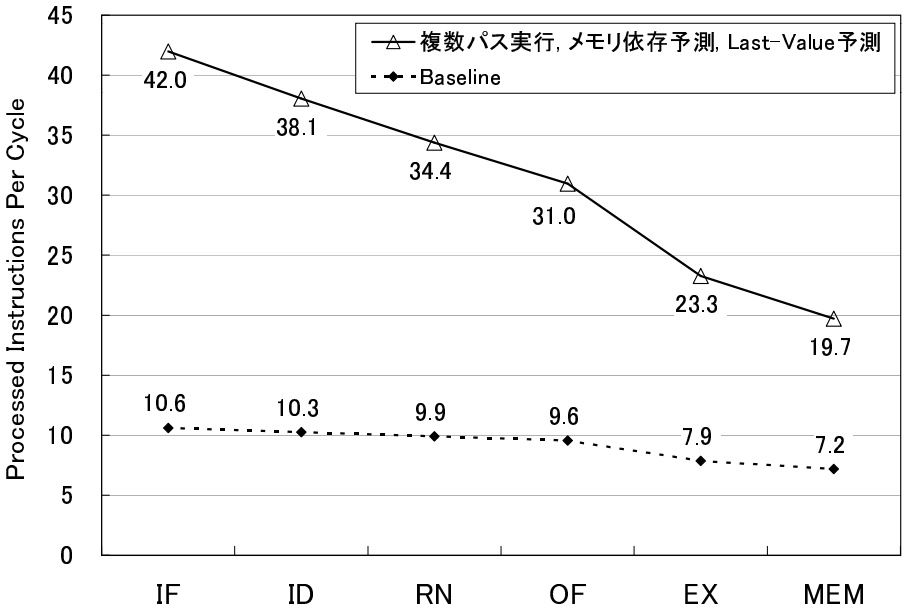



図 8.11: 各パイプラインステージで処理される平均命令数

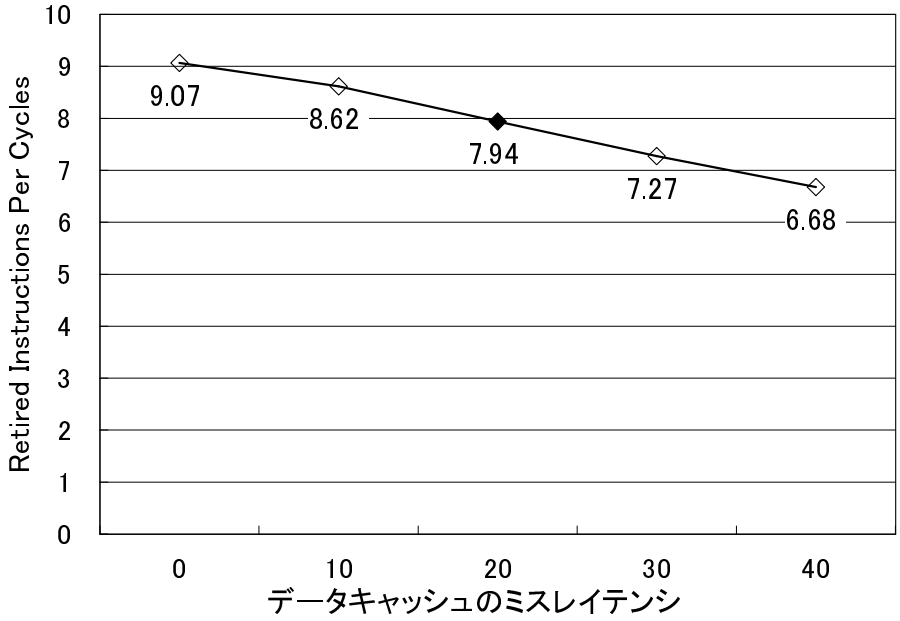



図 8.12: データキャッシュのミスレイテンシの影響

### 8.4.5 データキャッシュの影響

以上の評価はデータキャッシュにミスした際のペナルティを 20 サイクルに固定したものであった．このペナルティを 0 サイクルから 40 サイクルまで変化させて測定した IPC を図 8.12に示す．ペナルティ0サイクルの構成は，データキャッシュのミスが発生しない場合の性能を示している．

図 8.12の結果より，データキャッシュを理想化した場合の IPC は 8.9に達することがわかる．ミスペナルティ20サイクルの設定と比較し，ミスペナルティを 0 にすることで 15%の性能向上を確認した．

## 8.5 今後の課題

### 8.5.1 予測テーブルにおけるポート数の確保

メモリ依存予測と値予測は，フェッチ時に予測結果を生成しなければならない．このため，3 つの高レベル投機技術の融合時には，サイクル当たり平均 42 命令という多くの命令に対する予測が必要となる．ロード・ストア命令の実行頻度は 36%であり，これらの値からメモリ依存予測の対象となる命令数を概算すると，サイクル当たり 15 個のロード・ストア命令がフェッチされることになる．メモリ依存予測における予測テーブルではこれに見合うポート数を確保しなければならない．Last-Value 予測に関しても，値を生成し値予測の対象となる命令の実行頻度 70% を用いて概算すると，サイクル当たりに平均 29 回の予測が必要となる．ただし，以上の概算には平均の処理命令数を用いており，ピーク時には数倍の命令が処理される．先に議論した予測のためのテーブルの記憶容量に加え、これらのポート数を考慮した実現可能性の議論が必要となる．

### 8.5.2 より積極的な融合手法の検討

本評価では，値予測として Last-Value 予測を利用した．7章で議論したようにストライド値予測や 2 レベル・ストライド値予測を用いることで更に数%の性能向上を期待できる．

検討した融合手法では，値予測とメモリ依存予測における予測ミスの検出を命令のリタイア時におこない，予測ミスからの回復は命令のフラッシュにより実現した．予測ミスの早期検出や，間違った予測結果を利用した命令の再実行によるミス回復により，予測ミスのペナルティを削減できる．

これら，より積極的な手法の利用は今後の課題である．

### 8.5.3 消費電力の制御

市場では，プロセッサの性能向上と同時に消費電力の低減に対する要求が増大している．本節では，高レベル投機技術を用いた複数パス実行プロセッサにおける消費電力の制御手法を検討する．

高レベル投機技術のためのハードウェアを追加したとしても，状況に応じて追加したハードウェアの利用を禁止することで，消費電力の増加を抑えることが可能である．このために，プロセッサの内部回路を多くの区分に分割し，それぞれの区分の動作モードを状況に応じて切替える．

メモリ依存予測と値予測において投機技術の利用を禁止するためには，予測結果を生成する部分において予測が無効となるように出力を固定すればよい．また，この場合には，予測ミスの検出と回復のための回路を動作させる必要はない．複数パス実行では，利用するパスの数を制御することで必要ないパスのためのハードウェアにおける消費電力を抑えることができる．

投機技術の利用状態の変更は，例えば，利用する投機技術と規模を指定する命令の追加により実現できる．実行中のシステムの負荷をオペレーティング・システムが判断し，プロセッサのアイドル時間が多い場合に投機技術の利用を禁止すればよい．また，ユーザがプログラムの高速動作を希望する時に，明示的に投機技術の利用を指定する方法も考えられる．

アーキテクチャのレベルでは，状況に応じた高レベル投機技術の利用切替えにより，消費電力の調整が可能である．ただし，プロセッサの消費電力は，プロセッサを実装する際に利用する半導体技術や，グローバル・クロックの有無，動作モードを指定する区分の分割方法などに依存する．消費電力の見積もりに関しては，これらを考慮した詳細な検討が必要である．

### 8.5.4 高レベル投機技術を支援する静的手法の可能性

本研究では，従来の RISC コードとの互換性を維持することを前提として，より高い命令レベル並列性の利用を可能とする高レベル投機技術と，その融合手法を議論してきた．

コンパイラやプログラマによる支援により，ハードウェアの複雑化の軽減と，より高い性能を達成できる可能性がある．本節では，高レベル投機技術を支援する静的手法の可能

性を議論する.

### 予測成功の確率が高い命令の選別

Last-Value予測を例に考えると，その予測精度は28%である．一方，値を生成する命令の実行頻度は70%程度であり，約42%の命令はLast-Valu予測が成功しないと判断されていることになる．これら予測が成功しないと判断される命令に関しても値履歴テーブルを参照して予測結果を生成する必要がある.

コンパイラ等により，Last-Value予測が当たりそうな命令に印を付けることができれば，印の付いていない命令を予測の対象から除外することができ，値履歴テーブルで必要となるポート数を削減することができる.

このように，静的な情報を用いて予測の対象を絞り込むことでサイクル当たりに予測すべき命令の数を削減し，ハードウェアの負担を軽減できる可能性がある.

### 予測成功時における利得の見積もり

投機実行では，予測を用いて制約を解消し，予測が正しいという仮定のもとでハードウェアを投資する．この時，投資すべきハードウェアが豊富に存在する場合には問題ないが，実際には，プロセッサ内で処理される命令の数が増加するに従って，投機の対象となる命令の数が投資可能な命令の数を上回るといった状況が発生する．この状況は，特に複数パス実行を利用する場合に顕著となる.

このような場合には，限られたハードウェア資源を用いて，できる限り高い性能向上が得られるような優先度により投資する命令を選択しなければならない.

例えば，値予測により得られたデータが直ちに必要ない場合には，この値予測の優先度は低く設定したい．同様に，メモリ依存予測を用いてロード命令の処理を早期に開始したとしても，ロードした値を直ちに利用しない場合には，この投機による性能向上は期待できない．このような場合における投機の優先度を低く設定したい.

これらの優先度をハードウェアで設定する場合には，複雑なアルゴリズムの利用は困難であるし，そのために新たなハードウェアを追加する必要がある．一方，コンパイラやプログラマがこれらの優先度を設定する場合には，例えば，データフロー解析を用いた複雑なアルゴリズムを利用して，精度の高い優先度を設定できる可能性がある.

## 8.6 本章のまとめ

本章では，複数パス実行，メモリ依存予測，値予測という 3 つの高レベル投機技術の融合手法を議論した．

ベースラインプロセッサにおける IPC 5.28 に対して，複数パス実行による性能向上率が 23.7%，メモリ依存予測による性能向上率が 15.0%，値予測による性能向上率が 10.6%となることを示した．また，これらの高レベル投機技術の融合手法を提案し，3 つの高レベル投機技術の融合により 50.4%という高い性能向上率を達成できることが明らかになった．この結果から，制御依存関係，曖昧なメモリ依存関係，真のデータ依存関係のそれぞれを同時に解消することの有効性を確認できた．

高い命令レベル並列性を利用するプロセッサでは，大規模化に伴い，命令発火機構とストアキューの複雑さが増加する．本章で提案した融合手法において，値予測とメモリ依存予測の追加がこれらの複雑さを増加させないことを議論した．

3 つの高レベル投機技術の中で，特に，複数パス実行は高い性能向上をもたらすが，一方では，命令パイプラインの各ステージで処理される命令数を倍増させる．3 つの高レベル投機技術の融合時には，サイクル当たり平均 42 命令がフェッチされる．値予測やメモリ依存予測の予測機構は，これら多くの命令に対して予測をおこなわなければならない．これら処理命令数の増加を考慮した実現可能性の検討は今後の課題である．

# 第 9 章

# 結論

本研究では，既存の RISC プロセッサとのコード互換性を維持しながら，より高い命令レベル並列性の利用を可能とする高レベル投機技術を議論した．制御依存関係を解消する複数パス実行，曖昧なメモリ依存関係を解消するメモリ依存予測，真のデータ依存関係を解消する値予測の各手法の議論に加えて，3つの高レベル投機技術の融合手法を議論し，以下の知見を得た．

一つ目の投機技術として複数パス実行を議論した．従来からおこなわれている単一パスの実行では，分岐結果を予測し，予測された単一のパスにおける命令列を投機的に実行する．一方，全ての分岐命令において分岐成立と分岐不成立の両方のパスを実行する Eager Execution では，多くの分岐命令を跨いでパスを展開するに従って，必要となるハードウェア資源が爆発的に増加する．このため，現実的なハードウェア量を考慮した場合には，パスを複製する分岐命令を選択するパス割付け戦略が複数パス実行の性能を決める重要な要素となる．提案されている幾つかのパス割付け戦略を比較検討した上で，各手法の利点を組み合わせる JRS-PM 方式を提案した．複数パス実行プロセッサのシミュレータを用いて，命令レベル並列性，各パイプライン・ステージで処理される命令トラフィック，データキャッシュのヒット率を測定した．評価結果より，利用できるパスの数を 16 に設定することで 59%の分岐ミスペナルティを削減できることが明らかになった．この時，サイクル当たりの平均フェッチ命令数は 43，平均実行命令数は 21 というデータを得た．これらの評価により複数パス実行の可能性が一部確認された．

二つ目の投機技術として，曖昧なメモリ依存関係を解消するメモリ依存予測を議論した．まず，曖昧なメモリ依存関係がプロセッサ性能に与える影響を測定し，曖昧なメモリ依存関係を解消することで 13%から 18%の性能向上を達成できることを示した．これまでに提案されているストアセットを用いたメモリ依存予測は，実行中のストア命令にデータ依存

関係を持たないロード命令を予測する．しかし，32%のロード命令は実行中のストア命令にデータ依存関係を持ち，ストアセットだけでは曖昧なメモリ依存関係の 68%しか解消することはできない．このため，ロード命令が実行中のストア命令にデータ依存関係を持つ場合に，複数のストア命令から，データ依存関係を持つストア命令を特定するストア特定予測を提案した．シミュレーションによる評価により，ストアセットを用いた予測手法と，ストア特定予測を同時に利用することで 85%の曖昧なメモリ依存関係を解消できることが判明した．

三つ目の投機技術として，真のデータ依存関係を解消する値予測を議論した．これまでの研究では，予測する命令の前回の演算結果を今回の予測値として利用する Last-value 予測や，過去の 2 回の演算結果の差分と前回の演算結果の和を予測値とするストライド値予測などが検討されてきたが，これらの予測機構には予測ミスの回数が多いという欠点がある．この欠点の解消を目指して，2 レベル・ストライド値予測を提案した．2 レベル・ストライド値予測では，ストライド値予測でミスした際のグローバルな条件分岐命令の履歴を保存し，同様のミスの発生を抑えることで，予測ミスの削減と正しく予測できる割合の向上を目指す．シミュレーションにより提案手法を評価し，予測を有効とする確信度評価の閾値を 8 に設定した場合に，最も高い 14%の性能向上を確認した．この時，全実行命令の 34%の命令の演算結果を正しく予測できることが判明した．

提案した 3 つの高レベル投機技術は，制御依存関係，曖昧なメモリ依存関係，真のデータ依存関係，という異なる依存関係を解消する．これらの高レベル投機技術は，それぞれの単体利用によりプロセッサの性能向上を達成できるが，複数の高レベル投機技術を同時に利用することで，更なる性能向上が可能となる．本研究では，これらの高レベル投機技術の融合手法を提案し，複数パス実行プロセッサのシミュレータを用いて，その性能を評価した．その結果，複数パス実行，メモリ依存予測，値予測を単体で利用した場合には，それぞれ 23%, 15%, 10%の性能向上率しか得られないが，3 つの高レベル投機技術の融合により 50%という高い性能向上率を達成できることが明らかになった．

3 つの高レベル投機技術の中で，特に，複数パス実行は高い性能向上をもたらすが，一方では，命令パイプラインの各ステージで処理される命令数を倍増させる．3 つの高レベル投機技術の融合時には，サイクル当たり平均 42 命令がフェッチされる．値予測やメモリ依存予測は，これら多くの命令に対して予測をおこなわなければならない．これら処理命令数の増加を考慮した実現可能性の検討は今後の課題である．また，高レベル投機技術の融合がプロセッサの複雑さに与える影響に関しては，詳細な検討が必要である．

# 謝辞

指導教官である田中英彦教授には, 修士・博士過程 5 年間の長きに渡って指導していただきました. さらに，研究者としての心構えなど様々なことを学ばせていただきました. このように博士論文を書き終えることが出来たのは，田中教授の御指導のおかげです．ここに深い感謝の意を表します.

坂井修一助教授には, 研究全般に渡って的確なアドバイスを多数いただきました. さらに, 研究の進め方から細部に至るまで丁寧に指導していただきました．ここに深く感謝致します.

井上博允教授, 武市正人教授, 喜連川優教授，南谷崇教授には, 審査においてさまざまな有益なアドバイスをいただきました. ここに深く感謝致します.

富士通研究所の佐藤充博士には, プロセッサの基礎から丁寧に説明していただきました. また，研究を進める上で的確な助言を数多く頂きました.

日立中央研究所の中村友洋博士には，修士・博士過程に渡って VLDP プロジェクトをまとめ，研究の方向性から細部に至るまで熱心に指導していただきました．プロセッサの研究を続けることができたのも氏の指導力のおかげです．ここに深く感謝致します.

NEC C&C メディア研究所の荒木拓也博士，NEC 通信システムの渡辺正泰氏には，日頃からの議論を通して様々なアドバイスをいただきました．また，叱咤激励していただき正しい方向へと導いていただきました.

本研究の多くのアイデアは VLDP ミーティングの白熱した議論がもととなっています. より刺激的な研究生活を送ることができたのも VLDP プロジェクトのメンバーの皆様のおかげです．皆様には，深い感謝の意を表します．特に辻秀典氏，安島雄一郎氏，田中洋介氏，入江英嗣氏には深く感謝致します.

1 年間ではありますが，研究を共にした日立製作所の斎藤英一氏には，シミュレータの一部の実装とその詳細な検討を熱心に取り組んでいただきました．ここに深く感謝いたします.

コンパイラ・グループのメンバーの皆様，飯塚大介氏，中村実氏，服部直也氏には，日

頃のディスカッションを通じて，ソフトウェアから見た様々な可能性を議論することができました．

また，他の SIGIE (Special Interest Group on Inference Engine) のメンバーの皆様，その中でも特に，馬場恒彦氏，Antonio Magnaghi 氏，滝田裕氏からはさまざまな助言を頂きました．ここに深く感謝致します．

また，研究室で論文執筆の苦労を共にした井手一郎氏，五十嵐健夫氏，木下智義氏とは，異なる分野の研究者として興味深い議論を交わすことができました．

他にも，田中・坂井研究室の皆様には，打合せの際だけでなく，普段からたくさんの貴重なアドバイスをいただきました．これまでなんとか研究を続けて来ることが出来たのは皆様のおかげだと思っております．本当にありがとうございました．

最後に，これまでの研究を支えてくれた家族に深く感謝いたします．

# 発表文献


[1] 吉瀬謙二. 多数演算器方式のプロセッサ構成 に関する研究. 修士論文, 東京大学大学院 工学系研究科, 1997.

[2] 吉瀬謙二, 坂井修一, 田中英彦. 2レベル・ストライド値予測機構の可能性検討. 情報処理学会論文誌 並列処理特集号（採録済）.

[3] 吉瀬謙二, 坂井修一, 田中英彦. 複数パス実行におけるパス割付け戦略の検討. 並列処理シンポジウム JSPP2000（査読中）.

[4] 吉瀬謙二, 坂井修一, 田中英彦. マルチレベル・ストライド値予測機構による命令レベル並列性の向上. 並列処理シンポジウム JSPP'99 論文集, pp. 119–126, June 1999.

[5] 吉瀬謙二, 中村友洋, Antonio Magnaghi, 辻秀典, 安島雄一郎, 高峰信, 坂井修一, 田中英彦. 新しいアーキテクチャ Very Large Data Path. 並列処理シンポジウム JSPP'98 論文集, p. 155, June 1998.

[6] 吉瀬謙二, 中村友洋, 辻秀典, 安島雄一郎, 田中英彦. 多数演算器方式における演算器利用率の検討. 情報処理学会研究報告 計算機アーキテクチャ研究会, 97-ARC-125, Vol. 97, No. 76, pp. 121–126, August 1997.

[7] 吉瀬謙二, 斎藤英一, 入江英嗣, 坂井修一, 田中英彦. ストアキューの拡張によるロードトラフィックの削減方式. 情報処理学会第 59 回全国大会, Vol. 1, No. 3H-4, pp. 43–44, September 1999.

[8] 吉瀬謙二, 高峰信, 田中洋介, 坂井修一, 田中英彦. ショート・リブド・データの動的な予測に関する検討. 情報処理学会第 58 回全国大会, Vol. 1, No. 2H-6, pp. 155–156, March 1999.

[9] 吉瀬謙二, 中村友洋, 辻秀典, 安島雄一郎, 高峰信, 坂井修一, 田中英彦. 命令ウィンドウの動的最適化. 情報処理学会第 57 回全国大会, Vol. 1, No. 1Q-04, pp. 28–29, October 1998.

[10] 吉瀬謙二, 中村友洋, 辻秀典, 安島雄一郎, 田中英彦. ALU-Net を用いることによるデータ移動の効率化. 情報処理学会第 56 回全国大会, Vol. 1, No. 2N-1, pp. 109–110, March 1998.

[11] 吉瀬謙二, 田中英彦. 多数演算器方式における演算器利用率の向上手法. 情報処理学会第 55 回全国大会, Vol. 1, No. 3F-1, September 1997. 3F-01.

[12] 吉瀬謙二, 中村友洋, 辻秀典, 田中英彦. 制御依存関係による並列度利用の限界に関する定量的評価. 情報処理学会第 53 回全国大会, Vol. 6, No. 4F-1, pp. 81–82, March 1996.

[13] 吉瀬謙二, 中村友洋, 金指和幸, 田中英彦. データフローグラフを用いた複数命令のブロック化. 情報処理学会第 52 回全国大会, Vol. 6, No. 1L-5, pp. 81–82, March 1996.

[14] 吉瀬謙二, 中村友洋, 金指和幸, 田中英彦. データフローグラフ変換による並列度抽出. 情報処理学会第 51 回全国大会, Vol. 6, No. 1P-2, pp. 73–74, September 1995.

# 参考文献


[Ass99] Semiconductor Industry Association. *International Technology Roadmap for Semiconductors 1999 Edition.* Semiconductor Industry Association, San Jose, Calif, 1999.

[CE98] George Z. Chrysos and Joel S. Emer. Memory Dependence Prediction using Store Sets. In *In Proceedings of the International Symposium on Computer Architecture*, pp. 142–153, 1998.

[Che98] Tien-Fu Chen. Supporing Highly Speculative Execution via Adaptive Branch Trees. In *In Proceedings of HPCA*, pp. 185–184, October 1998.

[CW99] Lucian Codrescu and Scott Wills. Improving Value Prediction Accuracy with Global Correlation. Technical report, School of Electrical and Computer Engineering, Georgia Institute of Technology, 1999.

[DH98] Karel Driesen and Urs Holzle. Accurate Indirect Branch Prediction. In *In Proceedings of the International Symposium on Computer Architecture*, pp. 167–178, 1998.

[ECP96] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In *In Proceedings of the International Symposium on Computer Architecture*, pp. 3–11, 1996.

[FJMs96] Federico Faggin, Marcian E. Hoff Jr, Stanley Mazor, and Masatoshi shima. The History of the 4004. *IEEE Micro*, Vol. 16, pp. 10–20, 1996.

[GG98] Jose Gonzalez and Antonio Gonzalez. The Potential of Data Value Speculation to Boost ILP. In *In Proceedings of the International Conference on Supercomputing*, pp. 221–228, 1998.

[GKMP98] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence Estimation for Speculation Control. In *In Proceedings of the International Symposium on Computer Architecture*, pp. 122–131, 1998.

[HP95] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufman Publishers, Inc, 1995.

[HS96] Timothy H. Heil and James. E. Smith. Selective Dual Path Execution. Technical report, Technical report, University of Wisconsin-Madison, 1996.

[Kes99] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, pp. 25–36, 1999.

[KPG98] Artur Klauser, Abhihit Paithankar, and Dirk Grunwald. Selective Eager Execution on the PolyPath Architecture. In *In Proceedings of the International Symposium on Computer Architecture*, pp. 250–259, 1998.

[LS97a] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 227–237, December 1997.

[LS97b] Mikko H. Lipasti and John Paul Shen. Superspeculative Microarchitecture for Beyond AD 2000. In *IEEE Computer*, pp. 59–66, September 1997.

[LTT95] David Levitan, Thomas Thomas, and Paul Tu. The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor. *Compcon Proceedings, IEEE*, pp. 285–291, May 1995.

[LWS96] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, pp. 138–147, October 1996.

[McF93a] S. McFarling. Combining Branch Predictors. Technical report, WRL Technical Note TN-36, Digital Equipment Corporation, 1993.

[McF93b] Scott McFarling. Combining Branch Predictors. Technical report, TN-36, Compaq Computer Corp. Western Research Laboratoty, June 1993.

[MS97] Andreas Moshovos and Gurindar S. Sohi. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *In Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 235–245, 1997.

[NGS99] Tarun Nakra, Rajiv Gupta, and Mary Lou Soffa. Global Context-Based Value Prediction. In *The Fifth International Symposium on High-Performance Computer Architecture*, pp. 4–12, 1999.

[OAH+96] Kunle Olukotun, Basem A.Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–11, October 1996.

[PdWW89] Norman P.Jouppi and david W. Wall. Availabl Instruction-Leval Parallelism for Super Scalar and Superpipelined Machines. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.

[PEFS97] Yale N. Patt, Marius Evers, Daniel H. Friendly, and Jared Stark. One Billion Transistors, One Uniprocessor, One Chip. In *IEEE Computer*, pp. 51–57, September 1997.

[RC99] Glenn Reinman and Brad Calder. Predictive techniques for aggresive load speculation. In *In Proceedings of the 32th Annual International Symposium on Microarchitecture*, pp. 4–12, 1999.

[RTDA98] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, and Todd M. Austin. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *In Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 46–56, December 1998.

[Sat97] Toshinori Sato. Data Dependence Speculation Combining Memory Disambiguation with Address Prediction. *IPSJ SIG Notes 97-RRC-125-1*, Vol. 97, No. 76, pp. 1–6, August 1997.

[SET95] Gurindar S.Sohi, Scott E.Breach, and T.N.Vijaykumar. Multiscalar Processors. *In Proceedings of the 22th Annual International Symposium on Computer Architecture*, pp. 414–425, May 1995.

[SS97] Yiannakis Sazeides and James E. Smith. The Predictability of Data Values. In *In Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 248–258, December 1997.

[US95] Augustus K. Uht and Vijay Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *In Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 313–325, 1995.

[WF97] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *In Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 281–290, December 1997.

[Yea96] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, pp. 28–40, 1996.

[マイ 94] マイク・ジョンソン. スーパースカラ・プロセッサ. 日経 BP 出版センター, 1994.

[安島 98] 安島雄一郎, 中村友洋, 吉瀬謙二, 辻秀典, 田中英彦. スーパスカラ・アーキテクチャのための複数パス実行機構の提案. 並列処理シンポジウム JSPP'98 論文集, pp. 23–30, June 1998.

[吉瀬 99] 吉瀬謙二, 坂井修一, 田中英彦. マルチレベル・ストライド値予測機構による命令レベル並列性の向上. 並列処理シンポジウム JSPP'99 論文集, pp. 119–126, June 1999.

[高澤 93] 高澤嘉光訳. Alpha AXP アーキテクチャ概要. 共立出版株式会社, 1993.

[佐藤 98] 佐藤寿倫. アドレス名前替えによるロード命令の投機的実行. 並列処理シンポジウム JSPP'98 論文集, pp. 15–22, October 1998.

[佐藤 99] 佐藤寿倫. データ値予測とアドレス予測を組み合わせたデータ投機実行. 並列処理シンポジウム JSPP'99 論文集, pp. 111–118, June 1999.

[坂井 97] 坂井修一. オンチップマルチプロセッシングに関する初期的検討. 情報処理学会研究報告 計算機アーキテクチャ研究会, 97-ARC-122, Vol. 15, No. 97, pp. 33–38, February 1997.

[山田 99] 山田裕司, 小林良太郎, 安藤秀樹, 島田俊夫. 2 レベル表構成の導入による分岐先バッファの容量削減. 並列処理シンポジウム JSPP'99 論文集, pp. 103–110, June 1999.

[小沢 99] 小沢年弘, 新井正樹, 細井聡, 木村康則. 分岐予測と条件付実行. 情報処理学会研究報告 計算機アーキテクチャ研究会, 99-ARC-134, Vol. 99, No. 67, pp. 109–114, August 1999.

[小池 99] 小池汎平, 山名早人, 山口喜教. 投機的 制御/データ依存グラフと Java Jog-time Analyzer. 情報処理学会論文誌, Vol. 40, No. SIG1(PRO 2), pp. 32–41, February 1999.

[小林 98] 小林良太郎, 岩田充晃, 安藤秀樹, 島田俊夫. 非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャSKY. 並列処理シンポジウム JSPP'99 論文集, pp. 87–94, June 1998.

[森敦 97] 森敦司, 小林良太郎, 野口良太, 安藤秀樹, 島田俊夫. 直行性を考慮したハイブリッド分岐予測機構. 情報処理学会研究報告 計算機アーキテクチャ研究会, 97-ARC-125, Vol. 97, No. 76, pp. 115–120, August 1997.

[中村 97a] 中村友洋, 吉瀬謙二, 辻秀典, 安島雄一郎, 田中英彦. 分岐アドレス予想機構の比較検討. 情報処理学会第 55 回全国大会, Vol. 1, No. 3F-05, pp. 20–21, September 1997.

[中村 97b] 中村友洋, 吉瀬謙二, 辻秀典, 安島雄一郎, 田中英彦. 大規模データパスプロセッサの構想. 情報処理学会研究報告 計算機アーキテクチャ研究会, 97-ARC-124, Vol. 97, No. 61, pp. 13–18, June 1997.

[中村 99] 中村友洋. 大規模投機処理を特徴とするマイクロプロセッサ・アーキテクチャ–VLDP–. PhD thesis, 東京大学大学院 工学系研究科, 1999.

[鳥居 97] 鳥居淳, 近藤真己, 本村真人, 西直樹, 小長谷明彦. On Chip Multiprocessor 指向 制御並列アーキテクチャ MUSCAT の提案. 並列処理シンポジウム JSPP'97 論文集, pp. 229–236, May 1997.

[野口 98] 野口良太, 森敦司, 小林良太郎, 安藤秀樹, 島田俊夫. 2 レベル表構成の導入による分岐先バッファの容量削減. 並列処理シンポジウム JSPP'99 論文集, pp. 7–14, June 1998.