修士論文

フロントエンド・パイプラインを最小化する命令キャッシュ・アーキテクチャ

Instruction Cache Architecture that Minimizes Front-End Pipeline

平成25年2月6日提出

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

48-116432 伊達 三雄

概要

Out-of-order スーパースカラ・プロセッサのフロントエンドにおいて，命令間の依存解析を行うレジスタ・リネーミングと，命令をサブ・ウィンドウに分配するディスパッチは非常に高負荷なロジックである．これらのロジックに用いられるRMT と ディスパッチ・ネットワークは，それぞれリネーム幅と ディスパッチ幅の 2 乗に比例して面積と消費電力が増大する．

本稿ではこれらの高負荷なロジックを最小化する手法として Dispatched Image Cache(DIC) を提案する．DIC ではリネーミング及びディスパッチ済みの命令列をまとめてキャッシュし再利用する．依存解析結果は，後続の命令が依存元の命令を指定する形式で表現することで再利用を可能にし，ディスパッチ情報はサブ・ウィンドウに合わせて DIC の領域を区切ることで再利用可能にする．DIC ヒット時には，得られたイメージをそのままサブ・ウィンドウに格納すればよく，高負荷なレジスタ・リネーミングとディスパッチを行う必要がない．ミス時にのみ通常のリネーミングとディスパッチを行う．その際のリネーム幅やディスパッチ幅を最小限にすることで，これらのロジックの負荷を最小限に抑えながら性能を保つことができ，面積効率が向上する．

評価の結果，通常の命令キャッシュと同程度のキャッシュ容量で性能低下を 1% 近い性能向上を実現しつつ，これらのロジックの負荷を最小限にできることが確認できた．

目 次

図 目 次

表 目 次

第1章　はじめに

近年では，単一のチップ上に複数のプロセッサ・コアを集積するマルチコア・プロセッサが広く普及している．マルチコア・プロセッサの時代においては，コアの面積効率 (回路面積あたりの性能) がより重要な意味を持つ．面積効率の向上は，シングルコア・プロセッサではチップ面積を削減するだけであるが，マルチコア・プロセッサではコア数の増加による最大性能の向上につながるからである．

コアとして用いられるスーパースカラ・プロセッサの性能を向上させる一次的な方法は，そのウェイ数を増やすことである．しかし一般的なスーパースカラ・プロセッサの制御部 (非演算器部) の回路面積は，ウェイ数に対して 2 乗から 3 乗に比例して増加する [11]．これは，スーパースカラ・プロセッサの制御部の大部分が RAM/CAM で構成されており，それらの面積がポート数の 2 乗に比例するためである．ウェイ数を増やし多数の命令を同時に実行するためには，RAM/CAM に対して多数のアクセス・ポートが必要となり，回路面積は非常に大きくなってしまう．したがって単純にウェイ数を増加させることは，コアの面積効率の低下を招いてしまう．

本稿ではこのようなスーパースカラ・プロセッサの制御部の中で，フロントエンド・パイプラインに属する**レジスタ・リネーミング**と**ディスパッチ**のロジックに着目する．

レジスタ・リネーミングとは命令間の依存を解析する処理である．レジスタ・リネーミングには，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)** が用いられる．詳しくは 2.2 節で述べるが，RMT には通常，1 命令につき 4 本のポートが必要となる．リネーム幅 (同時にリネーミングを行う命令数) が 4 であれば，ポート数は 16 にもなる．RAM/CAM の面積はポート数の 2 乗に比例するため，RMT の面積は非常に大きなものとなる．また，このロジックは毎サイクル，ほぼすべての命令に対して処理を行うので，利用頻度も高い．このような面積の大きいロジックに頻繁にアクセスすると，その消費電力も非常に大きなものとなる．

命令ディスパッチは，各命令を対応する命令ウィンドウに分配・格納する処理である．命令ウィンドウは通常，整数，ロード/ストア，浮動小数点といった演算器の種別ごとのサブ・ウィンドウに非集中化される．そのため，リネーミング後の命令を適切なサブ・ウィンドウに分配するディスパッチ・ネットワークが必要となる．2.3 節で述べるように，ディスパッチ・ネットワークの面積もディスパッチ幅 (同時にディスパッチを行う命令数) の 2 乗に比例して増加する．このロジック

も RMT と同様に利用頻度が高いため，消費電力も大きくなる．

本稿では，これらのリネーミング・ロジックとディスパッチ・ネットワークに着目し，フロントエンド・パイプラインを簡略化する手法として **Dispatched Image Cache(DIC)** を提案する．DIC は端的には「リネーミング及びディスパッチ済み」の命令列をキャッシュする命令キャッシュであるといえる．リネーミング済みの命令を再利用する手法として，先行研究において **Renamed Trace Cache(RTC)** が提案されている [9]．3.2 節で詳しく述べるが，RTC は，後続の命令が，依存元の命令を相対的な変移で指定する形式で依存解析を行うことでリネーミング情報の再利用を可能にしている．DIC では，RTC の考え方をさらに推し進め，依存解析に加え，サブ・ウィンドウへの分配情報もキャッシュする．そのために，DIC はサブ・ウィンドウに合わせてキャッシュ領域が区切られ，対応するサブ・ウィンドウと直結される．

DIC にヒットした場合，基本的には，得られたイメージをそのままサブ・ウィンドウに格納すればよく，高負荷なレジスタ・リネーミングとディスパッチを行う必要がない．ミスした時にはレジスタ・リネーミングとディスパッチ・ネットワークの処理を行ってイメージを構成するが，これは小規模のロジックによって時間をかけて行えばよい．その結果，性能を保ちながらリネーミング・ロジック，およびディスパッチ・ネットワークに要する回路面積と消費電力を大きく削減することができ，面積/電力効率の向上を見込める．

本稿の構成は以下の通りである．2 章では一般的なスーパースカラ・プロセッサのフロントエンド・パイプラインにおけるレジスタ・リネーミングとディスパッチについて述べる．3 章と 4 章では本稿の提案手法である DIC について述べる．3 章では DIC の基本的な概念を，4 章ではより詳細な実装と解析を行う．5 章で提案手法の評価を行う．最後に 6 章で，まとめと今後の方針を述べる．

第2章　フロントエンド・パイプライン

本章では，一般的なスーパースカラ・プロセッサにおけるフロントエンド・パイプラインについて述べる．特にレジスタ・リネーミングとディスパッチの役割，およびこれらの負荷を明らかにする．

2.1　フロントエンド

Out-of-order スーパースカラ・プロセッサは基本的にはフロントエンドとバックエンドと呼ぶ部分に分けられ，それらが命令ウィンドウというバッファで結合された構成と考えることができる．**図 2.1** にそのようなスーパースカラ・プロセッサの基本構造を示す．同図はスーパースカラ・プロセッサの基本的なロジックと，それを用いるパイプライン・ステージの対応を表している．フロントエンド部分では命令の供給や命令の解析を行う．解析済みの命令は命令ウィンドウというバッファに蓄えられ，バックエンドで命令を Out-of-order に実行する，という流れになる．図 2.1 ではフロントエンドのパイプラインを

- フェッチ：命令キャッシュからの命令読み出し．分岐予測
- リネーム：命令間の依存解析
- ディスパッチ：命令ウィンドウへの分配/供給

という主要な 3 つのフェーズに簡略化して示している．以降，本章では，本研究で特に着目しているレジスタ・リネーミングとディスパッチについて述べる．

2.2　レジスタ・リネーミング

レジスタ・リネーミングは命令間の依存解析を行う処理である．すなわち，命令間の偽の依存を解消し，真の依存を明らかにする．

一般的にレジスタ・リネーミングは，命令セット・アーキテクチャで指定される**論理レジスタ**を，実行時に値を保持する**物理レジスタ**にマッピングすることで偽の依存を解消する．このマッピングは，**RMT** と呼ばれる表を用いて行われる．

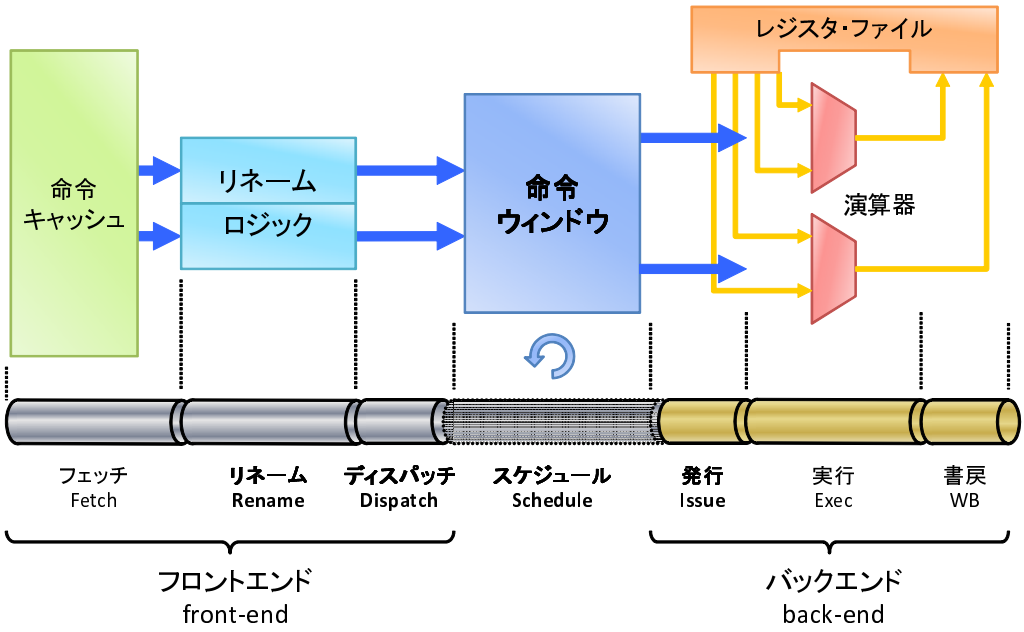


図 2.1: スーパースカラ・プロセッサの基本的な構成

通常 RMT は RAM や CAM[1] によって構成され，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．そしてこの RMT は非常に高負荷なロジックとなっている．

レジスタ・リネーミングの負荷　RAM で構成された RMT を用いた一般的なレジスタ・リネーミングを考える．ソース・オペランド 2 つ，デスティネーション・オペランド 1 つを持つ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新たに割り当てられた物理レジスタ番号の書き込みに 1 ポート
- 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に 2 ポート
- デスティネーション論理レジスタに割り当てられていて，必要なくなった物理レジスタを解放するための読み出しに 1 ポート

が必要であり，リネームする命令 1 つにつき RMT のポートは 4 本必要である．リネーム幅 4 のスーパースカラ・プロセッサを仮定すると RMT のポート数は 16 と

なる．RAM/CAM の面積はポート数の 2 乗に比例するので，RMT の回路面積は 16 の 2 乗に比例する非常に大きなものとなる．

また，RMT に対するアクセス頻度は非常に高い．これは，リネーミング・ロジックはスーパースカラ・プロセッサのパイプラインのフロント・エンドに属しており，基本的にすべての命令を処理する必要があるからである．たとえば，レジスタ・ファイルも概念的には RMT と同様に，オペランド 1 つにつき 1 回アクセスを行うが，

- 投機ミスのためリネームは行われたがレジスタにアクセスしない命令が存在する
- 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
- RMT は物理レジスタ解放のための読み出しが必要である

などの違いにより，RMT はレジスタ・ファイルと比べてもアクセス頻度が高くなる．RMT のような多ポートの RAM に対して高頻度でアクセスを繰り返すと，消費電力や熱の問題も深刻となる．

以上のように RMT はリネーム幅の 2 乗に比例して面積が大きくなるため，回路規模・消費電力の点から，フロント・エンド全体のスループットにも制約を与えている．逆にレジスタ・リネーミングを省略することができれば，RMT の面積を大幅に削減し，消費電力や発生する熱量を削減することができ，スループットに対する制約を緩和することも可能となる．

2.3 ディスパッチ

スーパースカラ・プロセッサにおけるディスパッチとは，各命令を対応する命令ウィンドウに分配・格納する処理である．

一般的に命令ウィンドウは整数演算，ロード/ストア，浮動小数点演算といった演算器の種別ごとのサブ・ウィンドウに非集中化される．これは命令ウィンドウを非集中化することで，ロジックの実行サイズの縮小と，クリティカルパスの分離の効果が得られるためである [11]. このサブ・ウィンドウに，対応する命令を分配する処理がディスパッチである．

以降では，同時にディスパッチ可能とする命令数を**ディスパッチ幅**と呼ぶこととする．また命令ウィンドウは，上述のような整数 (INT)，ロード/ストア (MEM)，浮動小数点 (FP) の 3 つのサブ・ウィンドウを仮定する．

図 2.2 にディスパッチ幅 4 のときのディスパッチ・ロジックを示す．図 2.2 はフェッチ幅 4 として，命令キャッシュから命令をフェッチしてから，レジスタ・リネーミングを行い，命令ウィンドウにディスパッチされるまでの一連の回路を表している．図 2.2 のリネーミング・ロジックから出ている 1 本の線は，リネーム済みの 1 命令

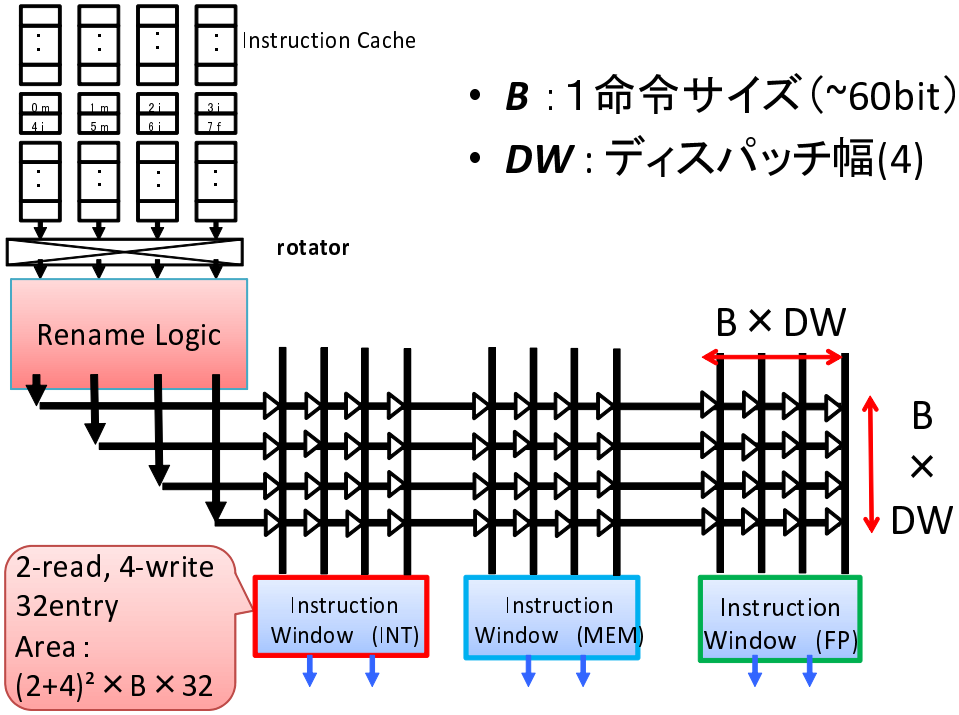


図 2.2: ディスパッチ・ネットワークの複雑性

を表している．リネーム済みの命令サイズは 60bit 程度に拡張されるため，これらの線はそれぞれが 60 本程度のビット・ラインをまとめて表していることとなる．

命令キャッシュ内の命令ミックスは，各種類の命令が不規則に入り混じって格納されている．したがって 4 命令すべてが同一種類の命令であったときにも対応するために，各命令はすべてのサブ・ウィンドウにセレクタを通して繋がる必要がある．そのためには図 2.2 のように，交点にセレクタを配した格子状のロジックが必要となる．このようなディスパッチのためのロジックを**ディスパッチ・ネットワーク**と呼ぶこととする．

ディスパッチの負荷　各サブ・ウィンドウは，基本的にディスパッチ幅分の書き込みポートと，**発行幅**分の読み出しポートを備えた RAM で構成される．ただし実際には，サブ・ウィンドウへの書き込みは連続した領域に書き込まれるため，図 2.2 のようにランダム・アクセス可能な書き込みポートを 4 つ増やす必要はない．命令ウィンドウもバンクを行うことで，ロジックの簡略化が図れる．

以上のようなディスパッチ・ネットワークの回路面積は，命令ウィンドウの幅，ディスパッチ幅，リネーム済み命令のサイズの積で表される．命令ウィンドウの幅もディスパッチ幅に比例するため，図 2.2 におけるディスパッチ・ネットワークの面積は $B^2 \times DW^2 \times 3$ で表される．すなわちディスパッチ・ネットワークの面積もディスパッチ幅の 2 乗に比例して増大する．各命令のサイズも 60bit 程度に拡

張されているので，ディスパッチ・ネットワークは非常に大きなものとなり，単純に計算すると命令ウィンドウのペイロードRAMの3倍近い面積となっている．

またこのロジックもリネーミング・ロジックと同様にパイプラインのフロント・エンドに属し，すべての命令が利用するため，利用頻度が高く，消費電力や発熱量も大きくなる．このディスパッチ・ネットワークを省略することができれば，レジスタ・リネーミングの省略と同様に，回路面積を大幅に削減でき，消費電力や熱の面からも大きな改善が得られる．またこれら2つのロジックを同時に除去できれば，フロント・エンドのスループットに対する制約がかなり弱くなる．

第3章　Dispatched Image Cache

本章と 4 章では提案手法である **Dispatched Image Cache(DIC)** について説明する．本章では DIC の基本的な概念を説明し，4 章では DIC の性能を高める手法に関して詳しく論じる．

3.1　DIC の基本的な考え方

DIC は，リネーミングが完了し，各サブ・ウィンドウにディスパッチされた後の命令列の「イメージ」を格納する命令キャッシュである．DIC ヒット時には「リネーミング及び ディスパッチ済み」の命令列が得られるので，通常のレジスタ・リネーミングとディスパッチの処理を行う必要がない．DIC ミス時にのみ通常のレジスタ・リネーミングを行い，サブ・ウィンドウへの分配を行った命令列をまとめて DIC へ格納するが，このときのリネーミングとディスパッチは小規模のロジックによって時間をかけて行えばよい．すなわち DIC ミス時のリネーム幅とディスパッチ幅は最小限に抑えることで，性能の低下は防ぎつつ，RMT とディスパッチ・ネットワークの負荷を最小化することができる．

このような DIC は，

- リネーミング情報を再利用するために
  - レジスタ参照モデルの変更を行い，命令を再利用可能な命令形式 (dualflow 形式) に変換する
  - 命令の実行経路毎に区別してキャッシュする
- ディスパッチ情報を再利用するために，DIC の命令格納領域は各サブ・ウィンドウに合わせて区切り，対応するサブ・ウィンドウと直結させる

という大きく 2 つの要素から構成されており，本章で順番に説明する．

図 3.1 は以上のような DIC の構成を表す．区切られたキャッシュ領域には対応する種類の命令のみを格納するので，サブ・ウィンドウと直結させることができ，ディスパッチ・ネットワークは大幅に簡略化される．さらに図 3.1 では，各命令種別用のキャッシュ領域のバンク分けを行い，2 命令ずつ最大 6 命令同時にフェッチ (ディスパッチ) できるような DIC の構成をとっている．また図 3.1 では，DIC ミス時にフロント・エンドを流れる命令数を 1 としている．これによって DIC ミス

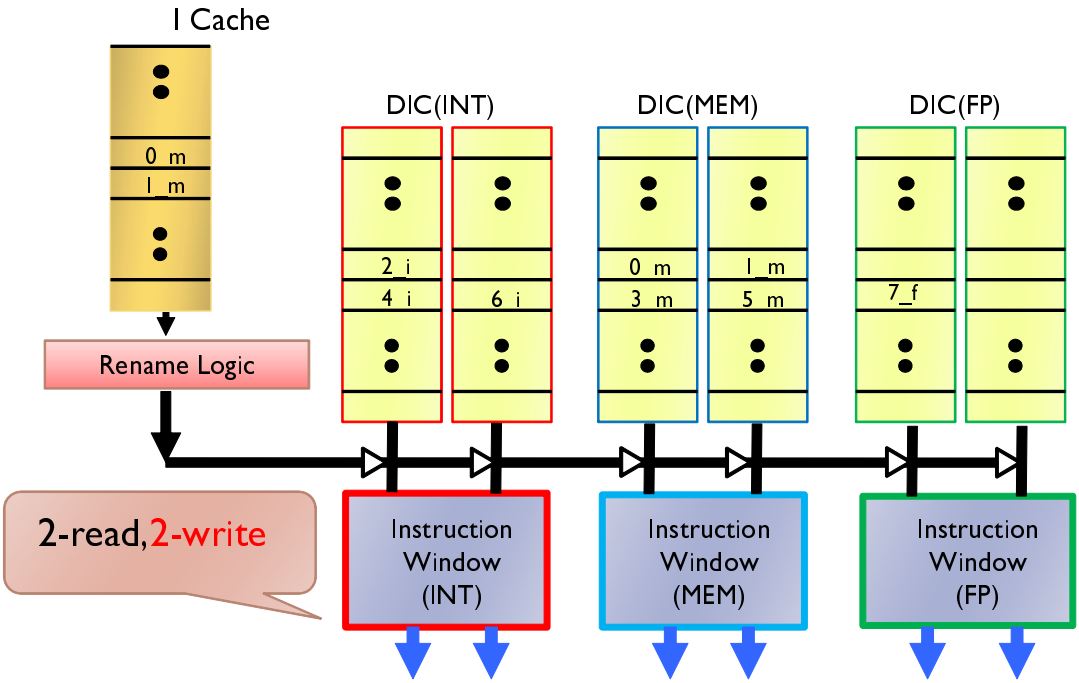


図 3.1: ディスパッチ・ネットワークの省略

時のリネーム幅とディスパッチ幅は最小限に抑えられており，その負荷も最小化されている．

3.2 リネーミング情報の再利用

DIC は依存解析結果の再利用を行うが，通常のレジスタ・リネーミングではリネーミング結果を再利用することはできない．なぜなら通常のレジスタ・リネーミングでは，同じ命令に対しても異なるマッピング情報を与えることがあるからである．これは，

- 通常のレジスタ・リネーミングは，フリーリストからそのとき使用可能な物理レジスタ番号を割り当てる
- 同一の命令に対しても，それまでの実行経路によって依存する命令が異なることがある

からである．したがって通常のリネーミングには再現性がなく，その変換結果を再利用することは不可能である．

これを再利用可能にするために DIC ではレジスタ参照モデルを変更し，命令の実行経路毎に区別してキャッシュする．

3.2.1 DICのレジスタ参照モデル

DICでは通常のリネーミングと同様に，物理レジスタ・ファイルと論理レジスタ・ファイルという2種類のレジスタ・ファイルを用いるが，それらの参照の仕方が通常とは異なる．

DICの物理レジスタ・ファイルは，サイクリックに使用されるリング状の構造を取る．この各エントリには，各命令のデスティネーション・オペランドがシーケンシャルに割り当てらる．こうすることで物理レジスタを読み出す際は，自命令に割り当てられた物理レジスタのインデックスに 依存元を指定する」変位を加算するだけで，読み出すエントリを決定できる．また物理レジスタ・ファイルの各エントリには，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．

論理レジスタ・ファイルはリタイア済みの命令の実行結果を保持しておくために用いられる．命令の実行結果は，実行完了したものから out-of-order に物理レジスタ・ファイルに格納する．その後命令が in-order にリタイアする際に，物理レジスタの先頭のエントリから，付随された論理レジスタ番号を用いて，論理レジスタ・ファイルへ順番にコピーする．

そしてDICでは依存解析結果を,「後続の命令が依存元の命令を指定する形」で表す．基本的には，依存元の命令が何命令前に実行されたか，という物理レジスタ上の相対的な変位でその指定を行う．この情報は同一の実行経路を通った場合は常に一定であり，再現性がある．以下ではこの変換形式を **dualflow 形式**と呼ぶ．

図 3.2 にdualflow形式に変換された命令の表現形式を例示する．$DstReg$でデスティネーション・オペランドを論理レジスタ番号で指定する．$SrcL/SrcR$でソース・オペランドを指定する．ソース・オペランドは，前述の通り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依存元を指定する形式に内部的に変換される．ただし，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．そのため，RL/RRが1のとき$SrcL/SrcR$は依存命令への変位を表し，RL/RRが0のとき$SrcL/SrcR$は論理レジスタ番号を表すこととする．ここで，最大参照距離 (nの最大値) はリオーダ・バッファ(アクティブ・リスト)・サイズ (ROBS) と一致する．ROBS以上前の命令はリタイア済みであることが保証されるからであり，その実行結果は論理レジスタから得ることとなる．逆にROBS以内の命令は実際にはリタイア済みであっても，それが保証されるわけではないので，物理レジスタから得られるようにする．そのために物理レジスタのエントリ数はROBSの2倍用意する必要がある．

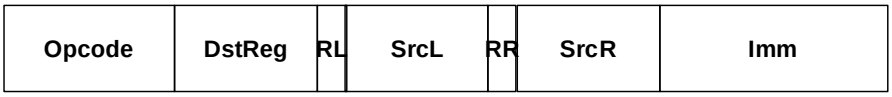


図 3.2: Dualflow 形式の命令表現

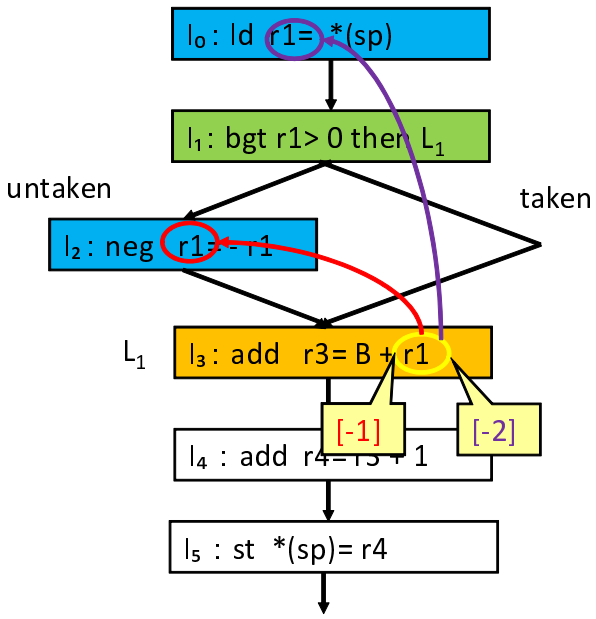


図 3.3: 実行経路による変換結果の変化

3.2.2 依存経路毎の区別

DIC では，命令の実行経路毎に区別してキャッシュすると述べた．ここでは，経路毎に区別する意味とその手法を述べる．

まず**図 3.3** に実行経路によって依存解析結果が変化する命令列の例を示す．分岐命令 $I_1 : bgt$ が $untaken$ であれば，$I_3 : add$ 命令のソースオペランド $r1$ は 1 命令前に依存しているため，変換結果は $[-1]$ となる．$taken$ であれば，neg 命令は実行されないため add の参照距離は $[-2]$ となる．したがって $I_3 : add$ 命令から始まる命令列をキャッシュするとき，それらが同一 PC の命令であっても，bgt の結果によって区別してキャッシュしなければならない．

パス　命令の実行経路を識別し，区別してキャッシュするための経路情報をパスと呼ぶこととする．DIC ではこのパスをタグアレイとして用いて依存経路を区別することを可能にする．

パスの表現としては，経路上にあるすべての命令アドレスを用いれば一意に識別可能である．しかしそれでは冗長であるので情報量の圧縮を行う必要がある．一般にある命令の次に実行される命令は，ある命令が

- 条件分岐命令のとき：分岐の成立/不成立によって一意に決まる
- 間接分岐命令のとき：間接分岐のターゲットによって決まる
- これら以外の命令のとき：常に一意に決まる

以上より，経路を一意に特定するためには，依存経路の先頭 *(*同一トレース内の命令が依存している，最も古い命令*)* のアドレスと，そこからの全ての条件分岐の成否，間接分岐のターゲットの情報を用いればよいということになる．そのために，*PC* 履歴と分岐情報を命令実行時に保持しておき，*DIC* へのアクセス時に必要となるパス情報を取り出す．このタグを比較することによって，同一の命令アドレスを持つが，依存解析結果が異なるトレースを識別することができる．

しかし，本当に経路上の全ての分岐情報を保持しようとすると，すべての命令アドレスを保持するコストと変わらなくなってしまう．そこで，パスの情報量に応じて *DIC* への格納には制限を設けることとしている．例えば，パスとして格納できる間接分岐のターゲット数の上限を決めてしまい，それを超過する依存経路を持つ命令は *DIC* へ格納しない，という手法がとれる．

3.2.3 リネーミング情報の再利用に伴う DIC の性能への影響

以上のようにパス情報をタグとして用いるため，通常の *TC* と比べてタグアレイは大きくなる可能性がある．逆に格納する条件を厳しくしすぎると，*DIC* に格納できる命令数が減少し，全体の処理性能が低下してしまう可能性もある．このようなパスの情報量と *DIC* の性能の関連性については，*4.2.2* 節でより詳しく考察を行う．

また，同一の命令列であっても実行経路によって区別するため，*DIC* のキャッシュ利用効率は低下することとなる．この経路区別による利用効率の低下についても *4.2.2* 節で考察を行う．

3.3 ディスパッチ情報の再利用

3.1 節で述べたように，図 *3.1* のように *DIC* を構成することでディスパッチ情報は再利用可能となる．すなわち *DIC* の命令格納領域は各サブ・ウィンドウに合わせて区切り，対応するサブ・ウィンドウと直結させる．そして，区切られたキャッシュ領域には対応する種類の命令のみを格納するので，フェッチ時には複雑なロジックを介さずにディスパッチ可能となっている．

INT		MEM		FP	
0_i	1_i	–	–	–	–
2_i	4_i	3_m	X	5_f	X
6_i	7_i	8_m	–	–	–
9_i	10_i				

図 3.4: ナイーブなキャッシュ・ライン生成方法

3.3.1 キャッシュ・ライン形成モデル

このようにキャッシュ領域を区切ることで，*DIC* のキャッシュ・ラインには埋まらないスロットが生じる可能性がある．これは命令ミックスには偏りがあるからである．例えば，*INT* 系の命令がしばらく連続する場合もあれば，*INT* 系と *MEM* 系の命令が交互に出現する場合も考えられる．このような場合，命令の格納方法によってはキャッシュ・ラインに空きスロットが生じたり，命令が実行順に並んでいないこととなる．

ここで *DIC* のキャッシュ・ラインをどのように形成し，その際にどのようにして正しい実行結果を得るか，ということを考える必要がある．

基本的なキャッシュ・ライン形成モデル　ディスパッチされた命令を**命令順に**一定数まとめることが基本的なキャッシュ・ライン形成方法となる．その具体的な例を**図 3.4** を用いて説明する．図 *3.4* は *DIC* の内部状態を表しており，各数字は格納されている命令の順番，添え字は命令の種類を表している．0_i から 10_i の命令が順に出現した際に，それぞれのサブ・ウィンドウに合わせて，*INT*，*MEM*，*FP* と領域を区切った *DIC* に格納する．横一列がキャッシュ・ラインであり，この命令列を *1* つのディスパッチ・グループとする．図 *3.4* では {*0_i*，*1_i*} や，{*2_i*，*4_i*，*3_m,5_f*} が同一キャッシュ・ライン上の命令であることを表す．実際には連続したキャッシュ・ラインを，*DIC* 上で物理的に連続したエントリに確保する必要はないが，説明のためにこのように表わしている．この手法では

- ラインをまたいで命令を追い越さない

というルールに従ってキャッシュ・ラインを形成している．すなわち *1* 種類のキャッシュ領域が埋まれば，同一ラインの他のキャッシュ領域が空いていてもライン形成を完了することとしている．図 *3.4* 上では 8_m は 3_m の隣に詰めたりせず，5_f もその上のラインに格納しないということである．もしこれらの命令を詰める場合，フェッチ順とプログラム・オーダが一致しなくなる．そのようにしないこの手法では，同一キャッシュ・ライン内でしかプログラム・オーダが乱れ得ないという点で

簡単である．このモデルでは，同一キャッシュ・ライン内で自身が何番目の命令であるかを表す *3bit* を付け加えればよい．

3.3.2 ディスパッチ情報の再利用に伴う DIC の性能への影響

以上のように *DIC* では命令種別毎に格納領域を区切るため，命令ミックスの偏りによっては利用効率が低下してしまう可能性がある．それは *DIC* の容量に無駄が生じるというだけでなく，*DIC* にヒットした際のスループットの低下に繋がることも意味している．例えば極端に *INT* 系の命令のみが連続した場合，図 *3.1* の *DIC* のスループットは最大 *2* 命令となってしまう．これはディスパッチ幅 *4* の場合と比べると，半分程度に供給能力が低下してしまうことを意味する．

これらの，命令ミックスの偏りと *DIC* の利用効率の関連性については，*4.1* 章で詳しく考察を行う．

3.4 DIC のパイプライン

図 *3.1* のように，*DIC* ヒット時にはフロント・エンドの複雑な処理を必要とせずに，直接ディスパッチできる．したがって，ヒット時は通常のフロントエンドのパイプライン・ステージ分を短縮することができる．これによって分岐予測ミス等のペナルティが削減され性能向上が図れる．ただし，この恩恵は分岐予測ミス後に *DIC* ヒットした場合にのみ受けられる．逆にパイプラインの短縮を行う設計で，ヒット/ミスの切り替えが多発してしまうと，パイプラインにバブルが発生してしまい，性能低下の原因となる *[7, 6]*.

したがって，*DIC* のパイプライン構成としては

- ヒット時のパイプラインを短縮する
- ミスを仮定したパイプライン構成にし，短縮を行わない

という *2* 種類の設計が考えられる．

短縮のメリットがデメリットを上回る条件は，*DIC* のヒット率をかなり高くし，またヒット/ミスの切り替えが発生しにくいようにコントロールできることである．このようなときには，*[3]* の *TC* のように，*DIC* を *L1* 命令キャッシュとして用い，稀なミス時には *L2* キャッシュまでアクセスする，という構成を実現することも可能になる．

逆に *DIC* がある程度ミスをすることを許容し，*DIC* ミス時のペナルティを最小限にできるならば，パイプラインの短縮は行わない方が良い．キャッシュ階層の構成としては，*[7]* のように，*DIC* と *L1* 命令キャッシュが並列に存在するような構成が現実的なものとなる．

3.5 DICの効果

以上が*DIC*の概要である．その利害得失をまとめる．*DIC*ヒット時は，*DIC*から複雑な論理を一切介さずにサブ・ウィンドウに格納できる．*DIC*ミス時のフェッチ幅を*1*にするならば，図*3.1*のように*RMT*とディスパッチ・ネットワークが最小化され，フロントエンドの負荷を大幅に削減できる．図*2.2*における命令キャッシュのローテータも不要となっている．

また，*DIC*ヒット時にはリネームやディスパッチに充てられていたパイプラインステージを省略することができる．パイプラインの段数が短くなると分岐予測ミス等のペナルティを削減することができる．さらに*DIC*のライン・サイズや(INT, MEM, FP)の比率は，フロント・エンドのロジックの制約を受けずに自由に設定することができ，高いスループットを実現することも可能となる．

ただし*DIC*ミス時のフェッチ幅が少数であっても性能が落ちないようにするためには，十分な*DIC*ヒット率，スループットが要求される．しかし*DIC*では，パスによって命令列が区別されることと，命令出現パターンの偏りによってキャッシュ利用効率が低下する可能性もある．

第4章　Dispatched Image Cache の利用効率の解析

*3*章で述べたように，*DIC*は通常の命令キャッシュや *TC*と比べると

- 命令種類毎に格納領域を区切る
- 依存経路毎に分けて命令列を格納する

という点で利用効率が低下してしまう可能性がある．それによって *DIC*自体の容量が大きくなってしまうと，フロントエンド・パイプラインの負荷を削減することで得られる利点が損なわれてしまう．したがって *DIC*の容量は最小限にしつつ性能が低下しない構成を考える必要がある．

本章では *DIC*の利用効率がどのような要因で，どのように低下するかを解析し，*DIC*の利用効率を向上させる手法を論じる．本章において示されているデータは全て，*5.1*章に示される評価環境で行っており，命令セット・アーキテクチャは *Alpha[4]*ベースのもので，*SPECCPU 2006[8]*の *ref* 入力データ・セットをベンチマークとして用いている．

4.1　命令ミックスの偏り

*3.3.1*節で述べたように，命令ミックスの偏りよっては *DIC*のキャッシュ・ライン上に空きスロットが生じることがあり，*DIC*の利用効率は低下してしまう．

4.1.1　動的命令数の偏り

まず，評価プログラムにおける動的な命令数の計測を行った．評価は動的な命令系列に対して，各サブ・ウィンドウにディスパッチされる命令を *[INT,MEM,FP]*の種別毎にカウントし，その出現割合を計測した．**図 4.1** にその測定結果を示す．*SPECCPU 2006_INT(12*本*)*，*SPECCPU 2006_FP(17*本*)*，及び *SPECCPU 2006* 全体 *(29*本*)* のそれぞれの平均値に関して計測結果を示している．全体平均に対して *FP*が占める割合は *10%* 程度であり，圧倒的に少ないことがわかる．最も *FP*の占める割合の多かったプログラムは *470.lbm*で *60%* であったが，その他は *FP*系のプログラムでも *20%* 程度という低い割合を示した．

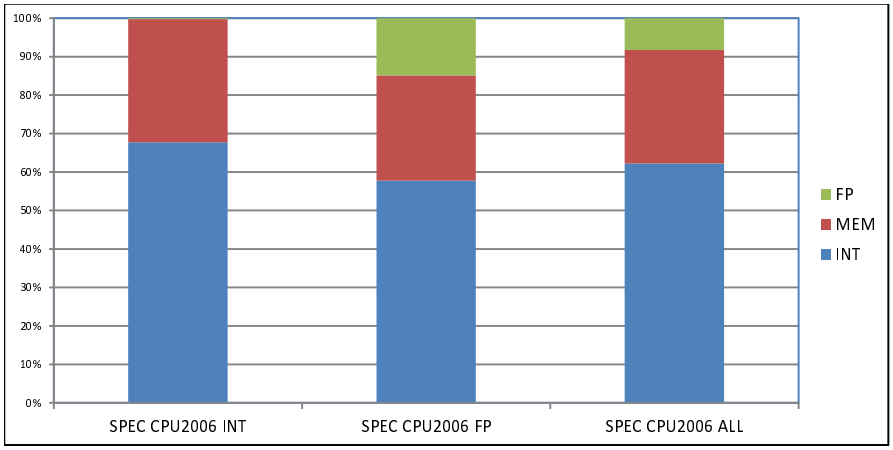


図 4.1: 命令出現頻度の偏り

INT			MEM		FP
0_i	1_i	2_i	3_m	X	–
4_i	6_i	7_i	8_m	X	5_f
9_i	10_i				

図 4.2: キャッシュ・ラインの命令スロットに偏りを持たせた場合

この結果から，*DIC* のキャッシュ領域を区切る際に，容量を均等にしてしまうと *INT* 領域が圧倒的に不足し，*FP* 領域が余るということが考えられる．*DIC* はこれらのキャッシュ領域を均等に分割する必要はなく，容量に対して偏りを持たせることでこれに対応できる．

簡単には，

- *DIC* のキャッシュ・ラインあたりの命令割合を変化させること
- *DIC* の各キャッシュ領域のライン数 *(*セット，ウェイ数*)* の割合を変化させること

によって *DIC* の容量に偏りを持たせることができる．

キャッシュ・ライン上のスロット数　図 *3.1*，図 *3.4* では，同一キャッシュ・ライン上の $([INT, MEM, FP])$ の命令スロットは $(2, 2, 2)$ の最大 *6* 命令で構成している．全体的に *INT* 系の命令が多いのであれば，これを**図 4.2** のように $(3, 2, 1)$ とすることで容量に偏りを持たせることが可能である．

この際に，単純に総容量が *3:2:1* になるだけでなく，*DIC* ヒット時の最大スループットが *3:2:1* になるということにも意味がある．*INT* 系の命令が連続的に出現す

る命令区間は，*MEM*系の命令が多く含まれている区間に比べると，フロントエンドに高い供給能力を要求すると考えられるからである．すなわち，このような区間はロード/ストアのキャッシュ・ミスによってストールすることが少ないため，*INT*の最大スループットが*1*違うことが，性能に少なからず影響すると考えられる．

以上のように，キャッシュ・ラインのスロット数に偏りを持たせたときの性能評価は*5*章で行う．

各キャッシュ領域のライン数　また$[INT, MEM, FP]$のキャッシュ領域のウェイ方向やセット方向に偏りを持たせ，キャッシュ・ラインの総数の割合を変更することも可能である．前述の，ライン上のスロット数に偏りを持たせる手法でも，キャッシュ容量の割合は変化しているが，それにも限界がある．必要以上にキャッシュ・ラインを大きくしても性能は向上しないためである．

基本的に*DIC*は，*[INT,MEM,FP]*をまとめてキャッシュ・ラインとし，図*3.4*の横一列*(6*命令*)*に*1*つのタグを付けることを想定しているが，これらのキャッシュ領域を独立に扱い，*2*命令ずつにタグを付けることも可能ではある．そうすれば，*[INT,MEM,FP]*のライン数は完全に独立に決定することも可能となる．図*3.4*をこのようにした場合，[–] 領域は他のラインが利用することができ，無駄になる領域は削減される．

ただし，これはタグの容量が*3*倍になることを意味するので効果的ではない．タグ増加と*DIC*利用効率のバランスを考えると，*INT*のキャッシュ領域の半分はキャッシュ・ラインに*INT*命令しか埋まらなかったときに使い，残りの半分は複数種類の命令が埋まったときに用いる，という程度が望ましいと考えられる．

4.1.2　命令出現パターンの偏り

*4.1.1*節では各命令種別毎の，出現総数の偏りについて述べた．図*4.1*のように*INT:MEM:FP*系命令の総実行回数は*6:3:1*であるが，それが常に*6:3:1*のミックスで出現する場合と，*INT*系の命令のみを実行するような比較的長い命令区間が存在する場合とでは，効果的な戦略は異なる．本節では，このような出現パターンの偏りについて簡単に考察する．

プログラム毎に傾向は違うが，基本的には，*INT*系の命令ばかり実行する命令区間がある程度存在し，常に*6:3:1*のミックスで出現するわけではない．プログラムの基本的な流れが，データをロード*(MEM)*し，そのデータになんらかの計算*(INT,FP)*を行い，そのデータをストア*(MEM)*するというものだからである．各ロード/ストアにはアドレス計算*(INT)*が付随すると考えられるので,「なんらかの計算部分」は*INT*が連続し，その前後で*INT*と*MEM*が混ざって出現すると考えられる．

一方で，図*3.4*の*DIC*にとって，最も不運な命令出現パターンは，*INT*が*3*命令連続し，*MEM*が*3*命令連続するというパターンが交互に現れるケースである．

INT		MEM		FP	
0_i	1_i	–	–	–	–
2_i	4_i	3_m	8_m	5_f	X
6_i	7_i	–	–	–	–
9_i	10_i				

図 4.3: キャッシュ利用効率の高いキャッシュ・ライン生成方法

このような際には図 *3.4* の [×] 領域が全ラインに発生することとなる．*4.1.1* 節で述べたように，同図の [–] 領域は他のラインが利用することができるが，[×] 領域は再利用不可能なケースと考えられる．このようなケースが頻発するのであれば，キャッシュ・ライン形成モデルを工夫することでそれに対処できる．

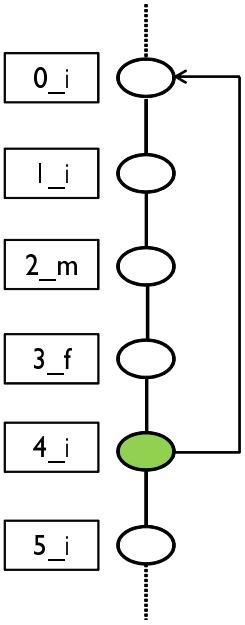
キャッシュ・ライン形成モデルの工夫　以上のケースに対しては，キャッシュ・ラインを形成する際に”詰めて”格納することでキャッシュの利用効率を高めることができる．たとえば**図 4.3** のようにキャッシュ・ラインを形成することができれば，[×] 領域は減少し，利用効率が向上する．図 *3.4* の手法と図 *4.3* の手法で無駄になっている [×] 領域の割合を比較すると，全体の *25%*だったものが *10%*に向上する程度であった．

しかし *3.3.1* 節で触れたように，このような格納手法を実現するためには，プログラム・オーダを再現するためのロジックが複雑になり，リオーダ・バッファの複雑化も招く．*10%*程度の効率向上に対して，このようなロジックの複雑化は割に合わないと考えている．

4.2　実行経路と命令コピーの増加

3 章で述べたように，*DIC* は同一命令を依存経路毎に区別する．このことによって，命令の実行経路とキャッシュ・ラインの形成方法次第では，同一 *PC* の命令が複数のキャッシュ・ラインに存在することがある．通常の命令キャッシュでは単一のエントリを占めていた命令が，*DIC* 上では複数のエントリにコピーを持つことになるので，これはキャッシュの利用効率低下を招く．この，命令コピーの増加は

- 単一 *PC* から複数の μ *Op* にデコードされる場合
- 分岐命令の飛び先や，ループの切れ目によって格納される命令の組み合わせが異なる場合
- 同一の組み合わせであっても，命令の依存関係が違う場合



DIC

INT		MEM		FP	
0_i	1_i	2_m	X	3_f	X
4_i	0_i	–	–	–	–
1_i	4_i	2_m	X	3_f	X
4_i	5_i				

図 4.4: 分岐の切り方による命令コピーの増加

の *3*つの場合に分けられる．上 *2*つのケースは ***Trace Cache(TC)****[2]* の場合でも同様に発生する問題である．ただし，単一 *PC* が複数の *Op* に分かれるケースは，今回用いた *Alpha* の場合は稀なケースなので特別には考えない．本節では *2*つ目の分岐命令の取り扱いでコピーが増えるケースと，*3*つ目の依存関係で区別されるケースについて述べる．

4.2.1 分岐命令の扱い

命令組み合わせの増加は，同一キャッシュ・ラインに異なる基本ブロックの命令を格納することで生じる．例えば**図 4.4** のように，*5*命令から成る小さなイタレーションがキャッシュ・ライン長で割り切れないとき，イタレーション毎にトレースの先頭命令がずれて行き，多数のトレースが生成されることとなる．図 *4.4* の場合は各命令が最低 *2* ラインずつにコピーされている．このループの入り方や出方次第ではより多数のコピーが生成されることとなる．

これに対して，分岐命令の後続の命令を，同一キャッシュ・ラインに格納しないという方針をとれば，イタレーション毎に組み合わせが異なるということはなくなる．例えば図 *4.4* において，4_i と同一のキャッシュ・ラインに 0_i や 5_i を格納しなければ，コピーは増加しない．しかし，これは複数の基本ブロックを同時にフェッチ可能にするという *TC* の利点を損ない，フェッチ幅を低下させることにもなる．

[2, 5] などの，高バンド幅フェッチを目的とした *TC* ではキャッシュ・ライン長は大きく設定されており，フェッチ幅向上のメリットに主眼を置いている．一方で *DIC* は *[3, 7]* と同様に，高バンド幅フェッチを目的とはしておらず，各キャッシュ・ラインは 6 命令程度を想定している．したがって，分岐命令で区切ることによるコピーの減少，それに伴う *DIC* ヒット率の向上が，分岐を区切らずフェッチ幅を増加させる利点を上回ることも考えられる．

また，短いイタレーションによるトレース増加を制限するために，後方分岐であれば区切るという手法も考えられる．図 *4.4* において，4_i と同一のキャッシュ・ラインに 0_i は格納しないが，5_i は格納することで，コピーの増加とスループットの増加のバランスをとるというものである．

これらの手法で *DIC* ヒット率と *CPU* 全体の性能がどのように変化するかについては *5* 章で評価を行う．

4.2.2 依存経路の区別

3.2.2 節で述べたように，同一の命令列であっても，命令の実行経路が異なれば，依存関係が異なるため，これを区別してキャッシュする必要がある．そしてこの区別を行うために，依存経路の先頭アドレスと，そこからの分岐情報をタグとして用いると述べた．ここで依存経路の先頭というのは，同一キャッシュ・ラインの命令が依存している命令の中で，最も古い命令となる．

この依存距離が長くなれば長くなるほど，経路上に含まれる分岐情報は増大する．このことは，同一命令列に対する依存経路数の増加を招き，命令コピーが増加することとなる．したがって，できる限り参照距離を短くすることによって，経路区別による命令コピーの増加を抑えることができる．そうすることで，タグとして用いるパスの容量も削減できる．

依存距離　まず，一般的な命令の依存距離に関して計測を行った．**図 4.5** に命令の依存距離の累積頻度分布を示す．値は *SPECCPU2006* 全ベンチマークの平均で計測している．横軸は参照距離，縦軸はその参照距離を示した命令の割合を累積で表示している．図では最大参照距離を *128* としており，*128* 以上の参照距離を示した場合は，*128* という値でカウントしている．また *4* つの系列は $[Insn]$ が全命令に対する計測で，$[INT, MEM, FP]$ はそれぞれの命令種別毎に参照距離をカウントしたものとなっている．

全命令の平均参照距離は *39* 命令程度となっているが，*6* 割が *12* 命令以下に収まり，*128* を超えるものが *2* 割以上存在するという結果が得られた．特筆すべきは，*[INT,MEM,FP]* の参照距離の差であり，*INT* 系の *7* 割が *10* 命令以内を参照しているのに対し，*MEM* 系で *10* 命令以内なのは *3* 割程度という結果が得られている．

一般的に，*INT* 系の命令が直前に変更した値を使うのに対して，*MEM* 系の命令は古い値を使い回す割合が高く，例えば *MEM* 系のベース・アドレスの値は長い

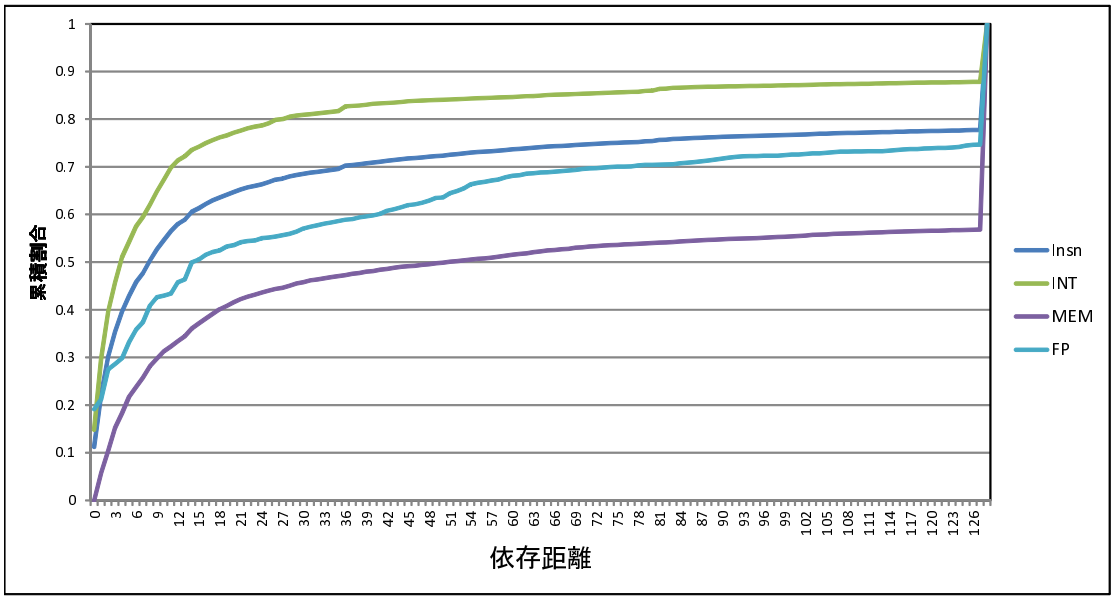


図 4.5: 依存距離の分布

期間で使われることが予想できる．また，単純に *[INT,MEM,FP]* の命令ミックスの割合から *INT* 系の命令よりも，フロントエンド・パイプライン系の命令の方が命令の間隔が空きやすいということも図 *4.5* には表れていると考えられる．

パスの情報量　図 *4.5* の結果から，*100* 命令以上の参照距離をもつ命令も割合としては少なくないが，参照距離にはかなりばらつきがあることがわかった．

キャッシュ・ラインとしての参照距離は，同一キャッシュ・ライン上の最も遠くを参照する命令に引きずられることになるため，それに合わせてパス情報を持たせてしまうと，設定が悲観的すぎるといえる．一方で *3.2.2* 節で述べたように，すべての経路情報を保守的に覚える必要は必ずしもなく，極端に複雑な依存経路を持つ命令は *DIC* に格納しないなどの方法がとられている *[9]*．しかし，ある閾値以上の参照距離を持つ命令を *DIC* に格納しないということにすると，*DIC* に格納されない命令数が増加し過ぎることが想定される．

そこで，パスの閾値は低めに設定し，閾値を超えた少数の命令は投機的に *DIC* に格納するという手法が考えられる．方針としては

- *dualflow* 形式への変換を行わずに格納し，実行時に「少数」の命令は改めてリネーミングを行う
- 変換は行うが，パスで保証はせずに，実行時に依存関係の正誤をチェックする

という *2* 種類を提案する．どちらの場合も，*DIC*・ヒット時でも，ある種のリネーミングを「少数」の命令に対しては行うということとなる．今まで述べてきた，図 *3.1*

のような *DIC* の構成では，ヒット時には通常のリネーミングを行わないことによって，ロジックの簡略化を実現していた．ヒット時もこれを行うようにすると，ロジックの負荷減少量が低下する．ただし，この例外的「少数」の命令を，*1* キャッシュ・ラインにつき *1* 命令程度に抑えられるならば，ロジック規模の減少量は変わらない．本来 *DIC*・ミス時にのみ使用していた *1* 命令分のリネーミング・ロジックを用いればよいからである *(*ただし消費電力は増加する*)*.

このような手法で，極端に参照距離の長い少数の命令によって，キャッシュ・ライン全体の参照距離が長くなってしまう事態を防ぐことが可能である．現実的な実装例としては，

- パスが保証する範囲は分岐命令を *1* つまでとする
- 参照距離が分岐命令を *2* つ以上跨ぐ命令は，キャッシュ・ラインに *1* 命令のみ格納できることとする
- このパスで保証されない *1* 命令は，実行時に改めてリネーミングされる．

という程度のバランスを想定している．パスとしては，経路の先頭アドレスと，経路長，及び経路上での分岐命令の位置のみで一意に特定できる．この実装例に関しては，*5* 章で評価を行う．

4.3 DIC 利用効率のまとめ

本章でここまで述べてきたように，*DIC* は通常の命令キャッシュより利用効率が低下する可能性がある．ここまで述べた内容は，基本的には容量性のミスを引き起こす要因である．その他にも *DIC* と通常の命令キャッシュには相違点が存在し，性能低下の原因に成り得る．

本節では通常の命令キャッシュと比較しながら，*Capacity,Conflict,Compulsory* のミスという観点から *DIC* の利用効率についてまとめる．

4.3.1 Capacity Miss

DIC は以下の要因により，命令キャッシュより多くの容量を必要とする．

- 命令コピーの増加
  - 分岐命令の格納方による
  - 依存経路の違いによる
- 命令が格納されないスロットの増加
  - 命令種別毎の出現総数の偏りによる

– 命令種別の出現パターンの偏りによる

*4.1.1*節で示したように，命令出現数に大きな偏りがあるため，単純に総容量を増やすのではなく，不足している命令領域を増やす必要がある．

また*4.2.2*節で述べたように，命令コピーを増やさないためには，参照経路を最小限に抑える必要があり，そのことがタグ容量の削減にもつながる．

4.3.2 Conflict Miss

*DIC*では同一*PC*から始まるトレースが複数のキャッシュ・ラインを占めることがある．このとき，*DIC*のインデックスをその*PC*情報のみを用いて決めると，それらのコピーが同一セットに集中してしまい，*Conflict Miss*を引き起こすこととなる．

これに対して*DIC*では,「一定命令数前」の*PC*と，そこからの分岐履歴を用いてインデックス決定を行う．例えば,「一定命令数前」を*10*と予め決めておき，トレースの*PC*と，*10*命令前の*PC*と分岐履歴をハッシュしてインデックスを決定するというものである．こうすることで，同一命令列であっても直前の命令列が異なれば違うセットに格納され，*conflict*を避けることができる．以降ではこの「一定命令数」を**最小参照距離**と記述する．

ただし，この最小参照距離まで同一になった場合には*conflict*が発生する．稀ではあるが,

- 小さなループで，ループ回数が多い
- ループ内に，ループ外の命令に依存している命令が存在する

というケースが典型的である．ループ外の命令に依存しているとき，周回毎に依存距離のみが伸びていき，*DIC*では区別してキャッシュされる可能性がある．インデックスに用いられる命令履歴が同一になってから，この参照距離が最大参照距離に到達するまで，これらの区別されるトレースが同一のセットに集中し，*Conflict Miss*が発生する．

この問題も，基本的には参照距離を最小限に縮めることによって緩和される．

4.3.3 Compulsory Miss

*DIC*では通常の命令キャッシュと比べると，*Compulsory Miss*が拡張される形となっている．これは，経路毎に命令列を区別することによる，初期参照回数の増加が一次的な要因である．

同時に，リフィル能力の差によっても*Compulsory Miss*が増加する．例えば，仮に経路も同一の命令列が連続的に出現しても，*DIC*にそのラインが形成される前

であった場合は，*DIC*にヒットしないこととなる．これは命令キャッシュと *DIC*のリフィル能力の差であるともいえる．一般的に *L1* 命令キャッシュのライン・サイズは *64byte(16* 命令*)* などであり，一度キャッシュ・ミスを起こすと *16* 命令リフィルされる．それに対して *DIC* は *1* ライン *6* 命令程度であり，一度 *DIC*・ミスを起こしても，そのラインが形成されるまでには時間がかかるからである．

第5章　評価

本章では *DIC* の性能評価を行う．ここまで論じてきた多数のパラメータに対して，*DIC* の利用効率がどのように変化するか評価する．

5.1　評価環境

評価環境はプロセッサ・シミュレータ「鬼斬弐」*[10]* に，*TC* と *DIC* を実装して評価を行った．

性能評価には *SPECCPU 2006[8]* に含まれる全 *29* 本のベンチマーク・プログラムを用いた．入力データ・セットには *ref* を用い，最初の *1G* 命令をスキップし直後の *100M* 命令の評価を行った．評価したプロセッサの基本的なパラメータは **表 5.1** の通りである．

5.2　評価モデル

以下のモデルを中心に評価を行った．

***Base*(通常の命令キャッシュのみ)*:**
評価のベース・ラインとなるモデルで，*32KB* の命令キャッシュを持つ，*4way* のスーパースカラ・プロセッサ．表 *5.1* 参照．

TC:
通常の *TC* と命令キャッシュを持つモデル．各キャッシュ・ラインに最大 *6* 命令を含み，その内分岐を *1* 命令のみ許すようなモデルとした．容量は *DIC* と同一になるように調整．

DIC:
DIC を持つモデル．基本的な各種パラメータは**表 5.2** に示す．各キャッシュ・ラインの (INT, MEM, FP) の命令スロット数を $(2,2,2)$，$(3,2,1)$ とした *2* 種類を主に考える．パイプラインの短縮は行わず，*DIC* ミス時のリネーム幅，ディスパッチ幅は *1* 命令とし，リネーミング/ディスパッチステージを *1cycle* とした．また *DIC* 全体の容量は一定で，*INT* 領域のライン数を増やし，*MEM* や *FP* を減らしたモデルとも比較を行った．

表 5.1: プロセッサの構成

ISA	Alpha21164A
pipeline stages	Fetch:3,Rename:2,Dispatch:2,Issue:4
fetch width	4 inst.
issue width	Int:2，FP:2，Mem:2
instruction window	Int:32,FP:16, Mem:16
reorder buffer	128 entries
branch predictor	8KB g-share
BTB	2K entries,4way
RAS	8 entries
L1C	32KB,4way,3cycles,64B/line
L2C	4MB,8way,10cycles,64B/line
main memory	200cycles

表 5.2: DIC のパラメータ

pipeline の短縮	しない (ミスを仮定したパイプライン)
pipeline stages	Fetch:4,Rename:1,Dispatch:1,Issue:4
fetch width on DIC Miss	1 inst.
cache line	6inst(I:M:F = [2:2:2] or [3:2:1]).
branch	1 branch/cache line.
最小参照距離	8insnt.
capacity	8way*256set*6insn*4B=48KB.

5.3 予備評価

述べてきたように，*DIC* の実装方式には多数のパラメータが存在する．すべての組み合わせを比較するのは困難なため，ここではそれらのパラメータを決定するための予備評価を行う．

5.3.1 最大参照距離とパス

4.2.2 節で述べたように，*ROB* サイズ *(128* 命令*)* 分の経路を常に一意に特定させるためのパス情報は大きくなりすぎる．そのための命令コピー増加も大きくなる．ここでは，*4.2.2* 節で述べたパスの短縮について評価を行う．

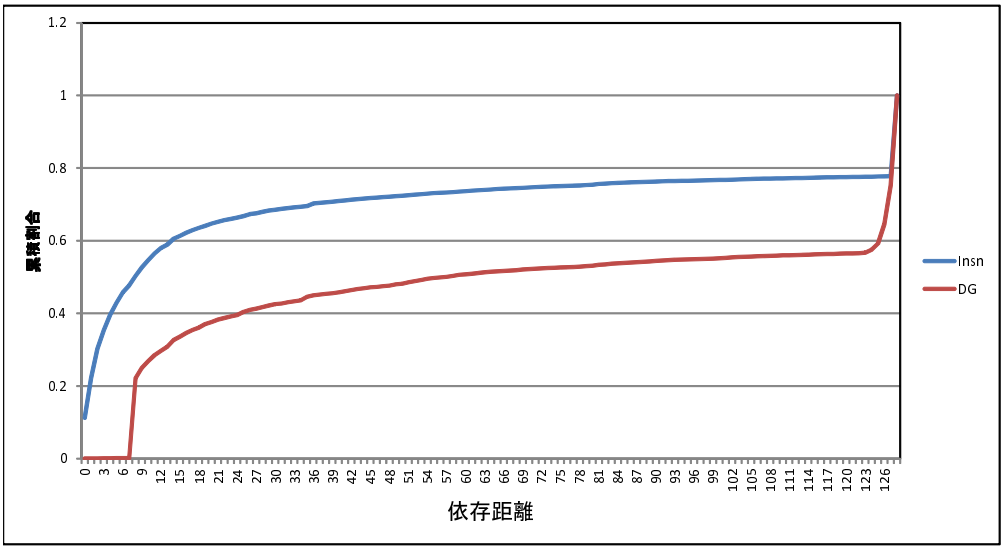


図 5.1: ライン毎の依存距離の分布

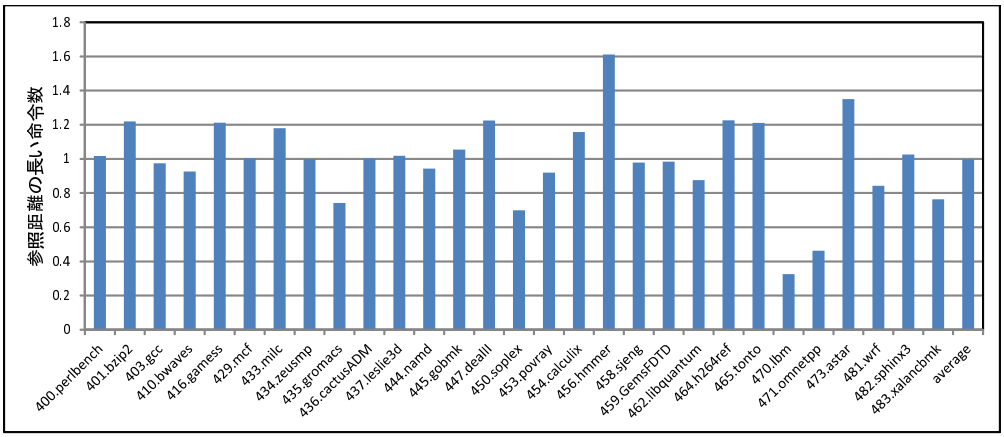


図 5.2: キャッシュ・ライン上で，2 分岐以上過去の命令を参照している命令数

まず**図 5.1** に *DIC* のキャッシュ・ラインを $(3, 2, 1)$ としたモデルにおける，キャッシュ・ラインの参照距離の分布を示す．図 *4.5* と見方は同様で，参照距離を累積分布で表示している．$[Insn]$ が命令毎の分布で，$[DG]$ がライン毎 *(Dispatch Group* 毎*)* の分布になっている．

命令毎の参照距離に対して，キャッシュ・ライン毎になると大きく参照距離が伸びていることがわかる．*4.2.2* 節で見積もった通りに，この格納モデルでも，参照距離の偏りは大きく，少数の命令がラインとしての参照距離を伸ばしてしまうことがわかる．ラインの参照距離の平均は *73* 命令程度と大きいで，この距離の経路情報をパスとして用いるのは効率が悪いと考えられる．

そこで *4.2.2* 節でも述べた，

- パスには分岐命令を *1* つまで格納できることとする

- 参照距離が分岐命令を *2* つ以上跨ぐ命令は，キャッシュ・ラインに *1* 命令のみ格納できることとする
- このパスで保証されない *1* 命令は，実行時に改めてリネーミングされる．

という手法を用いることとする．ライン構成中に分岐を跨ぐ命令が *2* つ存在したら，分岐命令と同様に，そこでキャッシュ・ラインを切ることとなる．

図 5.2 でこの戦略の妥当性を示す．図 *5.2* は，先ほどと同一の *DIC* でラインを作った際に，*2* 分岐以上過去の命令を参照している命令数を表している．各ベンチマーク毎に表示しており，数値はキャッシュ・ラインあたりの平均で示している．この数値が *1* 以下ならば，この格納手法を用いることによって，*DIC* のライン数が増加することはほとんどないと考えられる．

図 *5.2* の通りに，極端に値の大きいベンチマークもなく，平均 *1* 命令程度である．したがってこのくらいの条件が，経路の短縮及びパスの情報量減少と，*DIC* のライン増加のバランスがとられていると考えられる．以降は，このようなモデルを元に評価行う．

5.3.2 命令コピーの増加度合

4.3 章でまとめたように，*DIC* 上には同一の *PC* をもつ命令が複数存在し得る．そのコピーの増加の度合の評価を行う．ベンチマーク毎に，このコピーの増加度合は変わるが，肝心なのは *DIC* の容量がどの程度必要なのかである．すなわち，コピーが *3* 倍 *4* 倍になるようなベンチマークが存在したとしても，元々必要な容量が小さければ問題はない．

ワーキングセットの差　そこで，ワーキングセットの大きさそのものが問題になる．*SPECCPU2006* において，ワーキングセットの大きさにはかなり差がある．そのことは，単純に命令キャッシュのヒット率を比べてかなり差があることからもわかる．今回の実行区間 *(1G* 命令スキップ，*100M* 命令実行*)* では，*gcc* が圧倒的に大きなワーキングセットを持っている．続いて *gobmk* や *xalancbmk* などが大きい．

したがって *DIC* の容量を決定する際には，*gcc* のワーキングセットをカバーできるように設計すれば，基本的には他のベンチマークに対しても十分であると考えられる．

命令コピーの時間変化　**図 5.3** に命令コピーの割合を示す．図 *5.3* は *DIC* の全容量に対して，ユニークな *PC* の割合 *[PC]*，格納されている命令の割合 *[Op]*，命令が格納されていない領域 *[nop]* の割合の時系列変化を示している（*[op]* と *[nop]* を足すとほぼ *1* になる）．横軸はサイクル数を表しており，*1G* 命令スキップした後の，*1M* 命令実行するときの様子を表している．そして *[PC]* に対する *[OP]* の割合が命令コピーの増加分を表す．図 *5.3* を見る限り，命令コピーの増加分は *1.3* 倍程

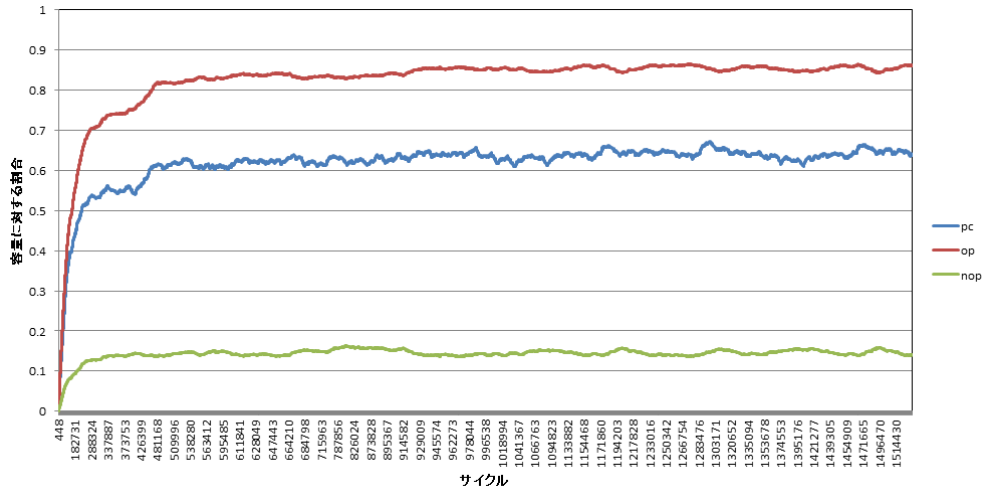


図 5.3: 命令コピーの割合

度ということになる．また，命令が格納されていない領域 *[nop]* は *2* 割以下程度となっており，*4.1.2* 節で述べたように，キャッシュ・ラインの格納方法を複雑にする必要はないと考えられる．

このことから，*DIC* がベースの命令キャッシュと同じサイズのワーキングセットを扱うためには，
$32KB(\text{命令キャッシュの容量}) \times 1.3(copy\text{ の増加}) \times 1.2(\text{格納されない領域}) = 48KB$
程度の容量があれば十分ということになる．

5.4 キャッシュ・ヒット率

図 5.4，**図 5.5** に *DIC* の利用率を *SPECCPU 2006* の *INT* 系と *FP* 系で分けて示す．各モデルの容量は *48KB* とし，キャッシュ・ラインのスロット割合を $(2,2,2)$，$(3,2,1)$ で構成する *2* 種類の比較を行った．さらに，両方のモデルについて，*INT* 命令の容量を大きくし，*MEM* や *FP* の領域を小さくしたモデル $(2,2,2)_I$，$(3,2,1)_I$ に関しても比較を行った．

INT 系に関しては，基本的には *INT* 命令を格納する領域が大きい程，*DIC* ヒット率は高くなっている．最もワーキングセットが大きいと見積もった *gcc* に関しては，十分にヒットしているとは言えないが，特に *INT* 系の容量が不足していることがわかる．

一方で *FP* 系のベンチマークに関しては，*INT* 系と比べると基本的に高いヒット率が得られており，ベンチマーク毎の差は小さいといえる．*FP* 領域が大きいモデルの方が *DIC* ヒット率は高くなっているベンチマークも見られるが，平均的にはスロットの比率が $(3:2:1)$ のものが有利に働いている．$(3:2:1)$ と $(3:2:1_I)$ では *FP* や *MEM* の領域が減っているが，そのことが *DIC* のヒット率に与える悪影

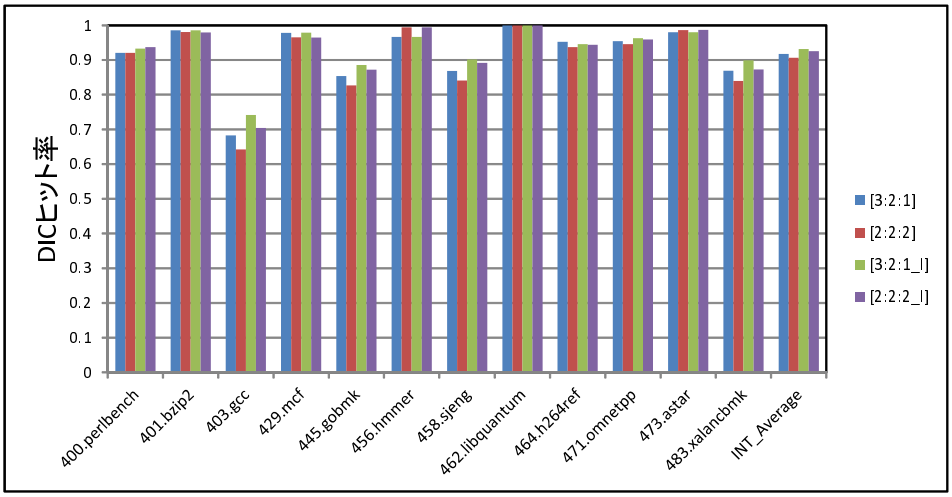


図 5.4: 各モデルのキャッシュ・ヒット率 (INT 系)

響は小さいといえる．

5.5 IPC

図 5.6，**図 5.7** に，*Base* モデルに対する *TC*，*DIC* の相対 *IPC* を，*SPECCPU 2006* の *INT* 系と *FP* 系で分けて示す．*TC*，*DIC* のすべての系列は *48KB* の容量を持たせている．*DIC* はキャッシュ・ラインの命令割合を $(2, 2, 2)$，$(3, 2, 1)$ として二つのモデルで計測した．*INT* 系のベンチマークはベンチマーク毎に大きな傾向の違いはなく，予想の通りキャッシュ・ラインに *INT* 系命令を多く格納できる方が高い性能が得られており，特に *gcc* で差が大きい．また *TC* とも性能差はほとんどない．

FP 系のベンチマークの方が系列間の性能差は小さくなっている．*DIC*321 より *DIC*222 の方が高い性能が得られているベンチマークもいくつか見られるが，平均で見ると *DIC*321 の方が高い性能が得られている．

以上の結果から，*48KB* あれば，*DIC* が性能を発揮するのに十分であることが確認できたといえる．

最後に *DIC* のキャッシュ容量を *24KB,32KB,48KB* と変化させたときの平均相対 *IPC* を**図 5.8** に示す．*24KB* でも相対 *IPC* は *1%* 未満に抑えることが可能になっており，*32KB* ではベースを上回る性能が得られている．

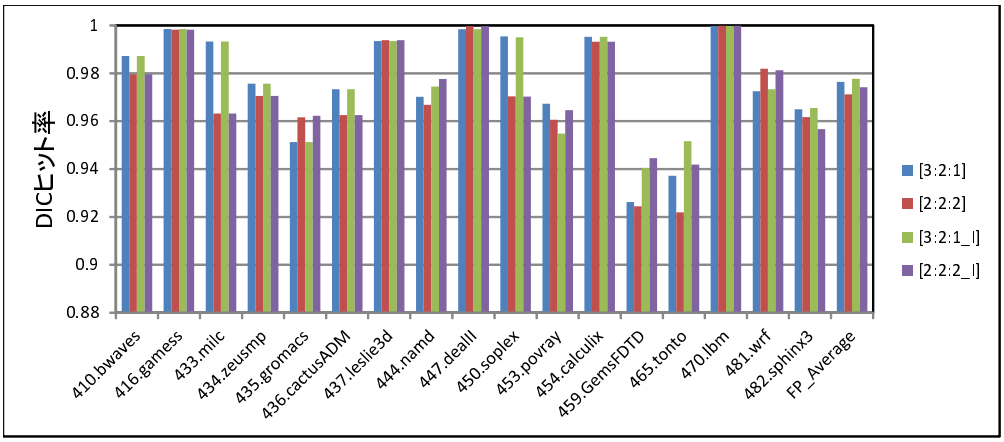


図 5.5: 各モデルのキャッシュ・ヒット率 (FP 系)

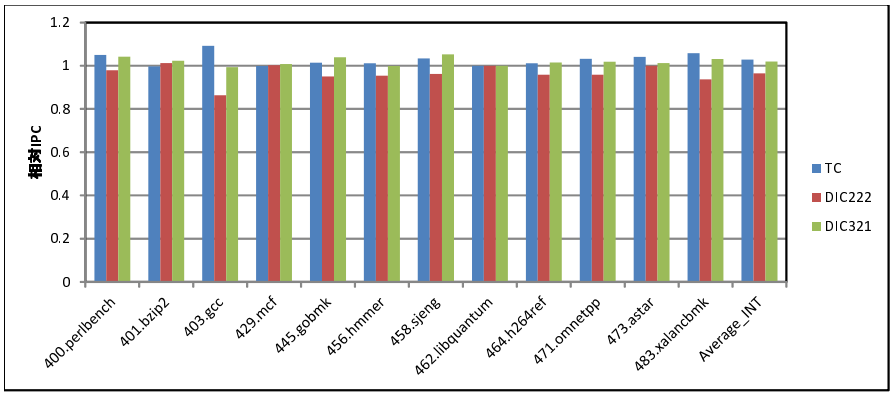


図 5.6: Base に対する相対 IPC(INT 系)

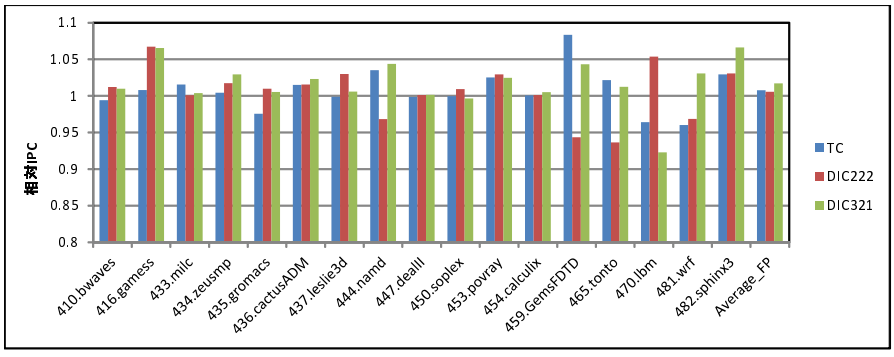


図 5.7: Base に対する相対 IPC(FP 系)

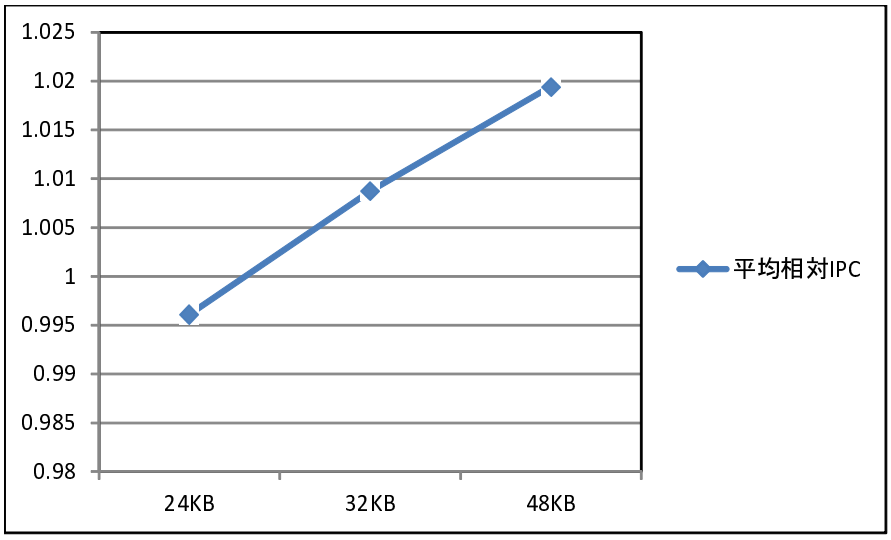


図 5.8: キャッシュ容量を変化させたときの Base に対する相対 IPC

第6章　おわりに

本稿ではフロントエンド・パイプラインの負荷を最小化する手法として，*DIC* を提案し性能の評価を行った．本手法では依存元の命令を指定する形 *(dualflow* 形式*)* に変換された命令列を，対応するサブ・ウィンドウ毎に分けてキャッシュすることで，リネーミング/ディスパッチ情報を再利用する．*DIC* ヒット時に複雑なロジックを要せずにディスパッチでき，*DIC* ミス時のリネーム幅/ディスパッチ幅を最小にすることで，*RMT* とディスパッチ・ネットワークの面積が大幅に削減され，消費電力や熱の問題も緩和することができる．

評価の結果，通常の命令キャッシュを持つ *4* ウェイのスーパースカラ・プロセッサと比べると，同程度の容量 *(32KB)* の *DIC* を追加することで，フロントエンド・パイプラインの負荷を削減しつつ，*1%* 近い性能向上を得られることがわかった．そして，*DIC* を導入することによる回路面積や消費電力の増減に関しては現在評価中であり，今後の課題である．

参考文献

[1] C. Asato, R. Montoye, J. Gmuender, E. W. Simmons, A. Ike, and J. Zasio. A 14-port 3.8ns 116-word 64b read renaming register file. In Proceedings of the Ineternational Solid-State Circuits Conference*, pages 104–105, Feb 1995.*

[2] James E. Smith. Eric Rotenberg, Steve Bennett. Trace cache: a low latency approach to high bandwidth instruction fetching. In Proceedings of the International Symposium on Microarchitecture*, pages 24–35, 1996.*

[3] Mike Upton Darrell Boggs Doug Carmean Alan Kyker Partrice Roussel Glenn Hinton, Dave Sager. The microarchitecture of the pentium 4 processor. In Intel Technology Journal Q1, 2001*, 2001.*

[4] R.E. Kessler. The alpha 21264 microprocessor. IEEE micro*, 19(2):24–36, Mar/Apr 1999.*

[5] John Paul Shen Ryan Rakvic, Bryan Black. Completion time multiple branch prediction for enhancing trace cache performance. In Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.*

[6] Ryota Shioya, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. Register cache system not for latency reduction purpose. In MICRO*, pages 301–312, 2010.*

[7] Baruch Solomon, Avi Mendelson, Doron Orenstein, Yoav Almog, and Ronny Ronen. Micro-operation cache: a power aware frontend for the variable instruction length isa. In ISLPED*, pages 4–9, 2001.*

[8] The Standard Performance Evaluation Corporation. SPEC CPU2006 suite http://www.spec.org/cpu2006/*.*

[9] 一林 宏憲*,* 塩谷 亮太*,* 入江 英嗣*,* 五島 正裕*,* 坂井 修一*. 逆 dualflow* アーキテクチャ*. In* 先進的計算基盤システムシンポジウム SACSIS2008*,* pp.245-254*, 2008.*

[10] 塩谷亮太, 五島正裕, *and* 坂井修一. プロセッサ・シミュレータ「鬼斬弐」の設計と実装. *In* 先進的計算基盤システムシンポジウム SACSIS, *pages 120–121, 2009.*

[11] 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. *2004.*

発表文献

主著論文

1. レジスタ・リネーミングとディスパッチ・ネットワークを最小化するプロセッサ・アーキテクチャ
   伊達三雄, 倉田成己, 塩谷亮太, 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(2012).*

2. レジスタ・リネーミングとディスパッチ・ネットワークを不要とするトレース・キャッシュ・アーキテクチャ
   伊達 三雄, 倉田 成己, 塩谷 亮太, 五島 正裕, 坂井 修一
   情報処理学会研究報告 *2011 ARC 196 (2011).*

3. ディスパッチト・イメージ・キャッシュ
   伊達 三雄, 倉田 成己, 伊藤 悠二, 塩谷 亮太, 五島 正裕, 坂井 修一
   情報処理学会 第 *73*回全国大会, *pp. 1-67-1-68 (2011).*

謝辞

五島正裕准教授には，研究テーマの決定から論文の添削まで，本研究に関する多くの相談に乗って頂き，御指導頂きました．

本研究を進めるにあたり，坂井修一教授には，相談会等を通じ，様々な御指導を頂きました．

塩谷亮太氏，倉田成己氏には，研究内容や論文執筆について様々な助言を頂きました．

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

また，坂井・五島研究室の皆様にも，論文執筆や研究室での生活のサポートなどで大変お世話になりました．心より感謝いたします．