修士論文

逆Dualflowアーキテクチャ

Anti-Dualflow Architecture

指導教員　五島正裕 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

66403 一林 宏憲

概 要

Out-of-order スーパースカラ・プロセッサにおいて，レジスタ・リネーミングは命令間の依存を解析するために行われる．しかし，レジスタ・リネーミングに用いる RAM である RMT(Register Map Table) は，ポート数が非常に多く遅延が大きい．また，アクセス頻度が高く消費電力も大きい．このため，レジスタ・リネーミングは高価な処理であり，またレジスタ・リネーミングの幅を増やすことも難しい．

本研究では，レジスタ・リネーミングを省略する手法として逆 dualflow アーキテクチャを提案する．逆 dualflow アーキテクチャは，通常の制御駆動の命令を，動的・内部的に制御駆動の性質とデータ駆動の性質をあわせ持った命令に変換して実行するアーキテクチャである．すなわち，制御フローについては，通常の制御駆動の命令セットアーキテクチャと同様，基本的には命令を順番に実行し分岐により実行する命令を変化させる．データ・フローについては，命令の実行結果の授受をレジスタを介して行う通常の方法とは異なり，オペランドを変換して，オペランドとなる実行結果を生産した命令の位置としてオペランドを指定するようにすることにより命令の実行結果の授受を命令間で直接行う．

逆 dualflow アーキテクチャを用いることにより，レジスタ・リネーミングの消費電力を削減し，またパイプラインを短くすることができる．その代わり，トレースの種類が増加することとコンフリクト・ミスが起こりやすくなることによりトレースキャッシュ・ミス率が増加するが，パイプラインが短くなる効果により性能は維持できる．

目 次

図 目 次

表 目 次

第1章　はじめに

Out-of-order スーパースカラ・プロセッサでは，命令間の依存を解析するために，**レジスタ・リネーミング**が行われている．しかし，このレジスタ・リネーミングは非常に高価な処理である．

レジスタ・リネーミングでは，論理レジスタと物理レジスタとの「現在の」マッピングを保持する **RMT (Register Map Table)** を読み出す．2 章で詳しく述べるように，RMT はアクセス頻度が非常に高い．RMT のサイクル当たりの平均アクセス回数は同じくアクセス頻度の高い要素であるレジスタ・ファイルと比べても 2.3 倍にもなる．このため，RMT の消費電力は大きくなってしまう．

また，RMT を RAM で実装した場合，典型的な 4-way スーパースカラ・プロセッサを仮定すると，RMT のポート数は 16 にもなる．このような多ポートの RAM の遅延は配線遅延に支配されており，LSI の微細化，動作周波数の向上とともに，レジスタ・リネーミングに必要なサイクル数が増加する傾向にある．やや極端な例であるが，Pentium 4 プロセッサでは，レジスタ・リネーミングに 3 サイクルが割り当てられている [2].

Dualflow アーキテクチャ　レジスタ・リネーミングを行わない命令セット・アーキテクチャとして，五島らの **dualflow アーキテクチャ**[4, 5, 3] がある．Dualflow アーキテクチャでは，通常の制御駆動型の命令セット・アーキテクチャにあるようなレジスタを定義しない．代わりに，データを生産する命令—**プロデューサ**と，データを消費する命令—**コンシューマ**を指定することで，明示的にデータを受け渡す.

Dualflow アーキテクチャは，元々は命令スケジューリング・ロジックを簡略化，高速化するために提案された．しかし，プロデューサがコンシューマを明示的に指定するので，レジスタ・リネーミングが不要になるというメリットもある．Dualflow アーキテクチャの命令列は，ある意味,「レジスタ・リネーミング済み」ということができる．

しかし，Dualflow アーキテクチャには，命令の配置に関して強い制約がある．Dualflow アーキテクチャでは,「n 命令後」というふうにコンシューマを指定する．そのため，たとえば，分岐命令を超えてデータを受け渡すには，taken 側

と untaken 側でコンシューマの位置を揃えなければならない．適切な位置に有用な命令を配置することができない場合には，転送命令や nop 命令などの本来無用な命令を挿入する必要がある．これらの無用命令のため，通常の制御駆動型の命令セット・アーキテクチャに比べて，命令数が増加してしまっていた．

そこで本研究では，dualflow アーキテクチャを基にした逆 dualflow アーキテクチャを提案する．逆 dualflow アーキテクチャは，以下のようなものである

- 命令セット・アーキテクチャは通常の制御駆動型のものとして，dualflow アーキテクチャを内部表現—マイクロアーキテクチャとして利用する．制御駆動型命令セット・アーキテクチャを動的に dualflow アーキテクチャに変換する．
- 変換後の命令は，**トレースキャッシュ**に格納する．

また，データの授受を指定する方向を逆にする，すなわち，後続の命令が，どの命令の実行結果をソース・オペランドとして使用するかを指定する．

トレースキャッシュからは，dualflow 形式の命令がフェッチされる．前述したように，dualflow 形式の命令は「レジスタ・リネーミング済み」であるので，負荷の高い処理であるレジスタ・リネーミングを省略することができる．

以下，2 章で，レジスタ・リネーミングの負荷について述べる．3 章で，命令の実行結果参照の局所性について述べる．4 章では，提案手法のもととなる Dualflow アーキテクチャについて簡単に説明する．5 章において，提案手法について説明する．6 章で，提案手法の評価を行う．最後に，7 章でまとめと今後の課題を述べる．

第2章　レジスタ・リネーミング

レジスタ・リネーミングでは，1つの命令の実行結果に対し，物理レジスタを1つ割り当てる．論理レジスタ番号から物理レジスタ番号への変換は，論理レジスタ番号と物理レジスタ番号との現在のマッピングを保持する **RMT (Register Map Table)** を読み出すことで行う．RMT を RAM で実装した場合，この RAM を論理レジスタ番号でアドレッシングして読み出すことで論理レジスタ番号と物理レジスタ番号の対応を得る．

1つの命令をリネームするには，デスティネーション論理レジスタに割り当てられていた物理レジスタを解放するための読み出し，デスティネーション論理レジスタに新たに割り当てられた物理レジスタ番号の書き込み，2つのソース論理レジスタに割り当てられている物理レジスタ番号の読み出しを行う．リネームする命令1個につき RMT のポートは4個必要である．4-way スーパースカラ・プロセッサを仮定すると，RMT のポートは 16 個必要である．また，予測ミス回復のために RMT のチェックポインティングを行う場合は，RAM セルがチェックポイントの数だけ追加で必要になる．

このような多ポートの RAM の遅延は配線遅延に支配されるので，プロセッサの動作周波数を高めるにつれて，レジスタ・リネーミングに必要なサイクル数は増加してしまう．また，フロントエンドの幅を増加させることは，RMT のポート数がさらに多く必要になるため，非常に困難である．

さらに，RMT のアクセス頻度は，同じく概念的にはオペランド1つごとに1回アクセスを行うレジスタ・ファイルと比べても非常に高い．これは，物理レジスタ解放のための読み出しが必要であること，ソース・オペランドの多くはフォワーディングにより供給されること，投機ミスのためリネームは行われたが実行はされない命令が存在することが原因である．

RMT とレジスタ・ファイルのアクセス頻度を調べるため，シミュレーションにより IPC，サイクルあたりのレジスタ・ファイルと RMT のアクセス回数を測定した．用いたモデルは6章のベース・モデルである．用いたベンチマークは SPECCPU2000/2006 INT/FP である．INT 系の結果を**図 2.1**，FP 系の結果を**図 2.2** に示す．ソース・オペランドのうちフォワーディングにより供給される割合，レジスタ・ファイルのアクセス回数に対する RMT のアクセス回数の

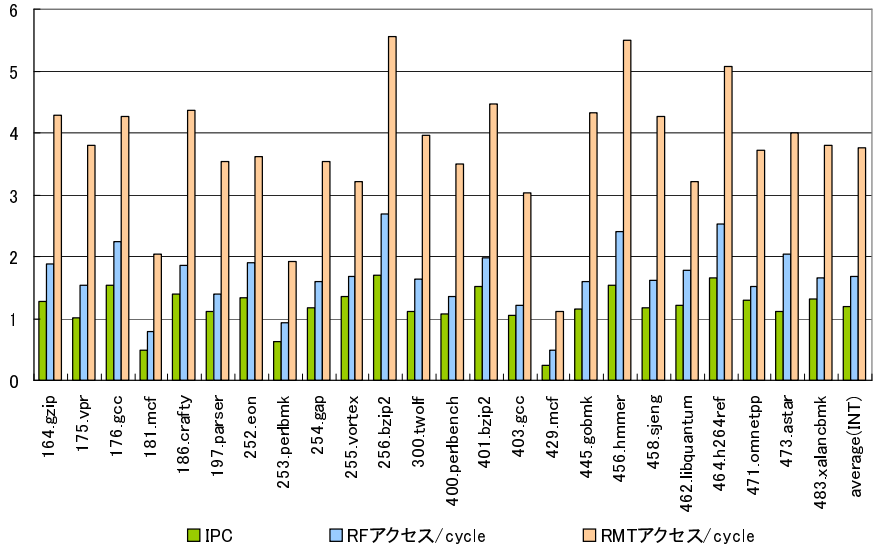

図 2.1: SPECCPU2000/2006 INT の IPC とサイクルあたりの RF アクセス・RMT アクセス

割合はそれぞれ，INT 系では 52%，2.3 倍であり．FP 系では 46%，2.0 倍であり，全体では 49%，2.1 倍であった．ベンチマークの一部に RMT のアクセス回数がレジスタ・ファイルのアクセス回数の 3 倍程度となっているものが存在するが，これらは分岐予測ミス率が 20%を超えて非常に高いものである．

このように，RMT は多ポート，高遅延，高アクセス頻度であるため，レジスタ・リネーミングを省略することによりパイプライン段数を削減でき，またフロントエンドの幅を増加させたり，消費電力を削減したりすることが可能になると考えられる．

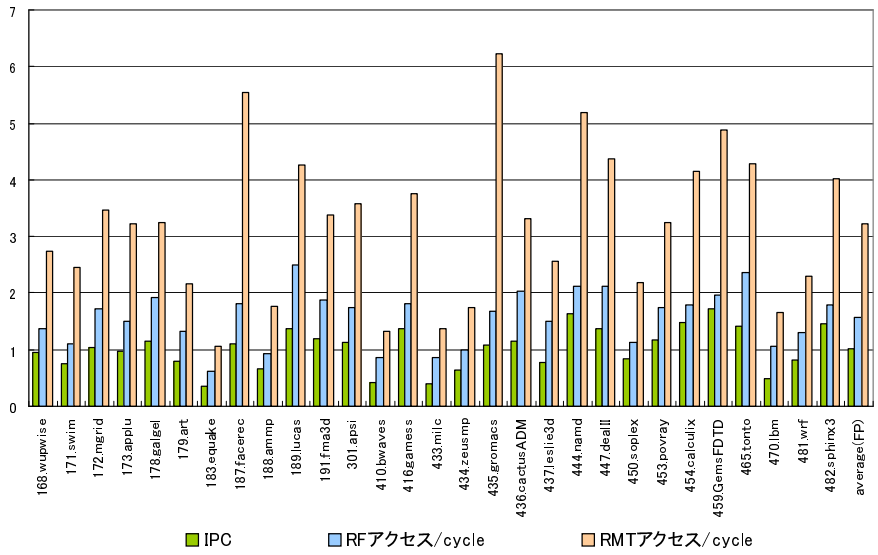

図 2.2: SPECCPU2000/2006 FP の IPC とサイクルあたりの RF アクセス・RMT アクセス

第3章　命令の実行結果参照の局所性

2章で述べたように，レジスタ・リネーミングの際論理レジスタ番号から参照すべき物理レジスタを特定するために参照する表であるRMTの負荷は高い．本研究では，命令が他の命令の実行結果を参照する方法には局所性がある，すなわち，多くの命令は近くの命令の実行結果をオペランドとすることに注目する．

このような局所性が現れるのは，プロセッサに実装されるレジスタの数は有限かつ通常少数であり，全ての値をレジスタに格納することはできないから，すぐに再利用される値や，長い期間にわたって頻繁に再利用される値，計算の途中結果をレジスタに格納しておき，それ以外の値はメモリに書き出して必要に応じて読み込むためであると考えられる．もっと言えば，メモリから値を読み込みし，計算を行い，結果をメモリに書き出すという処理がプログラムの主たる処理であるということである．

長い期間にわたって有効な一定の値を頻繁に再利用するような場合には次のような場合がある：

- 関数内で通常不変である，スタック・ポインタ，グローバル・ポインタ等，アドレスのベースとなる値
- ループ不変式

シミュレーションにより，命令が参照する最も遠くのプロデューサへの距離を測定した．命令セット・アーキテクチャはAlphaであり，使用したコンパイラはGCC 4.2.1，最適化オプションは-O2である．使用したベンチマークはSPECCPU2000とSPECCPU2006であり，入力セットはtrainを用いた．実行した命令は，1G命令をスキップして直後の100M命令である．SPEC全て，SPECCPU2000 INT，SPECCPU2006 INT，SPECCPU2000 FP，SPECCPU2006 FP，それぞれの測定結果を，**図3.1**，**図3.2**，**図3.3**，**図3.4**，**図3.5**に示す．横軸は命令数で計った距離を，縦軸は総実行命令のうち最も遠くのプロデューサへの距離が横軸の距離以下である命令の割合を示す．

この図から分かるように，実行された命令のうち平均で52%は6命令前までの命令の実行結果をオペランドとしている．また，命令の平均で74%が23命

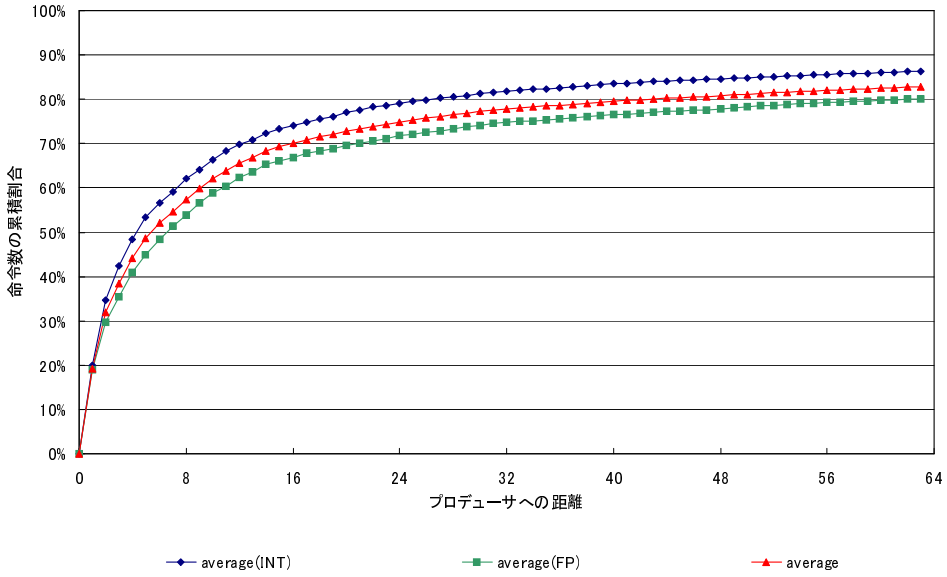

図 3.1: SPECCPU2000/2006，命令が参照する最も遠くのプロデューサへの距離と，命令数の累積割合

令前までの命令の実行結果をオペランドとしており，多くのベンチマークでは距離が 24 命令以上の領域においてグラフの傾きはなだらかでほぼ一定である．この領域は長い期間にわたって有効な一定の値を再利用する場合が多いと考えられる．

また，FP 系のプログラムの方が，より遠くの命令の実行結果を参照する傾向がある．これは FP 系のプログラムの方がループが多い傾向があるためであると考えている．

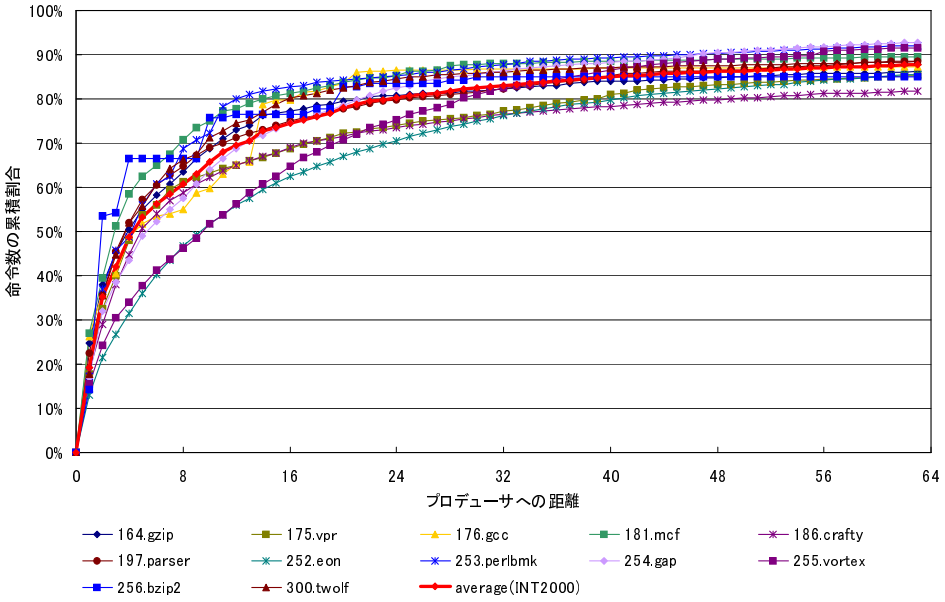

図 3.2: SPECCPU2000 INT，命令が参照する最も遠くのプロデューサへの距離と，命令数の累積割合

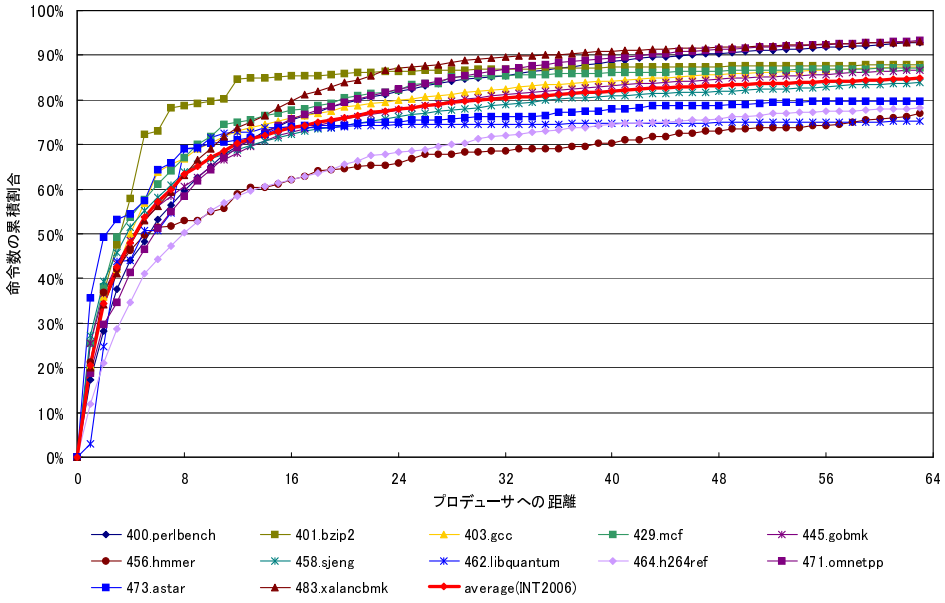

図 3.3: SPECCPU2006 INT，命令が参照する最も遠くのプロデューサへの距離と，命令数の累積割合

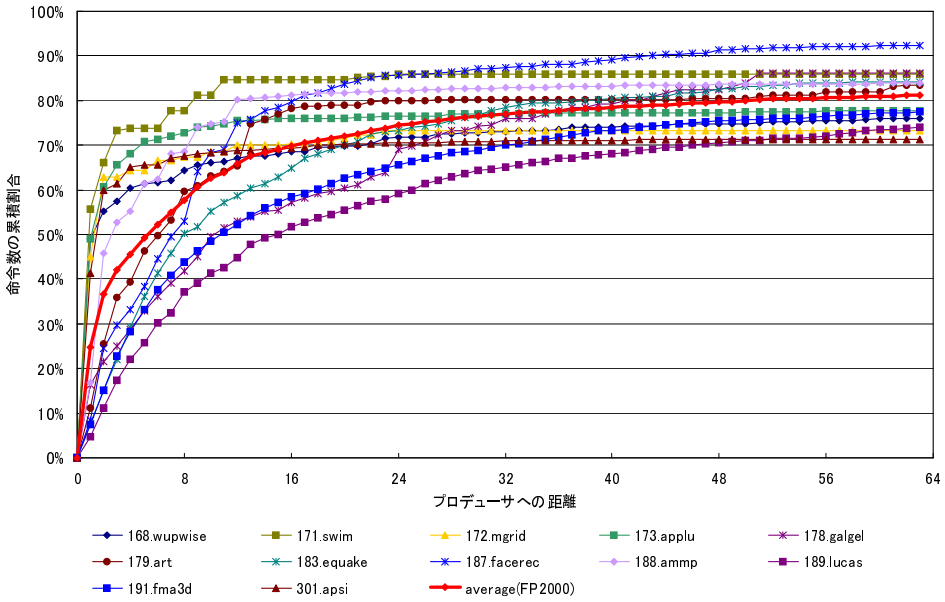

図 3.4: SPECCPU2000 FP，命令が参照する最も遠くのプロデューサへの距離と，命令数の累積割合

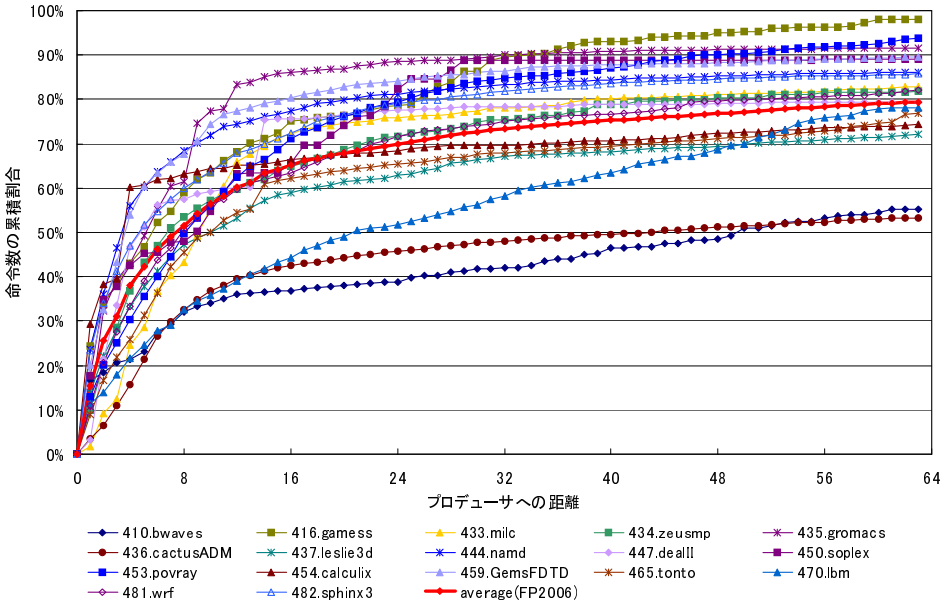

図 3.5: SPECCPU2006 FP，命令が参照する最も遠くのプロデューサへの距離と，命令数の累積割合

第4章　Dualflowアーキテクチャ

本章では，**dualflowアーキテクチャ**[4, 5, 3] について簡単に説明し，その問題点について述べる．Dualflowアーキテクチャ自体の詳細については文献 [4, 5, 3] を参照されたい．

Dualflowアーキテクチャは，制御駆動とデータ駆動の性質をあわせ持った命令セット・アーキテクチャであり：

- 制御フローは，通常の制御駆動の命令セット・アーキテクチャと同様である．すなわち，分岐命令により次に実行する命令を制御する
- データ・フローは，データ駆動方式にならう．すなわち，命令間での実行結果の授受を，通常の制御駆動の命令セット・アーキテクチャのようにレジスタを介して行うのではなく，プロデューサがコンシューマの位置を指定して実行結果をコンシューマに直接送りつけることで行う

4.1　Dualflowアーキテクチャの命令

図 4.1 に，dualflowアーキテクチャの命令フォーマット例を示す．前述したように，通常の制御駆動型の命令セット・アーキテクチャと異なり，ソース・オペランドやデスティネーション・オペランドをレジスタ番号で指定しない．かわりに，コンシューマを指定するフィールド $d1$/$d2$ がある．

$d1$/$d2$ フィールド中の $disp$ フィールドは，この命令の実行結果が $disp$ 命令後の命令のオペランドとして使用されることを示す．その実行結果が左オペランドになるか右オペランドになるかは s フィールドで指定する．imm フィールドは即値である．

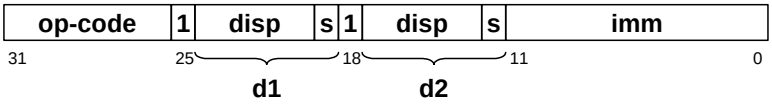

図 4.1: Dualflowアーキテクチャの命令フォーマット例

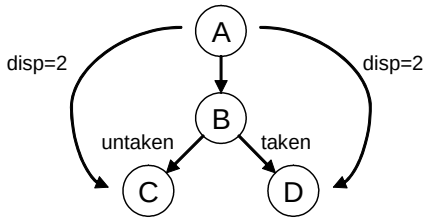

図 4.2: 分岐を挟んだ実行結果の授受

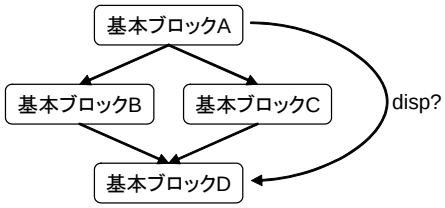

図 4.3: 基本ブロックを挟んだ実行結果の授受

4.2 レジスタ・リネーミングとの関係

レジスタ・リネーミングは．1つの命令の実行結果に対しその実行結果専用の物理レジスタを1つ割り当てる．これによって物理レジスタに書き込むのは最初のただ1回だけで，後は読み出されるだけになる．これによって，偽の依存関係は解消され，真の依存関係，すなわちデータ・フローが明らかになる．

さて，dualflow アーキテクチャにおいて命令に明示されたプロデューサとコンシューマの関係は，データ・フローそのものである．データ・フローが明示されているという意味で，dualflow アーキテクチャの命令列は「レジスタ・リネーミング済み」であると言える．よって dualflow アーキテクチャではレジスタ・リネーミングが不要である．

4.3 問題点

dualflow アーキテクチャでは，実行結果の授受に関する次の3種類の制約がコードに課せられる:

数と距離 コンシューマを指定するフィールドは2つしかないので，2つ以上の命令に実行結果を送りつけるには，実行結果を受け取ってそのまま後の命令に送るだけの転送命令を挿入する必要がある．また，disp フィールドは5ビットであるから，32 命令以上離れた命令に実行結果を送るには，転送命令を挿入するかメモリを介する必要がある．

条件分岐　コンシューマの位置はコンパイル時に静的に決定しなければならないので，条件分岐の前の命令が分岐の後の命令に実行結果を送る場合であっても，分岐の結果によって実行結果を送る位置を変えることはできない．いずれかの条件分岐先で使用される実行結果であれば，両方の分岐先に送ることになる．分岐先においては，同じ順番で，実行結果を受けとるか nop で捨てる必要がある．**図 4.2** に例を示す．命令 A から untaken 側の命令 C に実行結果を送る場合，同時に taken 側の命令 D も命令 A の実行結果を受け取ることになる．このため，命令 D でも命令 A の実行結果を使用するか捨てる必要がある

基本ブロック　1 つ以上の基本ブロックを越えて実行結果を送る場合，プロデューサからコンシューマへの変位が静的に定まらないため，直接実行結果を送ることができない．実行結果を送るためには，メモリを介する必要がある．関数呼び出し，if-then-else 構造，ループを越えた実行結果の授受が，これにあたる．**図 4.3** に if-then-else 構造における例を示す．一般に，基本ブロック B と C の大きさは異なるので，基本ブロック A から基本ブロック D への変位を静的に決定することはできない．

これらの制約によって，プログラムの本来の処理に関わらない転送命令，nop，ロード/ストア命令が増加してしまい，性能が低下する問題がある．距離の制約については，これらの制約によって挿入される命令のために命令間の距離が伸び，さらに転送命令やロード/ストア命令を挿入する必要が生じる悪循環がある．

距離の制約を除くと，これらの制約は次のことが原因である:

- 実行結果を生産するのは 1 回だけであるが，その実行結果を参照する回数に本来制限はない．にもかかわらず，実行結果の受け渡しを指定する方向が，プロデューサがコンシューマを指定するようになっていること
- 分岐の例に見るように，データ・フローは制御フローに依存する．にもかかわらず，静的にプロデューサとコンシューマとの距離を決定していること

第5章　逆Dualflowアーキテクチャ

2 章で述べたように，レジスタ・リネーミングの際論理レジスタ番号から参照すべき物理レジスタを特定するために参照する表である RMT の負荷は高い．逆 dualflow アーキテクチャは，レジスタ・リネーミングそのものを省略することを目的とする．

逆 dualflow アーキテクチャは，dualflow アーキテクチャと同様制御駆動とデータ駆動の性質をあわせ持ったアーキテクチャである．すなわち，命令のオペランドをプロデューサとコンシューマ間の変位で指定することにより実行結果を直接参照する．ただし，次の点が異なる：

- 逆 dualflow アーキテクチャは命令セット・アーキテクチャではない．命令セット・アーキテクチャとしては通常のレジスタを用いた命令セット・アーキテクチャを使用し，その命令のオペランドを「動的に」変位で指定する形式に変換する．これにより，プログラムに無用な命令を挿入しなければならない問題がなくなる
- プロデューサがコンシューマの位置を指定するのではなく，コンシューマがプロデューサの位置を「n 命令前」と指定する．これにより，プロデューサの実行結果を直接参照できるコンシューマの数に制限はなくなる

5.1　基本的な考え方

逆 dualflow アーキテクチャの基本的な考え方は次の通りである:

- 命令の実行結果を格納するサイクリックなバッファである**物理レジスタ・ファイル**を用意し，命令の出現した順番と同じ順番で実行結果を格納する
- 物理レジスタ・ファイルとは別に，論理レジスタの値を保持する**論理レジスタ・ファイル**を用意し，命令の実行結果を in-order に格納する
- 初回の実行時に命令を変換し，ソース・オペランドを「n 命令前」の実行結果として物理レジスタ・ファイル上での変位で指定するようにする．

図 5.1: Dualflow 形式

ただし，5.3 節や 5.5 節で述べるようにリタイア済みであり実行結果が論理レジスタ・ファイルに格納されていることが保証できる命令の実行結果を使用する場合には，ソース・オペランドを論理レジスタ番号のままで指定する

- 変換した命令をトレースキャッシュにキャッシュ・再利用する

このようにすることで，レジスタ・リネーミングを省略してもオペランドへのアクセスを可能にする．

以下，逆 dualflow アーキテクチャについて詳細に述べる．

5.2 命令の内部表現: dualflow 形式

逆 dualflow アーキテクチャでは，命令のソース・オペランドを，論理レジスタ番号で指定する通常の形式から「n 命令前」の命令の実行結果として物理レジスタ・ファイル上の変位で指定する形式に内部的に変換する．ただし，前述したようにリタイア済みであることが保証できる命令の実行結果を使用する場合には，ソース・オペランドを論理レジスタ番号のままで指定する．

以下，この形式を **dualflow 形式**と呼ぶ．**図 5.1** に，dualflow 形式の例を示す．各フィールドの意味は次の通りである:

$Opcode$ $Opcode$ は，命令のオペ・コードを示す

$DstReg$ $DstReg$ は，デスティネーション論理レジスタ番号を示す

$SrcL/R$，RL/R $SrcL/R$ フィールドは，ソースオペランドを示す．RL/R フィールドが 0 のとき $SrcL/R$ 命令前の命令の実行結果が，RL/R フィールドが 1 のとき $SrcL/R$ 番の論理レジスタがこの命令の左/右ソース・オペランドとして用いられることを示す

Imm Imm は，即値を示す

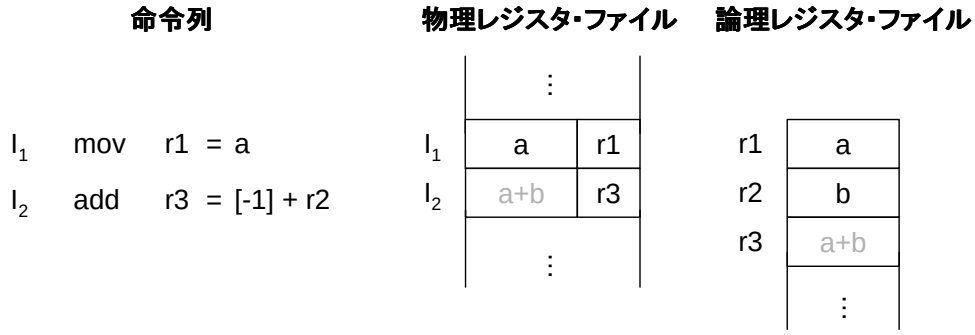

図 5.2: Dualflow 形式の命令の実行

5.3 Dualflow 形式の命令の実行

Dualflow 形式の命令の実行方法を，**図 5.2** を用いて説明する．$[-n]$ は,「n 命令前の命令の実行結果」を表す.

まず，各命令に対し，物理レジスタ・ファイルのエントリを命令の出現した順番と同じ順番で割り当てる．順番に割り当てることにより，読み出すべき物理レジスタは，命令に割り当てられた物理レジスタのインデックスに変位を加算するだけで決定することができる．物理レジスタ・ファイルのエントリには，対応する命令のデスティネーション論理レジスタ番号を付随させておく．命令の実行結果は，実行の終わったものから out-of-order に物理レジスタ・ファイルに格納する．その後，最も古い命令に対応する物理レジスタのエントリから順番に，すなわち in-order に実行結果を論理レジスタ・ファイルにコピーする．

ここで，ある命令のソース・オペランドが**命令ウィンドウサイズ** WS 以上前の命令の実行結果である場合を考える．WS 以上前の命令はリタイアしているので，実行結果は論理レジスタ・ファイルにコピーされている．そのような命令の実行結果は論理レジスタ・ファイルから読み出すことができる．よって，ソース・オペランドは，WS 以上前の命令の実行結果を参照する場合論理レジスタ番号のまま指定する.

物理レジスタ・ファイルは，命令ウィンドウ中の最も古い命令が，変位でオペランドを指定する最大の範囲である $(WS-1)$ 前の命令の実行結果を利用できるようにするため，$(2 \times WS)$ エントリを用意する.

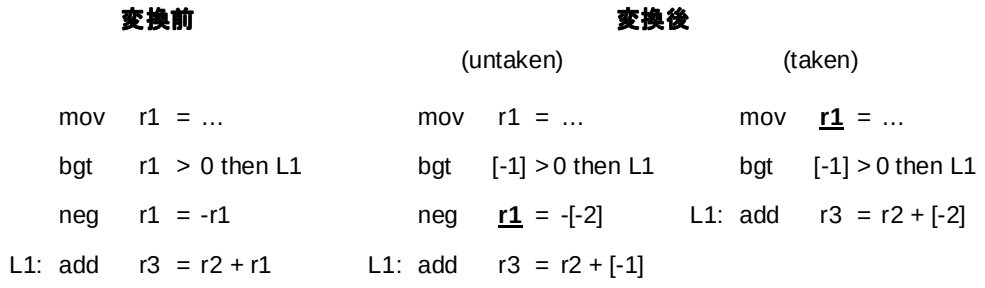

図 5.3: 変換結果の変化

5.4 Dualflow形式への変換とトレースキャッシュ

Dualflow 形式への変換は，レジスタ・リネーミングと同様に行うことができる．まず，命令にフェッチされた順の通し番号を振る．論理レジスタ番号とそのプロデューサの番号の対応表を用意し，命令がフェッチされるたびに更新する．対応表からソース・オペランド・レジスタに対応する命令の番号を読み出し，変換対象の命令の番号との差を取れば，何命令前の命令の実行結果をソース・オペランドとして用いればよいかが分かる．5.3 節で述べたように，WS 以上前の命令の実行結果を参照する場合には論理レジスタ番号で参照する．

ここで，同時に多数の命令を変換する場合，結局，この対応表の負荷が RMT と同様に高くなってしまう．しかし，変換は初回の実行時のみ行われ，以後はキャッシュした dualflow 形式の命令を再利用できるので，同時に変換できる命令が少なくても性能への影響は少ないと考えられる．今回は，1 サイクルに 1 命令を変換できるとした．

5.5 節で詳しく述べるが，変換対象の命令までに実行した命令によって，ソース・オペランドのプロデューサとなる命令の位置は変化し，変換結果も変化する．よって変換結果をキャッシュする際にキャッシュのエントリ内で命令の実行順を表現できる必要がある．変換結果のキャッシュには実行順に命令を並べたトレースをエントリとしてキャッシュを行うトレースキャッシュを用いるのがよい．

5.5 変換結果の識別子: パス

逆 dualflow アーキテクチャでは，命令のソース・オペランドを「n 命令前」と変位で表現する形式に変換する．しかし，変換中の命令までに実行した命令によって，ソース・オペランドのプロデューサとなる命令の位置は変化する．**図 5.3** に例を示す．分岐命令 bgt の成否により，r1 のプロデューサとなる命令

も変わり，add 命令の変換結果も異なってしまう．このため，同じアドレスのトレースであっても変換結果のトレースは区別してキャッシュする必要がある．

変換結果のトレースを一意に識別できる情報を**パス** と呼ぶことにする．変換結果はソース・オペランドのプロデューサとなる命令の位置で変化するのであるから，変換結果を一意に識別するためには，トレースにあるソース・オペランド全てについてそれらのプロデューサの位置が一意に特定できればよい．特定するためには，トレースにある命令のソース・オペランドのプロデューサとなる命令のうち，最も遠くの命令からそのトレースまでに実行した命令列が特定できれば十分である．すなわち，最も遠くのプロデューサを L 命令前とすると，パスは直前に実行した L 個の命令の列である．ただし，詳細は 5.7 節 で述べるが，トレースキャッシュのコンフリクト・ミスを軽減するためにパスの長さ L は一定の値 $L_{\min}$ を最短とする．

また，5.3 節で述べたように WS 以上前の命令はリタイアしており，それらの命令の実行結果は論理レジスタ番号で参照できるので，パスの長さ L は最長でも $(WS-1)$ でよい．論理レジスタ番号で指定する場合，そのレジスタのプロデューサの位置が WS 以上前であること，すなわち $(WS-1)$ 前までの命令がそのレジスタに書き込まないことを保証する必要があるため，パスは $(WS-1)$ 必要であることに注意する．

5.6 パスの表現

5.5 節で述べたように，変換結果のトレースはパスによって区別してキャッシュする必要がある．パスは直前に実行した L 個の命令の列であるから，単純には直前の L 命令全てのアドレスで表現することができる．しかし，この表現ではサイズが膨大になってしまうので，コンパクトな表現が必要である．

今，ある命令の次の命令について次のことが言える:

- 条件分岐の次の命令は，分岐の成否によって一意に定まる
- return を含む間接分岐の次の命令は，間接分岐のターゲットによって一意に定まる
- それ以外の命令の次の命令は一意に定まる

よって:

- パスの先頭の命令のアドレス H
- パスの長さ L

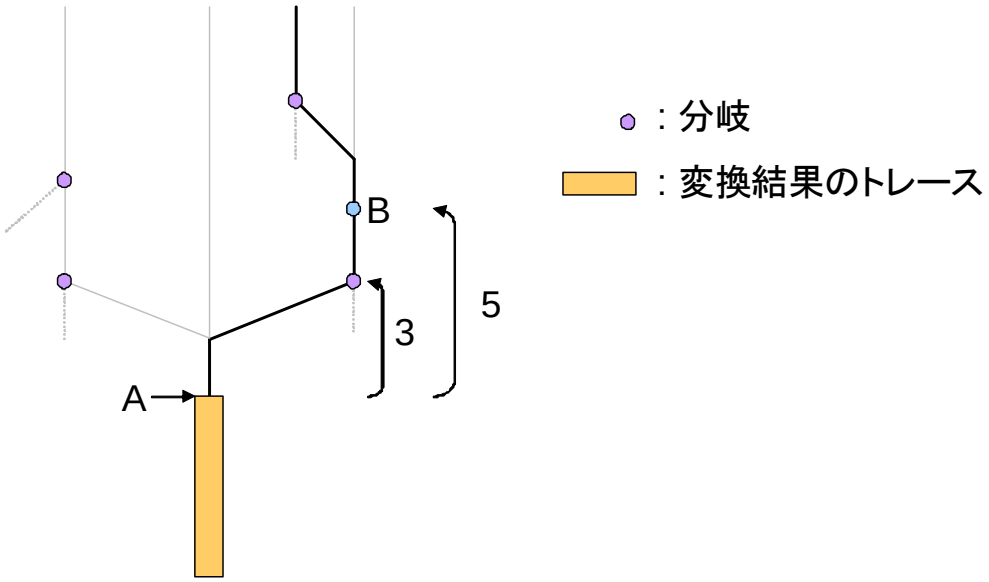

図 5.4: パスの表現

- パス内の条件分岐の成否 $bflag$
- パス内の間接分岐のターゲット $jtarget$ とその数 $jumpCount$

により，パスを表現することができる．また，これらの情報からこのパスの直後の命令，つまりトレースの先頭アドレスも一意に定まる．

条件分岐の成否は 1 ビットで表せるから，パス内の条件分岐の成否を保持するには $(WS-1)$ ビットあればよい．パス情報の生成を簡単にするため，条件分岐でない命令に当たる部分は 0 として，n ビット目に $(n+1)$ 命令前の分岐の成否を保持する．これにより，m 命令フェッチしたときには，このビット列を m ビットだけシフトし，フェッチした m 命令分の分岐成否を付け加えればよくなる．

また，パスに含めることのできる間接分岐の数を $J_{\max}$ 個に制限する．制限することによって $(WS-1)$ 命令前までの間接分岐ターゲットを全て保持することができなくなる場合，パス情報として，$(J_{\max}+1)$ 個前の間接分岐のターゲット（T とする）をパスの先頭とし，$J_{\max}$ 個前までの間接分岐ターゲットのみを保持することにする．このような間接分岐数の超過が起こった場合，変換時には，T より前の命令の実行結果は論理レジスタ番号で参照するようにする．トレースキャッシュのフェッチ時には，T より前の命令がリタイアし，実行結果が論理レジスタ・ファイルに格納されるまで，フェッチをストールさせる．

パスに含めることのできる間接分岐の数を増やすと，パスの表現に使用す

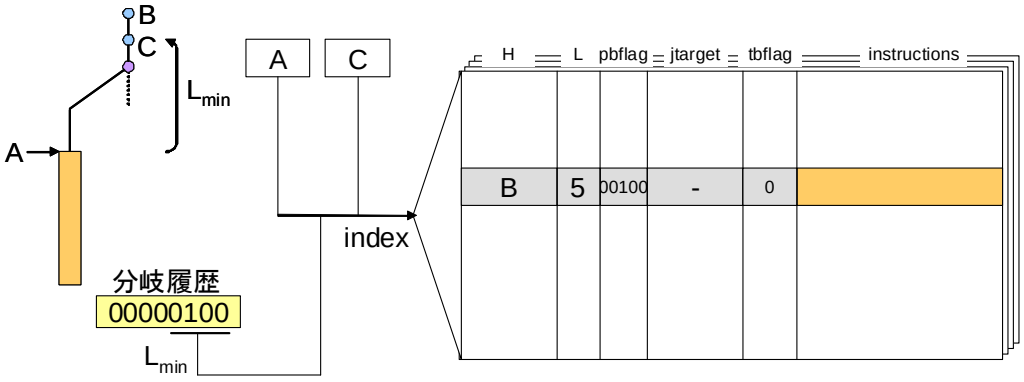

図 5.5: トレースキャッシュへの格納時のインデックス生成

る領域が大きくなり，使われない部分は無駄になってしまうが，フェッチのストールする頻度は低くなる．間接分岐の数を減らすと，パスの表現には無駄が少なくなるものの，フェッチのストールする頻度は高くなってしまい性能が悪化する．

図 5.4 に例を示す．トレースの直前に実行する命令には複数のパターンが存在する．今，太線の経路を通って変換結果のトレースに到達した場合で，トレースが参照する最も遠くのプロデューサが 5 命令前の命令 A であったとする．また 3 命令前の条件分岐は taken であったとする．5.5 で述べたように，パスは命令 B から 5 命令の命令を特定する情報であり，先頭のアドレス H は B，長さ L は 5，条件分岐の成否 bflag は 00100 を以て直前の 5 命令を一意に特定できる．

5.7 トレースのインデックス生成

前述したように，トレースキャッシュのコンフリクト・ミスを軽減するため，パスの長さ L は一定の値 $L_{\min}$ を最短とする．パスの最短の長さを $L_{\min}$ とすると，全てのパスには $L_{\min}$ 前までの命令が含まれる．このとき，インデックスの生成には，$L_{\min}$ 命令前の命令のアドレス，$L_{\min}$ 前までの条件分岐履歴 $bflag$，トレースのアドレスを用いることができる．ただし，5.6 節で述べた間接分岐数の超過によりパスの長さ L が $L_{\min}$ 未満になる場合は，L 命令前の命令のアドレス，L 前までの条件分岐履歴，トレースのアドレスを用いる．

図 5.5 にトレースを格納するときのインデックス生成の例を示す．格納するトレースは図 5.4 のトレースと同じである．このようにトレースの先頭アドレス，$L_{\min}$ 前の命令のアドレス，格納するトレース以前の $L_{\min}$ 命令分の分岐履

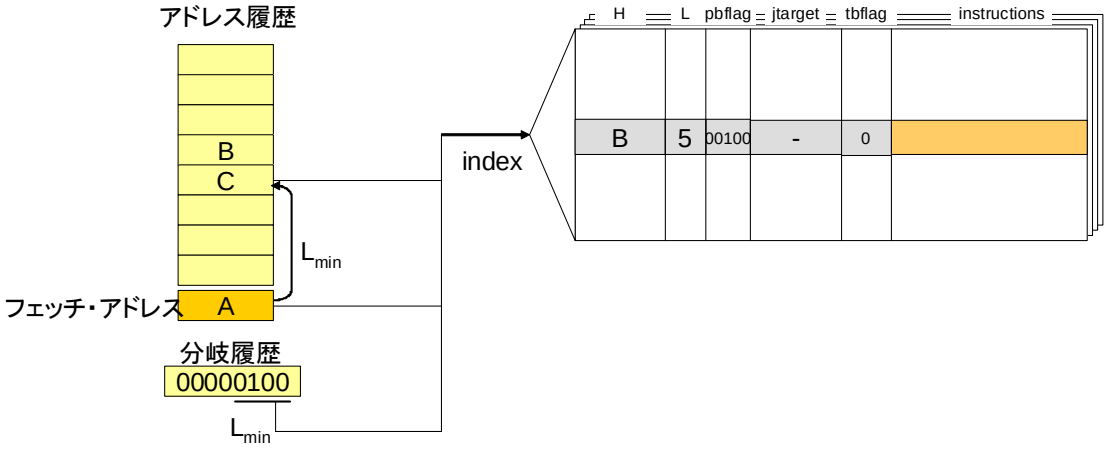

図 5.6: トレースキャッシュフェッチ時のインデックス生成

歴を用いてインデックスを生成する．インデックスの生成は，例えば各値をインデックスのビット数に畳み込み適当にローテートして XOR することにより行う．

もしパスの最小の長さ $L_{\min}$ を設定しない ($L_{\min}= 0$) 場合，全てのパスに共通して含まれる情報は存在しない．このとき，インデックスの生成にはトレースのアドレスしか用いることができない．トレースをパスによって区別してトレースキャッシュに格納する際，区別した全てのトレースに対して同じインデックスが生成されるため，コンフリクトが非常に起きやすくなる．

$L_{\min}$ は (例えばプログラム毎に) 切り替え可能である．切り替えるとトレースに対して生成されるインデックスが変わるため，既にトレースキャッシュに格納したトレースは引けなくなる．インデックスが合わなくなって目的のトレースの格納されたセットを特定できなくなるだけであり，タグが変わるわけではないので，わざわざトレースキャッシュのエントリを無効化する必要はない．

5.8 トレースのキャッシュとフェッチ

変換結果のトレースをトレースキャッシュに格納する際に用いるタグは，通常のトレースキャッシュでも用いるトレース内の条件分岐の数と成否に，5.6 節で述べたパスを合わせたものを用いる．トレースを格納するセットは，5.7 節で述べた方法で生成したインデックスにより指定する．

変換済みのトレースをフェッチする方法を次に述べる．フェッチ時にはそれ以前にフェッチした命令の，アドレスの履歴，条件分岐履歴，間接分岐ターゲットの履歴を保持しておく．まず，これらの履歴から，5.7 節で述べた方法でイ

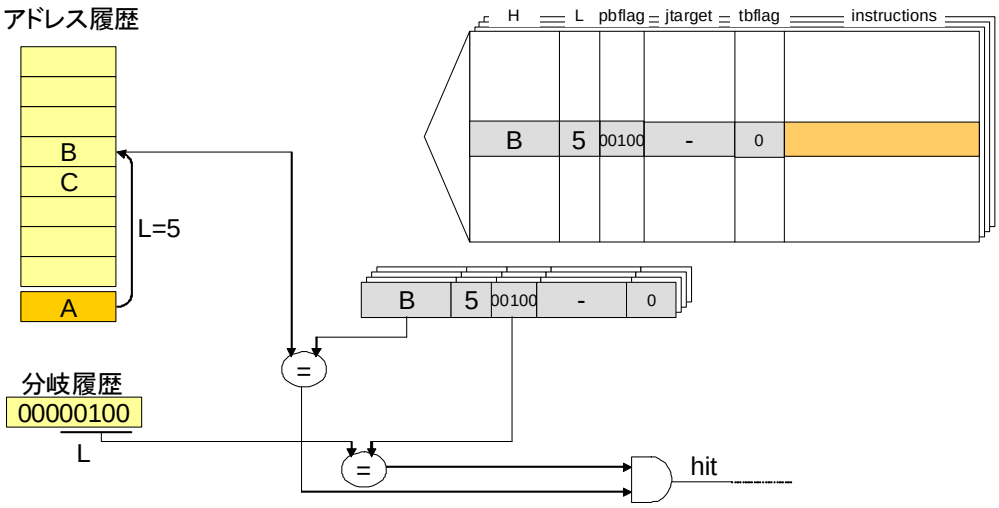

図 5.7: トレースキャッシュセット中の各エントリのタグ比較

ンデックスを生成し，フェッチ対象のトレースが格納されているであろうセットを特定する．このセットのタグ部分をトレースキャッシュから読み出せたら，各ウェイに対して，パスの長さ L を用いてアドレスの履歴から L 命令前の命令のアドレスを読み出し，タグ中のパス先頭アドレス H と比較する．また，条件分岐履歴の直近の L ビット，間接分岐ターゲットの履歴の直近の $jumpCount$ 個を，読み出したタグ中のパスの条件分岐成否 $bflag$，$jtarget$ とそれぞれ比較する．全てが一致すればそれが使用すべき変換済みのトレースである．

図 5.6 にセットを特定するためのインデックス生成の例を示す．命令をフェッチするときに，フェッチした命令のアドレスの履歴，条件分岐履歴を保持しておく．アドレスの履歴から $L_{\min}$ 前の命令のアドレスを，条件分岐履歴から $L_{\min}$ ビットをそれぞれ取り出し，フェッチ・アドレスとあわせてインデックスを生成する．このインデックスを用いてトレースキャッシュのセットを読み出す．セットのタグ部分が読み出せたら，タグの比較を行う．**図 5.7** に例を示す．読み出したタグ中のパスの長さ L は 5 であるから，アドレス履歴中の 5 命令前の命令のアドレス B を読み出し，タグ中のパス先頭アドレス B と比較する．また，分岐履歴の下位 5 ビットを取り出し，タグ中の分岐履歴 $bflag$ と比較する．いずれも一致すれば，それがフェッチすべきトレースである．

このように，タグの比較を行うロジックは通常のトレース・キャッシュに比べて若干複雑になる．また，トレースキャッシュからセットのタグ部分を読み出してからはじめて，タグ中のパスの長さ L が分かる．この L を用いて履歴からタグの比較に用いるパス先頭アドレスの読み出しを行い，タグ中のパスと比較するので，トレースキャッシュのヒット・ミスが判明するタイミングは遅

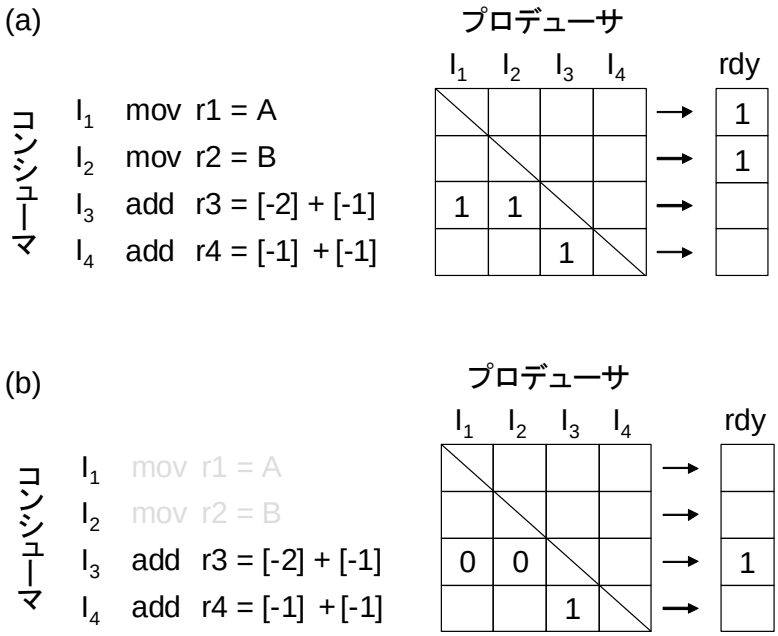

図 5.8: 依存行列を用いたスケジューリング．(a), (b) は，それぞれ I_1, I_2 を実行する前と後

くなる．しかし，タグはデータ部分よりも小さく読み出しが速いので，トレースのフェッチにかかるレイテンシが大きくなるほどではないと考えている．

5.9 逆 dualflow アーキテクチャの命令スケジューリング

逆 dualflow アーキテクチャの命令スケジューリングは，五島らの依存行列を用いる方法 [3, 1] により容易に行うことができる．逆 dualflow アーキテクチャにおける依存行列を用いたスケジューリングを，**図 5.8** の例を用いて説明する．(a) は，I_1，I_2 が実行される前，(b) は実行された後の状態である．

命令ウィンドウへのディスパッチ時に，プロデューサとコンシューマの依存関係を表す依存行列を生成する．依存行列の行は，その行に対応する命令がどの命令の実行結果を参照しているかを示す．図 5.8 の (a) では，I_3 は 2 命令前と 1 命令前の命令である I_1，I_2 の実行結果を参照しているので，I_3 行の I_1，I_2 列が 1 になっている．

逆 Dualflow アーキテクチャでは，この依存行列の生成を容易に行うことができる．命令の番号にプロデューサへの変位を足すだけで，依存行列の行のう

ち 1 にすべき列を特定できるからである．

依存行列の各行について，要素が全て 0 であれば rdy ビットを 1 にする．すなわち，rdy ビットは各命令に対応する行の要素全ての NOR である．rdy ビットが 1 であることは，その命令が依存する命令の実行結果は全て揃っており，発行可能であることを示す．図 5.8 の (a) では，I_1 と I_2 に対応する行が全て 0 であるため，I_1 と I_2 が発行可能であると判定する．

命令が実行されたら，実行された命令に対応する列を 0 クリアする．依存する命令が全て実行された命令は，全ての列がクリアされており発行可能であると判定できる．図 5.8 の (b) では，I_1，I_2 が実行されたことにより，I_1，I_2 列が 0 クリアされ，I_3 行が全て 0 であるので，I_3 が発行可能であると判定する．

5.10 逆 dualflow アーキテクチャの効果

逆 dualflow アーキテクチャにより，レジスタ・リネーミングを省略することができる．レジスタ・リネーミングを省略することにより:

- レジスタ・リネーミングに必要なハードウェア・電力 (特に RMT) が削減できる
- レジスタ・リネーミング・ステージのぶん，分岐予測ミス・ペナルティが減少する
- フェッチした後フロントエンドで行う処理に命令間の依存がなくなるため，フロントエンドの幅を増やせる可能性がある

ただし，変換結果のトレースをパスで区別することによりトレースの種類が増加するため，トレースキャッシュ・ミス率が増加してしまう．

第6章　評価

6.1　評価環境

シミュレーションにより，逆 dualflow アーキテクチャの評価を行った．シミュレーションには本研究室で開発したシミュレータ「鬼斬弐」を用いた．ベンチマークには，SPECCPU CINT2000 のプログラムから，164.gzip，175.vpr，176.gcc，181.mcf，186.crafty，197.parser，252.eon，253.perlbmk，254.gap，255.vortex，256.bzip2，300.twolf の 12 本を用いた．入力セットには train を用い，最初の 1G 命令をスキップし直後の 100M 命令を実行した．

表 6.1 にベース・モデルの主要なパラメータを示す．通常の命令キャッシュの使用タイミングについては，トレースキャッシュにヒットしている間は通常の命令キャッシュへ同時にアクセスを行わず，トレースキャッシュ・ミスが判明してから通常の命令キャッシュにアクセスするものとした．トレースキャッシュ・ミスが判明するタイミングは，ベース・モデルではフェッチの 2 サイクル目，逆 dualflow アーキテクチャでは 5.8 節で述べたようにタグ比較が複雑になるためフェッチの 3 サイクル目とした．

逆 dualflow アーキテクチャにおけるパスの長さの最小値 $L_{\min}$ は 11 とした．また，逆 dualflow アーキテクチャでは Rename ステージなし (0 サイクル) とした．

6.2　パスに用いる間接分岐数

5.6 節で述べたように，パスに用いることのできる間接分岐の数 $J_{\max}$ を大きくすれば，間接分岐数の超過によりフェッチがストールすることは少なくなるがパスの表現が大きくなる．逆に $J_{\max}$ が小さすぎると，フェッチが頻繁にストールすることになり，性能が悪化してしまう．

そこで，パス情報に用いる間接分岐数 $J_{\max}$ の性能への影響を測定した．トレースキャッシュ・ミスの影響を排除するため，逆 dualflow アーキテクチャのトレースキャッシュは，64k エントリ (1MB) とした．**図 6.1**，**図 6.2** に測定結果を

表 6.1: ベース・モデルの主要なパラメータ

フェッチ幅	4
発行幅	Int 2，FP 2，Mem 2
命令ウィンドウ	32 エントリ
L1 命令キャッシュ	32KB, 4 ウェイ, 3 サイクル，64B ライン
L1 データ・キャッシュ	32KB, 4 ウェイ, 3 サイクル，64B ライン
L2 データ・キャッシュ	4MB, 8 ウェイ, 10 サイクル，64B ライン
メイン・メモリ	200 サイクル
演算器	IntALU × 2, IntMUL × 1, Mem × 2, FPADD × 1，FPMUL × 1, FPDIV × 1
分岐予測	BTB: 8K エントリ, gshare: 32K エントリ PHT, 10 ビットグローバル分岐履歴 RAS: 8 エントリ
トレースキャッシュ	2K エントリ, 8 ウェイ, エントリ・サイズ: 4 命令 (16B), トレース中の分岐数: 1
パイプライン構成	Fetch: 3 サイクル, Rename: 2 サイクル, Dispatch: 2 サイクル，Issue：2 サイクル

示す．グラフの横軸がベンチマーク（右端は平均）であり，ベンチマークごとの 4 本の棒グラフは，左からそれぞれ，$J_{\max}$ が 0，1，2，3 の場合の逆 dualflow アーキテクチャに対応する．図 6.1 の縦軸は，ベース・モデルを 1 として正規化した IPC である．図 6.2 の縦軸は，プログラムの総実行サイクル数に対する，間接分岐数の超過によりフェッチがストールしたサイクルの割合である．

$J_{\max}$ が 0 の場合，return を含む間接分岐が出現するたびに全ての命令がリタイアするまでフェッチをストールさせなければならない．このため，平均で，総実行サイクル数の 40%もの間フェッチがストールしてしまいベース・モデルに比べて 23.7%も IPC が低下してしまう．

$J_{\max}$ が 1 の場合は，フェッチのストールは総実行サイクル数の 6.8%であり，パイプラインが短くなる効果によりベース・モデルに対して 1.3%IPC が向上している．

$J_{\max}$ が 2 の場合は，フェッチのストールは総実行サイクル数の 1.4%に抑えられており，$J_{\max}$ が 1 の場合に比べて 3.2%IPC が向上している．これに対し，$J_{\max}$ が 3 の場合は $J_{\max}$ が 2 の場合に対して平均 IPC が 0.3%しか向上していな

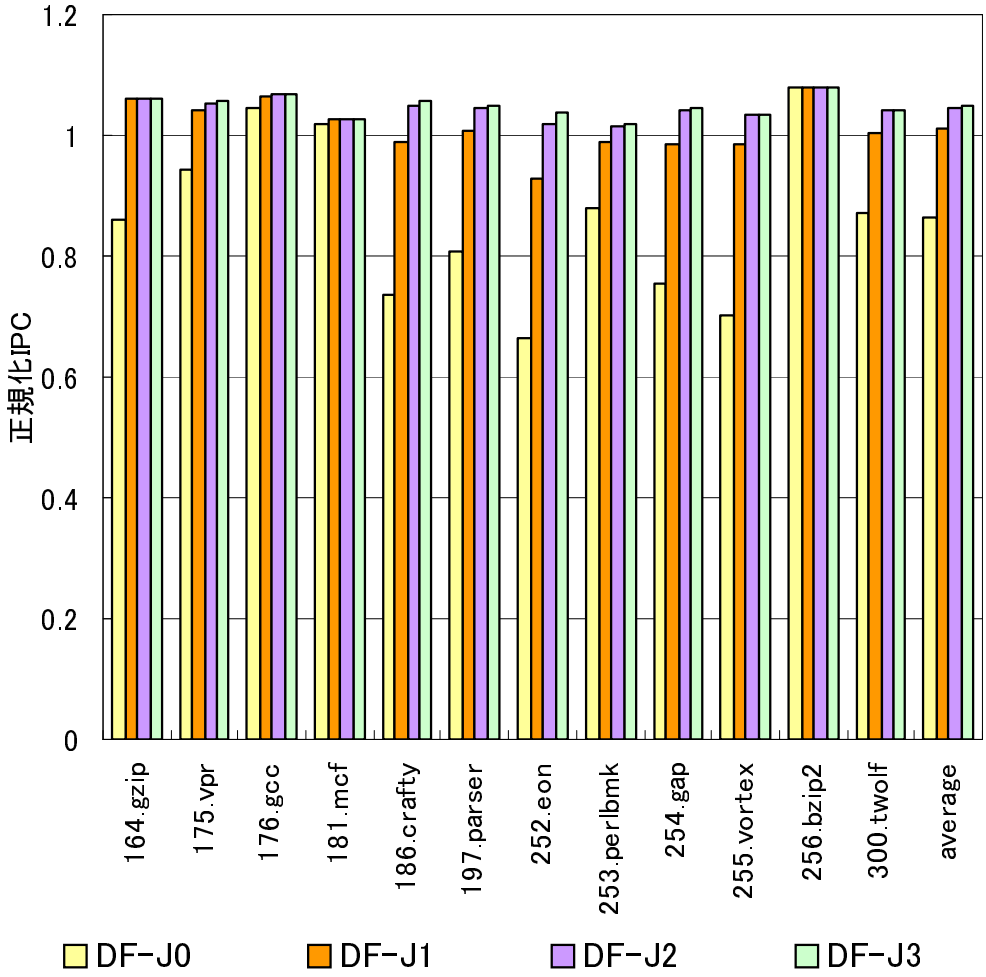

図 6.1: $J_{\max}$ と，Base=1 とした正規化 IPC

い．これは，3 個目の間接分岐ターゲットが格納されないことが多い，もしくはパス情報に間接分岐を 3 個しか含めない場合にパスとして表現できなくなる部分の長さが十分に小さいためと考えられる．

よって，パスには間接分岐を 2 個含むことができれば，間接分岐数の超過による性能の低下はほとんどないと言える．

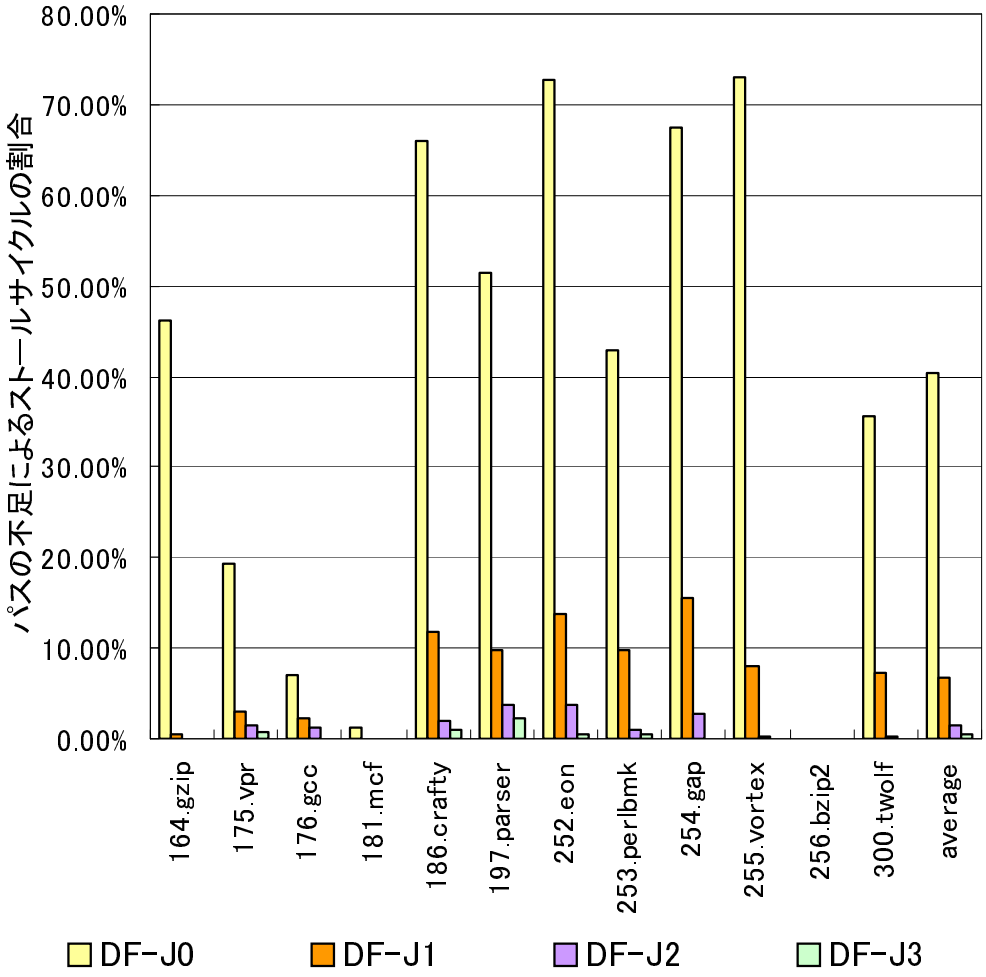

図 6.2: 総実行サイクル数に対する，間接分岐数超過によってフェッチがストールしたサイクル数の割合

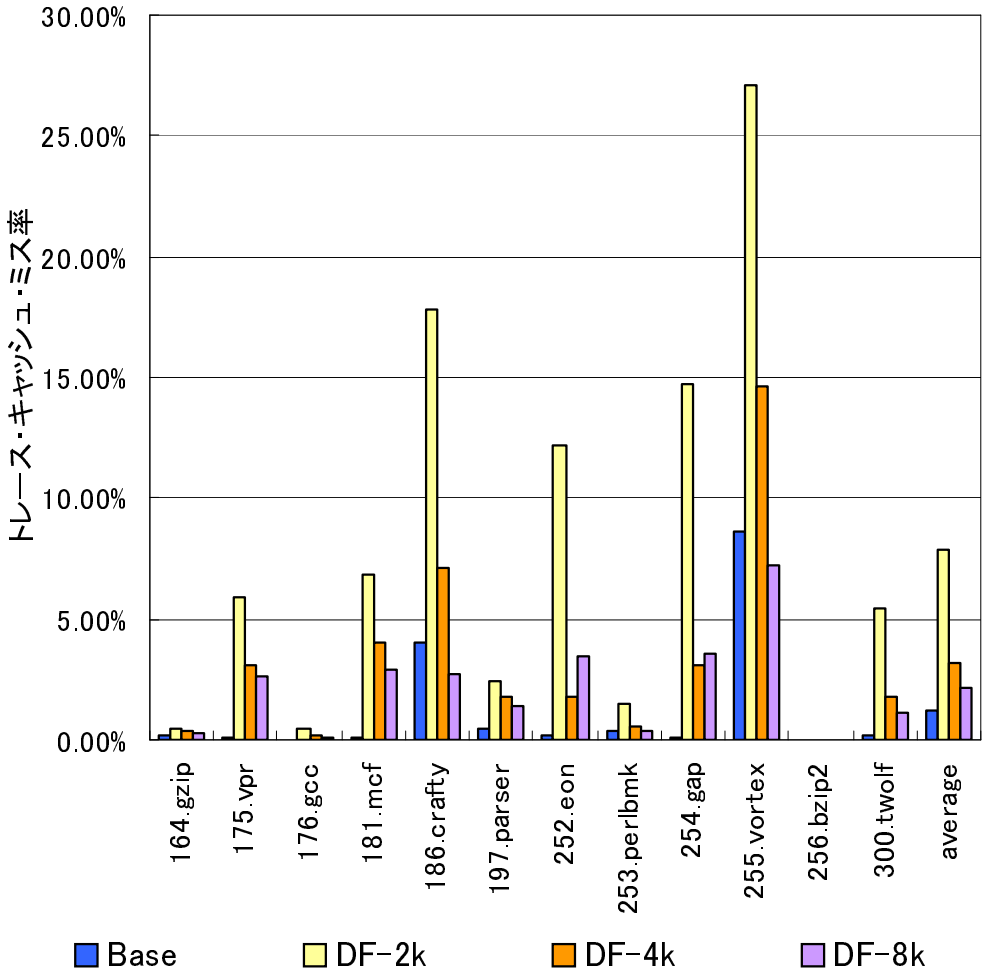

図 6.3: 逆 dualflow アーキテクチャのトレースキャッシュ容量とミス率

6.3 トレースキャッシュのウェイ数と容量によるトレースキャッシュミス率と IPC

5.10 節で述べたように，逆 dualflow アーキテクチャでは，変換結果のトレースをパスで区別することによりトレースの種類が増加するため，トレースキャッシュ・ミス率が増加し，IPC が低下してしまう．また，逆 dualflow アーキテクチャではトレースキャッシュのコンフリクトが起こりがちである可能性がある．

そこで，トレースキャッシュの容量とウェイ数を変化させてトレースキャッシュ・ミス率，IPC を測定した．**図 6.3**，**図 6.4** にウェイ数 8 の場合の，**図 6.5**，**図 6.6** にウェイ数 4 の場合の測定結果を示す．グラフの横軸がベンチマーク（右

端は平均）であり，ベンチマークごとの 4 本の棒グラフは，左からそれぞれ，ベース・モデル，トレースキャッシュのエントリ数が 2k，4k，8k の場合の逆 dualflow アーキテクチャに対応する．図 6.3，図 6.5 の縦軸は，全てのリタイアした命令に対するトレースキャッシュからフェッチできなかった命令の割合で計ったトレースキャッシュ・ミス率である．図 6.4，図 6.6 の縦軸は，ベース・モデルを 1 として正規化した IPC である

ウェイ数が 8 の場合は，トレースキャッシュの容量が 2k エントリのとき，ベース・モデルに対する IPC の低下は 0.5%と小さい．また，トレースキャッシュの容量を 4 倍の 8k エントリにすることによって，トレースキャッシュ・ミス率は 2.2%とベース・モデルの 1.8 倍に抑えられ，このときのベース・モデルに対する IPC 向上率は 4.3%である．

一方，ウェイ数が 8 の場合に比べ，ウェイ数が 4 の場合には，ベース・モデルに対してトレースキャッシュ・ミス率がより高くなってしまっている．トレースキャッシュの容量が 2k エントリでは，ベース・モデルに対する IPC の低下は 1.7%と大きくなっている．トレースキャッシュの容量を 4 倍の 8k エントリにした場合でさえ，トレースキャッシュ・ミス率は回復せず 4.0%とベース・モデルの 2.4 倍にもなっており，IPC 向上率も 3.2%と小さくなっている．このように，逆 dualflow アーキテクチャではベース・モデルに比べコンフリクト・ミスが多く起こっていることが分かる．5.8 節で述べたようなインデックス生成の工夫を行っても，このようにコンフリクト・ミスが起こりがちであることは問題である．

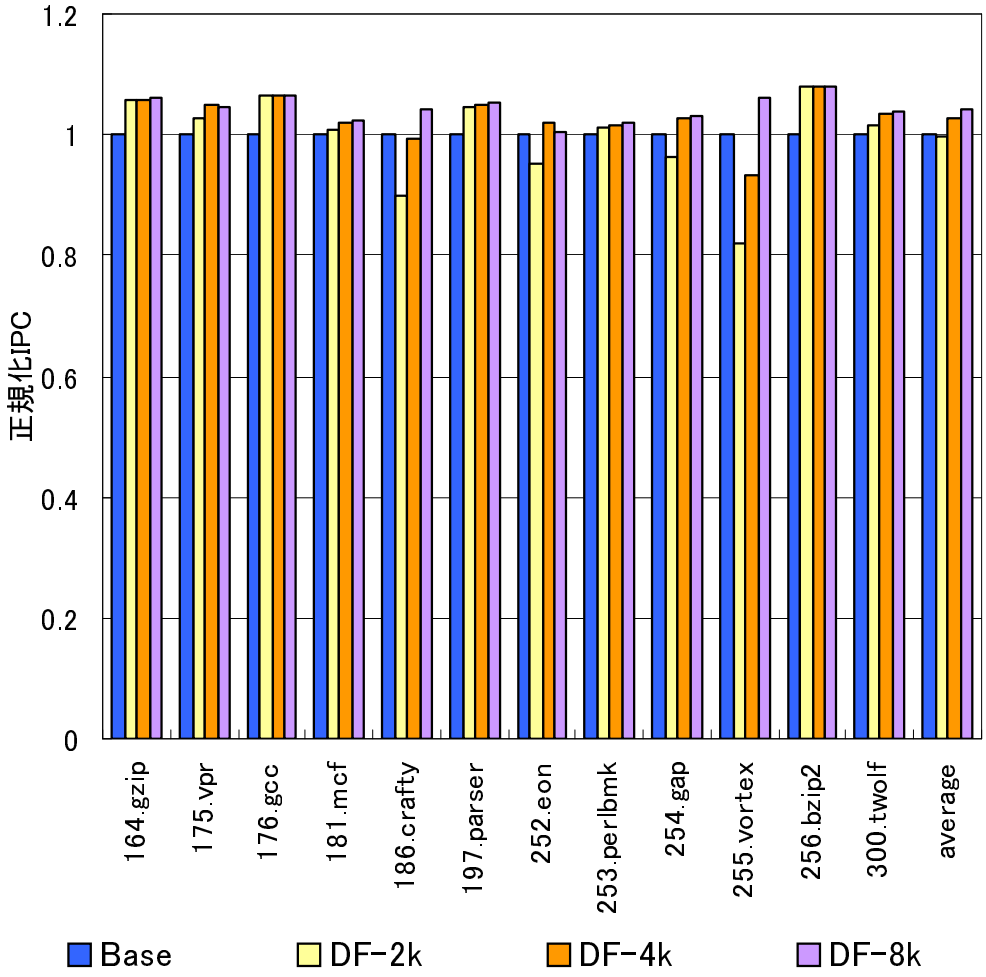

図 6.4: 逆 dualflow アーキテクチャのトレースキャッシュ容量と Base=1 とした正規化 IPC

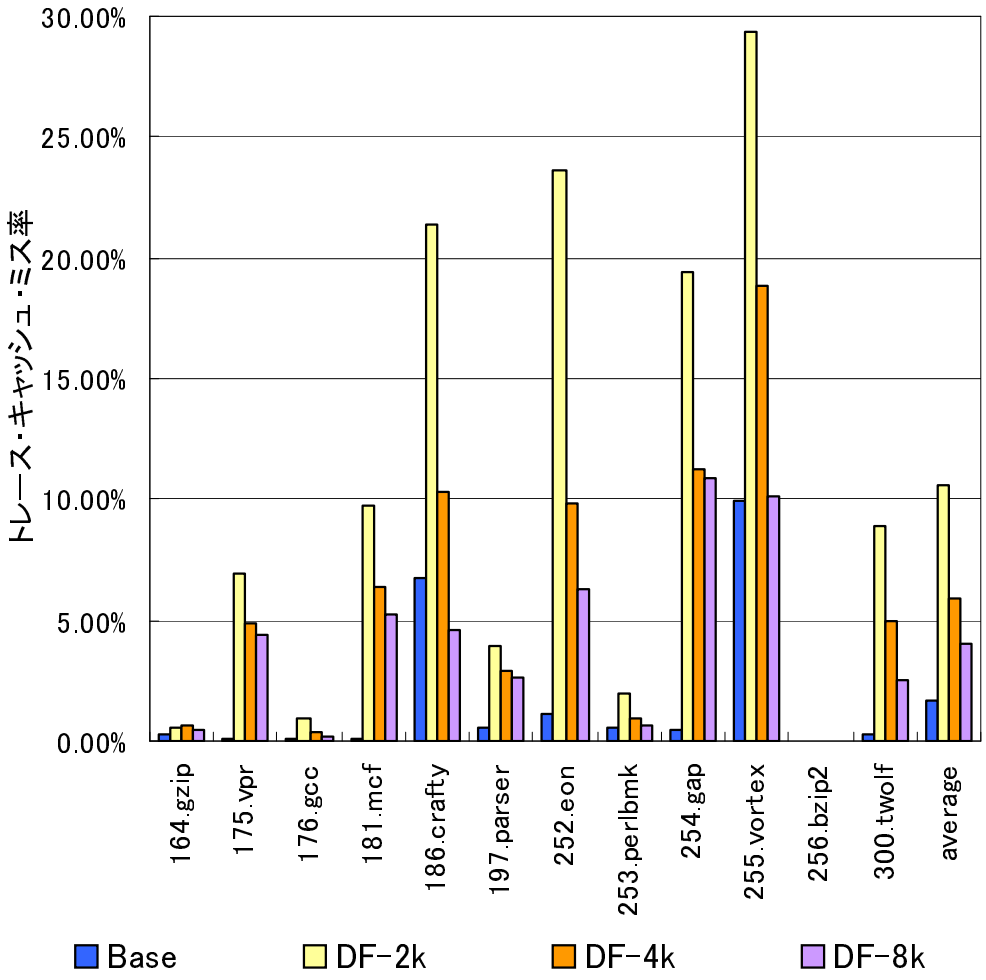

図 6.5: ウェイ数 4 の場合における逆 dualflow アーキテクチャのトレースキャッシュ容量とミス率

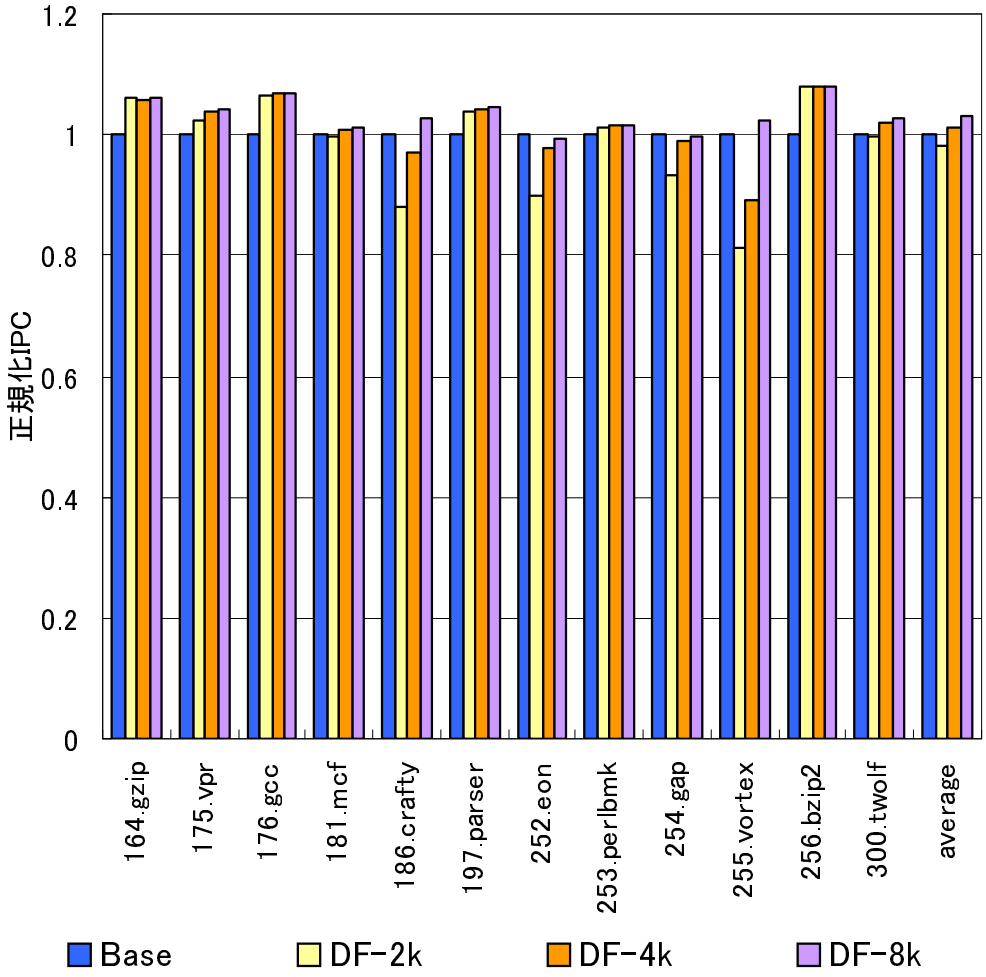

図 6.6: ウェイ数 4 の場合における逆 dualflow アーキテクチャのトレースキャッシュ容量と Base=1 とした正規化 IPC

第7章　まとめと今後の課題

本研究では，レジスタ・リネーミングを省略する手法として，逆dualflowアーキテクチャを提案し，性能への影響を評価した．逆dualflowアーキテクチャでは，命令のソース・オペランドをプロデューサへの変位に動的に変換してトレースキャッシュにキャッシュ・再利用することにより，レジスタ・リネーミングを省略することができる．レジスタ・リネーミングに用いるRMTへのアクセス頻度が高いため，レジスタ・リネーミングを省略することにより消費電力が低下すること，分岐予測ミス・ペナルティが減少すること，フロントエンドの幅を増やせるようになることが見込める．その代わり，変換した命令をパスによって区別して格納する必要が生じるため，トレースキャッシュ・ミス率が増加してしまうが，パイプラインが短くなる効果により性能への悪影響はなくなる．

今後の課題としては次のことがあげられる:

- 変換したトレースをパスによって区別する方法を工夫することにより，トレース数の増大を抑え必要なトレースキャッシュの容量を減らす
- 逆dualflowアーキテクチャではパスによって区別された同じアドレスのトレースのインデックスが衝突しがちであり，コンフリクト・ミスが増加してしまうので，これを軽減する
- 応用として，逆dualflowアーキテクチャではレジスタ・リネーミングが不要であるため，フロントエンドで行う処理に命令間での依存がない．これを利用して，トレースの長さとフロントエンドの幅を2倍にし，クロックを半分にすることにより，さらなる低消費電力を狙う

参考文献

[1] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita. A High-Speed Instruction Scheduling Scheme for Superscalar Processors. In *Proc. of the 34th MICRO*, pp. 225–236, 2001.

[2] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal Vol.5 Issue 1*, feb 2001.

[3] 五島正裕. Out-of-Order ILP プロセッサにおける命令スケジューリングの高速化の研究. PhD thesis, 京都大学, 2004.

[4] 五島正裕, グェンハイハー, 縣亮慶, 森眞一郎, 富田眞治. Dualflow アーキテクチャの提案. 並列処理シンポジウム JSPP 2000, pp. 197–204, 2000.

[5] 五島正裕, グェンハイハー, 縣亮慶, 中島康彦, 森眞一郎, 北村俊明, 富田眞治. Dualflow アーキテクチャの命令発行機構. 情報処理学会論文誌, Vol. 42, No. 4, pp. 652–662, 2001.

発表文献

主著論文

1. 逆 Dualflow アーキテクチャの提案
   一林 宏憲
   卒業論文，東京大学 工学部 (Feb. 2006)

2. 逆 Dualflow アーキテクチャ
   一林 宏憲，入江 英嗣，五島 正裕，坂井 修一
   情報処理学会報告 2006-ARC-169, 於高知商工会館, No.88, pp.37-42, Aug, 2006.

3. 逆 Dualflow アーキテクチャ
   一林 宏憲，入江 英嗣，五島 正裕，坂井 修一
   情報処理学会報告 2007-ARC-174, 於旭川市大雪クリスタルホール旭川国際会議場 , No.79, pp.1-6, Aug, 2007.

4. 逆 Dualflow アーキテクチャ
   一林 宏憲，塩谷 亮太，入江 英嗣，五島 正裕，坂井 修一
   先進的計算基盤システムシンポジウム SACSIS 2008
   （投稿中）

5. 逆 Dualflow アーキテクチャ
   一林 宏憲，塩谷 亮太，入江 英嗣，五島 正裕，坂井 修一
   情報処理学会論文誌コンピューティングシステム ACS 23
   （投稿中）

共著論文

1. 小容量 RAM を用いたオペランド・バイパスの複雑さの低減手法
   三輪 忍，一林 宏憲，入江 英嗣，五島 正裕，富田 眞治先進的計算基盤システムシンポジウム SACSIS 2007, pp.265-273, May, 2007.

2. プロセッサ・シミュレータ「鬼斬」の設計
   渡辺 憲一，一林 宏憲，五島 正裕，坂井 修一
   先進的計算基盤システムシンポジウム SACSIS 2007 (ポスター), pp.194-195, May, 2007.

3. 小容量 RAM を用いたオペランド・バイパスの複雑さの低減手法
   三輪 忍，一林 宏憲，入江 英嗣，五島 正裕，富田 眞治
   情報処理学会論文誌コンピューティングシステム (ACS 19), Vol.48, No.SIG3, pp.58-69, Aug, 2007.

謝辞

本研究を進めるにあたり，多くの方々から多大なご指導，ご協力をいただき大変お世話になりました．

坂井修一教授には，相談会等において研究内容に関する助言をいただきました．大変感謝しております．

五島正裕准教授には，至らない部分の多い私に対して，本研究の進行から論文の添削まであらゆる面においてご指導いただきました．本当にありがとうございました．

事務補佐員の八木原晴水さん，月村美和さんには研究を行う上での事務などでお世話になりました．

入江英嗣氏には，相談会等において，研究に関する助言をいただきました．

塩谷亮太氏には，本研究に関して重要な示唆をいただきました．

渡辺憲一氏は，本研究の評価に用いたプロセッサ・シミュレータ「鬼斬弐」の開発の中心であり，本研究の評価を行うにあたって大変お世話になりました．

その他にも，坂井研究室の皆様には研究や論文執筆，研究室での生活のサポートなど様々な面でご協力いただき，大変お世話になりました．