面積効率を指向するプロセッサの研究

塩 谷　亮 太

目 次

図 目 次

表 目 次

第1章

序論

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・プロセッサが実用化され，広く普及している．マルチコア・プロセッサでは，チップ上に集積するコアの数によってその最大性能が決定される．そのため，コアとして用いられるスーパスカラ・プロセッサの面積効率が重要となる．

スーパスカラ・プロセッサの面積効率とは，回路面積あたりの実行スループットであると定義することができる．かつて，シングルコア・プロセッサが中心であった時代では，面積効率の向上は，チップ・コストの削減に直結すると言う点において重要な課題の一つであった．これに対し，今日のマルチコア・プロセッサの時代では，面積効率の向上は，チップ上へ搭載することができるコアの総数を増やすことに繋がる．このため，より高い性能を狙えると言う点において，シングルコア・プロセッサの時代とは異なった意味でコアの面積効率が重要となっている．

コアとして用いられるスーパスカラ・プロセッサを構成する回路は，演算器とその制御部にわけて考えることができる．これらのうち，演算器では一般にウェイ数に比例した回路面積が必要とされる．これに対し，制御部にはウェイ数の 2 乗から 3 乗に比例した面積の回路が必要である．これは，スーパスカラ・プロセッサの制御部が主に多ポートの RAM や CAM によって構成されるためである．一般に RAM や CAM の回路面積はポート数の 2 乗とエントリ数のそれぞれに比例して大きくなる [2, 3]．多くの場合，スーパスカラ・プロセッサの制御部ではウェイ数に比例したポート数，あるいはエントリ数を持った RAM や CAM が必要となるため，結果としてウェイ数の 2 乗から 3 乗に比例した大きさの回路が必要となるのである．

筆者は，このスーパスカラ・プロセッサの制御部のうち，**リネーム・ロジック**と

レジスタ・ファイルに関して，その面積効率を向上させる研究を行った．

リネーム・ロジックに関する研究

スーパスカラ・プロセッサでは，命令間に存在する逆依存や出力依存などの偽のデータ依存を取り除くため，レジスタのリネーミングが行われる．このリネーミングを行うためのリネーム・ロジックは，通常**レジスタ・マップ・テーブル（Register Map Table**：**RMT**）と呼ぶ表によって構成される．RMT は通常 RAM によって構成されるが，この RAM には非常の多数のポートが必要となる．たとえば 3 オペランド形式を持つ命令セット・アーキテクチャでは，必要なポート数は 1 命令あたり 4 本にもなる．RMT を構成する RAM の回路面積はポート数の 2 乗に比例して大きくなるため，結果として RMT はその容量に比べて非常に大きな回路となる．

これに対し，筆者は依存関係をキャッシュすることによって，リネーミングに必要な RMT の規模を大幅に縮小する**リネームド・トレース・キャッシュ**の研究を行った [4]．リネームド・トレース・キャッシュでは，ミス時にのみ命令のリネーミングが行われるため，リネーミングのために必要となる RMT のポート数を大幅に削減することができる．ポート数の削減は，結果として RMT の回路面積を大幅に縮小させる．

レジスタ・ファイルに関する研究

スーパスカラ・プロセッサのレジスタ・ファイルもまた，多ポートの RAM によって構成される．通常，このレジスタ・ファイルには命令のオペランド数に応じたポートが必要となる．また，リネーミング済みの物理レジスタが格納されることから，そのエントリ数も非常に大きなものとなる．これらの結果，近年のプロセッサでは，レジスタ・ファイルの回路面積は L1 データ・キャッシュに匹敵するほどにまで巨大化している [5].

このレジスタ・ファイルの複雑さを緩和する手法として，**レジスタ・キャッシュ**が提案されている [6, 7, 8]．レジスタ・キャッシュは，低レイテンシの小容量なバッファであり，頻繁にアクセスされるメイン・レジスタ・ファイルの一部を保持する．レジスタ・キャッシュでは，ミスを起こした命令のみがメイン・レジスタ・ファイルにアクセスを行うため，そのポート数を大幅に削減することができる．しかし，従来のレジスタ・キャッシュでは，そのミス・ペナルティの影響により性能が大きく低下してしまうことが多かった．

これに対し，筆者はレイテンシを短縮しないレジスタ・キャッシュによって問題を解決する**非レイテンシ指向レジスタ・キャッシュ・システム**の研究を行った [9, 10, 11]. 非レイテンシ指向レジスタ・キャッシュ・システムでは，従来のレジスタ・キャッシュのペナルティを，それよりも発生確率の大幅に低い分岐予測ミス・ペナルティに転化させる．これにより，性能低下をほとんど起こすことなく，メイン・レジスタ・ファイルのポート数を削減する事ができる．

上記で述べた，リネーム・ロジックとレジスタ・ファイル以外の制御部についても，その面積効率を向上させる様々な技術が提案されている [12, 13, 14, 15, 16, 17]. それらの既存研究に加え，筆者が行った上記の研究を統合することにより，スーパスカラ・プロセッサの全域において，面積効率を大幅に向上させる事が可能となった．

本項の内容

次章以降の本項の内容は，以下の通りである．

2 章 Out-of-Order スーパスカラ・プロセッサ

本章では，Out-of-Order スーパスカラ・プロセッサについて述べる．次章以降の理解に必要な Out-of-Order スーパスカラ・プロセッサの動作原理について説明した後，制御部に必要な回路資源について検討を行う．

3 章 リネームド・トレース・キャッシュ

本章では，リネームド・トレース・キャッシュについて述べる．通常のレジスタ・リネーミングについて詳しく説明した後，リネーミングの結果をキャッシュする手法について述べる．

4 章 非レイテンシ指向レジスタ・キャッシュ・システム

本章では，非レイテンシ指向レジスタ・キャッシュ・システムについて述べる．既存のレジスタ・キャッシュ・システムについて説明を行い，それと比較を行いながら，なぜ非レイテンシ指向レジスタ・キャッシュ・システムが有効に働くのかを説明する．

5章 要素技術の統合と評価

これまでの章において説明した要素技術の統合と，それによるプロセッサの評価について述べる．

6章 結論

本項の内容をまとめる．

第2章

Out-of-Orderスーパスカラ・プロセッサ

本章ではOut-of-Orderスーパスカラ・プロセッサについて述べる．Out-of-Orderスーパスカラ・プロセッサを構成する回路は，演算器とその制御部にわけることができる．一般に，Out-of-Orderスーパスカラ・プロセッサでは，この制御部が回路全体に占める割合が支配的であり，演算器が占める割合は非常に低い．本章の目的は，このOut-of-Orderスーパスカラ・プロセッサの制御部について，どのような部分が大きな回路面積を占めているのかを明らかにすることである．

以下では，まず2.1節でOut-of-Orderスーパスカラ・プロセッサの基本的な原理について説明する．Out-of-Orderスーパスカラ・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呼ぶバッファにより緩く結合された構成を取る．2.1節では，これらの部分について説明する．

これらのOut-of-Orderスーパスカラ・プロセッサを実現する各コンポーネントは，主にRAMやCAMによって構成される制御部からなる．2.2節では，それらのRAMやCAMに必要となるエントリ数やポート数の検討を行い，必要となるハードウェアの規模についてまとめる．その後，2.3節でこれらの制御部の複雑さを削減する手法について述べた後，2.4節でまとめる．

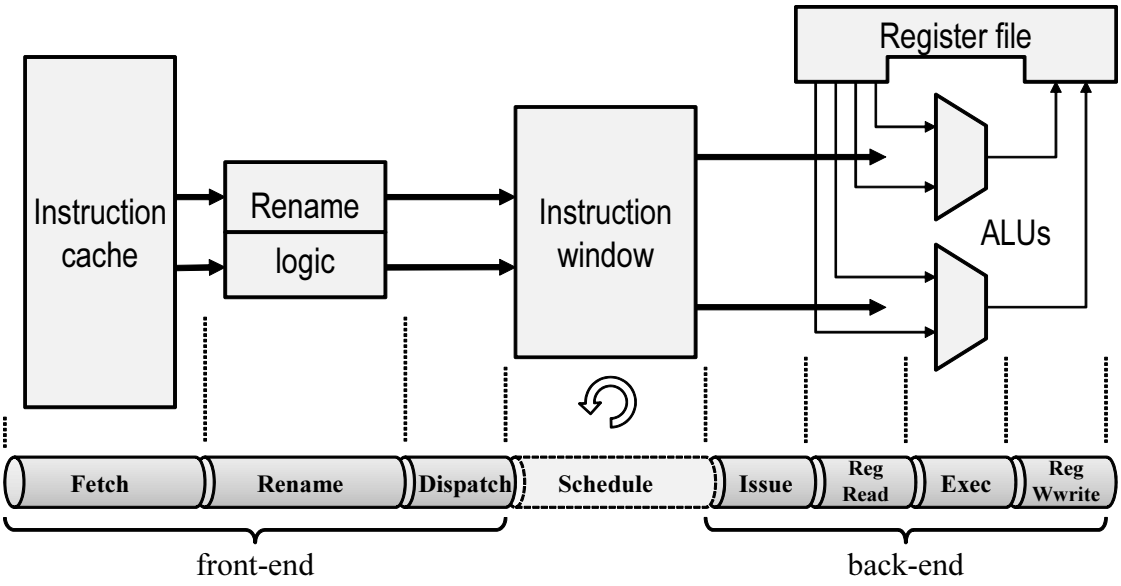


図 2.1: Out-of-Order スーパスカラ・プロセッサの基本構成

2.1　Out-of-Order スーパスカラ・プロセッサの原理

本節では，Out-of-Order スーパスカラ・プロセッサの基本的な原理について説明する．Out-of-Order スーパスカラ・プロセッサを実現する方式は，フロントエンドでレジスタを読み出す方式 [18, 19] と，バックエンドでレジスタを読み出す方式 [20, 21, 22] に大きく分けられる．両者は，異なる方式によって Out-of-Order 実行を実現しているものの，それぞれの制御部に巨大なマルチポートの RAM や CAM を必要とする点では変わらない．本節では，これらのうち，後者のバックエンドでレジスタを読み出す方式に着目して説明を行う．以下では Out-of-Order スーパスカラ・プロセッサの方式とある場合，このバックエンドでレジスタを読み出す方式を指すものとする．

2.1.1　基本的な構成

図 2.1 に，Out-of-Order スーパスカラ・プロセッサの基本的な構成とパイプラインを示す．同図は Out-of-Order スーパスカラ・プロセッサの基本的なコンポーネントと，それを使って処理を行うパイプライン・ステージの対応を表す．

Out-of-Order スーパスカラ・プロセッサは主に，

1. **フロントエンド**
2. **命令ウィンドウ**
3. **バックエンド**

と呼ぶ部分から構成される．フロントエンドは命令の供給や，命令の解析を行う部分である．命令ウィンドウはバッファであり，フロントエンドから供給された命令を蓄える．バックエンドは，命令ウィンドウ中にある命令から並列に実行可能な部分をみつけて，それらの並列実行を行う．フロントエンドとバックエンドは，それぞれが独立したパイプラインとなっており，命令ウィンドウによって緩く結合された構造をとる．以下ではこれらについて詳しく説明する．

2.1.2 フロントエンド

フロントエンドは，命令の供給や解析を行い，命令ウィンドウに書き込むためのパイプラインである．このフロントエンドは，主に以下のコンポーネントによって構成される．

1. **命令キャッシュ**
   命令キャッシュは，メイン・メモリ上にある命令の一部を保持する．フロントエンド・パイプラインへの命令の供給はここから行われる．
2. **分岐予測器**
   分岐予測器は，過去の分岐パターンに従い，分岐先の予測を行う機構である．命令キャッシュへのアクセスは，分岐予測器によって予測されたアドレスを用いて行う．
3. **リネーム・ロジック**
   リネーム・ロジックは，命令の依存解析を行い，逆依存や出力依存などの偽の依存関係を取り除くための機構である．

上記のコンポーネントを用いて行うフロントエンドの処理は，主に以下の 3 つのフェーズに分けられる．これらのフェーズは，通常それぞれ数段ずつにパイプライン化されている．

1. **フェッチ（Fetch）**
2. **リネーム（Rename）**

3. **ディスパッチ（Dispatch）**

命令はこれらのフェーズにおいて In-Order に処理され，命令ウィンドウに書き込まれる．以下では，これらの処理のフェーズについて，順に説明する．

フェッチ

フェッチ・フェーズでは，命令キャッシュからの命令の読み出しを行う．命令の読み出しは，プログラム・カウンタに保存されている命令アドレスを用いて行う．

プログラム中には分岐があるため，これに対しては特別な処理を行う．これは，分岐の結果がわかるまでは，次にフェッチを行う命令アドレスが決められないためである．一般に，Out-of-Order スーパスカラ・プロセッサはパイプライン化されており，命令が命令キャッシュからフェッチされてから実行されるまで，すなわち分岐先が決定するまでには一定のサイクル数がかかる．これを毎回待っていては，命令供給のスループットが著しく低下してしまう．このため，分岐命令のフェッチ時に分岐先の予測を行い，予測結果に基づいて次の命令アドレスを決定する．

この分岐先の予測には様々な方式が提案されている [23, 24]．多くの手法では過去に行われた分岐方向を記録する **Pattern History Table（PHT）**と呼ばれるテーブルと，分岐先ターゲットを記録する **Branch Target Buffer（BTB）**と呼ばれるテーブルを組み合わせることにより，これを実現する．

分岐によってフェッチが中断される問題は Out-of-Order スーパスカラ・プロセッサに特有のものではなく，パイプライン化されたプロセッサ全てで生じる問題である．このため，分岐予測器はスカラ・プロセッサや In-Order スーパスカラ・プロセッサにも搭載される．

リネーム

リネーム・フェーズでは，フェッチした命令を解析し，偽の依存である逆依存や出力依存を取り除く．この偽の依存の解消は，**レジスタ・リネーミング**と呼ぶ処理によって行う．レジスタ・リネーミングでは，命令セット・アーキテクチャで定義された論理レジスタ番号から，プロセッサ内部の物理レジスタ番号への動的な割付けを行う．このレジスタ・リネーミングは，通常**レジスタ・マップ・テーブル（Register Map Table : RMT）**と呼ばれる表を用いて行う．レジスタ・リネーミングについては，後の 3.1 節でより詳しく説明する．

ディスパッチ

ディスパッチ・フェーズでは，リネーム済みの命令を命令ウィンドウに書き込む．

2.1.3 バックエンド

バックエンドは，命令ウィンドウ中にある命令から並列に実行可能な部分をみつけ，それらの並列実行を行うパイプラインである．フロントエンドが命令を In-Order に処理するのに対し，バックエンドでは，命令は依存関係を満たしていれば Out-of-Order に実行される．このバックエンドは，主に以下のコンポーネントによって構成される．

1. **スケジューラ**
   スケジューラは，命令ウィンドウ中からその時点で依存関係を満たしている実行可能な命令を見つける機構である．スケジューラによって選ばれた命令は命令ウィンドウから読み出され，その指示に従って実際の計算を行う．
2. **演算器**
   演算器は，命令で指定された演算を実際に行う機構である．命令を複数並列して実行するために，演算器は通常複数用意される．
3. **レジスタ・ファイル**
   レジスタ・ファイルは，演算結果の値を保持する機構である．並列して複数の演算を行うため，複数の読み書きに対応するための多数のポートを持つ．

上記のコンポーネントを用いて行うバックエンドの処理は，主に以下のフェーズに分けられる．

1. **ウェイクアップ（Wakeup）**
2. **セレクト（Select）**
3. **発行（Issue）**
4. **レジスタ読み出し（Register Read）**
5. **実行（Execution）**
6. **レジスタ書き込み（Register Write）**

以下では，これらの処理のフェーズについて，順に説明する．

2.1.3.1　ウェイクアップ/セレクト

ウェイクアップとセレクトはお互いに緊密な動作を行うため，本項では合わせて説明する．前述したスケジューラは，このウェイクアップとセレクトを行う機構をあわせたものである．図 2.1 では，このウェイクアップとセレクトの動作を合わせて**スケジューリング（Scheduling）**としている．

ウェイクアップ

ウェイクアップ・フェーズでは，命令ウィンドウ中で“眠っている”命令を文字通り“起こす”処理を行う．ここで，命令ウィンドウ中で“眠っている”命令とは，依存関係が満たされていない命令の事である．命令は依存関係が満たされたら，すなわち，依存している全てのオペランドが利用可能になったら，ウェイクアップされる．このような命令の依存関係が満たされた状態を**実行可能（ready）**と呼ぶ．

セレクト

セレクト・フェーズでは，ウェイクアップされて実行可能な命令の中から，後続の演算器に送る命令を選択する．1 サイクルあたりに実行可能な命令数は，演算器やレジスタ・ファイルなどの演算資源の数によって制約される．セレクト・フェーズでは，命令ウィンドウ中で現在実行可能な命令の中から，1 サイクルあたりに実行可能な数の命令を選択する．

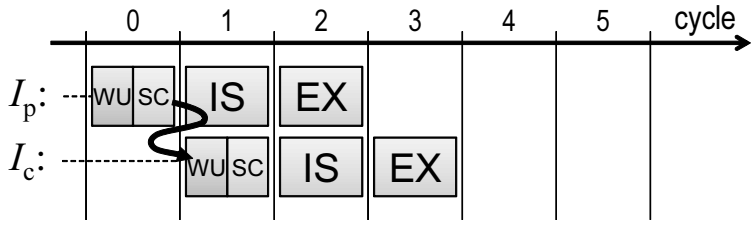
複数の実行可能な命令の中から，どの命令を選択するかと言う基準については，いくつかの方式が提案されている．代表的なものとしては，プログラム・オーダ上で最も古い命令を選択するものがある [21]．また，プログラム内のクリティカル・パスに着目し，クリティカル・パス上にある命令を優先して選択する方式もある [25, 26]．

スケジューリング

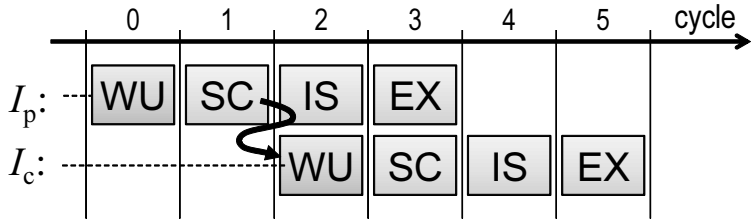
上記のウェイクアップとセレクトの処理はフィードバック・ループを形成している．各サイクルごとに，

1. 命令ウィンドウ中の実行可能な命令の中からセレクトを行い，
2. それによって新たに実行可能になった命令をウェイクアップする

ことを繰り返す．



(a) 1 サイクルでスケジューリングを行った場合



(b) パイプライン化した場合

図 2.2: スケジューリングのパイプライン化

このフィードバック・ループの存在は，スケジューリング処理のパイプライン化を困難にしている．図 2.2 にスケジューリングの際のパイプライン・チャートを示す．図 2.2(a) は，ウェイクアップとセレクトをそれぞれ半サイクルで行った場合の図である．これに対し，図 2.2(b) はウェイクアップとセレクトをそれぞれ 1 サイクルで行うようパイプライン化した場合の図である．同図では I_c は I_p に依存しており，I_p がセレクトされたために，I_c がウェイクアップされている．図内の WU, SC はそれぞれウェイクアップとセレクトである．

同図では，I_p をセレクトした後でなければ，それに依存する I_c をウェイクアップすることはできない．このため，ウェイクアップとセレクトをパイプライン化した場合，図 2.2(b) のように命令を連続して実行することができなくなる．この結果，実行スループットが大幅に低下してしまう．

このため，通常はウェイクアップとセレクトはパイプライン化されず，それぞれ半サイクルで処理を終えられるよう，その規模を制限する．

2.1.3.2 発行以降のフェーズ

ウェイクアップ/セレクトより後にあるフェーズについては，以下のように処理が行われる．

1. **発行**
   セレクト・フェーズにおいて選択された命令は，続く発行フェーズにおいて命令ウィンドウから読み出される．この命令ウィンドウからの命令の読み出しを発行と呼ぶ．
2. **レジスタ読み出し**
   発行によって命令ウィンドウから命令が読み出された後，読み出された命令内のレジスタ番号を用いてレジスタ・ファイルからソース・オペランドを読み出す．なお，発行によって命令が読み出されるまでは，そのソース・オペランドとなるレジスタ番号もわからない．このため，このレジスタ読み出しは発行よりも後にしか行えない事に注意されたい．
3. **実行**
   実行フェーズでは，レジスタ読み出しによって読み出された値を用いて演算を行う．
4. **レジスタ書き込み**
   実行フェーズにおいて得られた演算結果は，レジスタ書き込みフェーズにおいて，レジスタ・ファイルに書き戻される．

2.1.4 ロード/ストア命令のスケジューリング

以上で述べた Out-of-Order スーパスカラ・プロセッサの原理は，レジスタに着目したものである．データの依存関係には，レジスタの他にメモリについても存在する．Out-of-Order スーパスカラ・プロセッサでは，このメモリに関してもスケジューリングを行う．ロード/ストア命令のスケジューリングに関してはいくつかの手法が存在するが，本節では**ロード・ストア・キュー**を用いるものについて説明する．

ロード・ストア・キュー

ロード・ストア・キューは，ロード/ストア命令のみが格納されるバッファである．命令ウィンドウが全ての命令のスケジューリングを行うのに対し，ロード・ストア・キューはロード/ストア命令のみのスケジューリングを行う．これは，

1. メモリ・アドレスについては，レジスタ番号とは異なり，実行時にその実効アドレスが決まることと，
2. レジスタ番号がたかだか数ビットで表せられるのに対し，メモリ・アドレスは数十ビットになるため，

そのスケジューリングが，通常の命令よりも複雑なためである．

ロード・ストア・キューの各エントリは，主にロード/ストア命令の実効アドレスと，ストア命令用の値を格納するフィールドから構成される．ロード命令とストア命令はインオーダにエントリが割り当てられ，それぞれ以下の制約を満たした場合にウェイクアップされる [27].

ロード命令

ロード命令については，以下の条件が満たされている場合に，ウェイクアップされる．

1. ロード命令の実効アドレスが得られている．
2. 先行する全てのストア命令の実効アドレスが得られている．
3. 先行するストア命令の実効アドレスの中で，対象となるロード命令のアドレスと一致するものが無い．

ストア命令

ストア命令については，通常の命令と同様に，実効アドレスとストアを行う値の依存関係が解決された場合にウェイクアップされる．実行時は，その実効アドレスと値をロード・ストア・キューに格納し，リタイアの際に，キャッシュへの書き戻しを行う．

2.2　制御部が占める割合

以上で述べた Out-of-Order スーパスカラ・プロセッサの原理に基づき，本節では，その制御部の回路面積について議論を行う．Out-of-Order スーパスカラ・プロセッサを構成する回路のうち，演算器を除く大部分は RAM や CAM によって構成される制御部からなる．一般に，RAM や CAM の回路面積は，そのポート数の 2 乗とエントリ数に比例する [2, 3]．このため，必要とされるポート数やエントリ数を検討することにより，おおよその回路面積を知ることができる．

各コンポーネントに必要となるポート数は，その処理の幅によって決まる．このため，以下ではまず，Out-of-Order スーパスカラ・プロセッサの処理の幅について説明する．その後，各コンポーネントの説明を行う．

2.2.1　処理の幅

図 2.3 に，各コンポーネントのポート数を決める上で重用となる処理の幅を示す．同図では，以下の 3 つの処理の幅がある．

1. **リネーム幅（Rename width）**
2. **ディスパッチ幅（Dispatch width）**
3. **発行幅（Issue width）**

これらの幅は，それぞれ 1 サイクルあたりにレジスタ・リネーミング，ディスパッチ，発行を行う事ができる命令数を意味する．通常，リネーム幅とディスパッチ幅は同じだけ用意される．これは，リネーミングとディスパッチは同じフロントエンド・パイプライン上にあるためである．どちらかの幅を広くしたとしても，狭い方の幅によってスループットが決定されてしまう．後で述べるように，リネーミング幅を広げると RMT を巨大化させてしまうため，これらの幅は通常リネーム幅によって決まる．

これに対し，発行幅は，リネーム幅やディスパッチ幅よりも広くすることが多い [18, 22]．バックエンド・パイプラインは，命令ウィンドウによってフロントエンド・パイプラインと切り離されているため，フロントエンド・パイプライン上の処理幅の制約を受けない．

表 2.1: 主要なコンポーネントのエントリ数とポート数

ステージ	コンポーネント	エントリ数	ポート数
フェッチ	命令キャッシュ	格納する命令数	1
リネーミング	RMT	論理レジスタ数	4 × リネーム幅
ディスパッチ/発行	命令キュー	インフライト命令の数に比例	ディスパッチ幅 + 発行幅
ウェイクアップ	ウェイクアップ・ロジック		発行幅
レジスタ・アクセス	レジスタ・ファイル		3 × 発行幅
メモリ・アクセス	ロード・ストア・キュー		発行幅

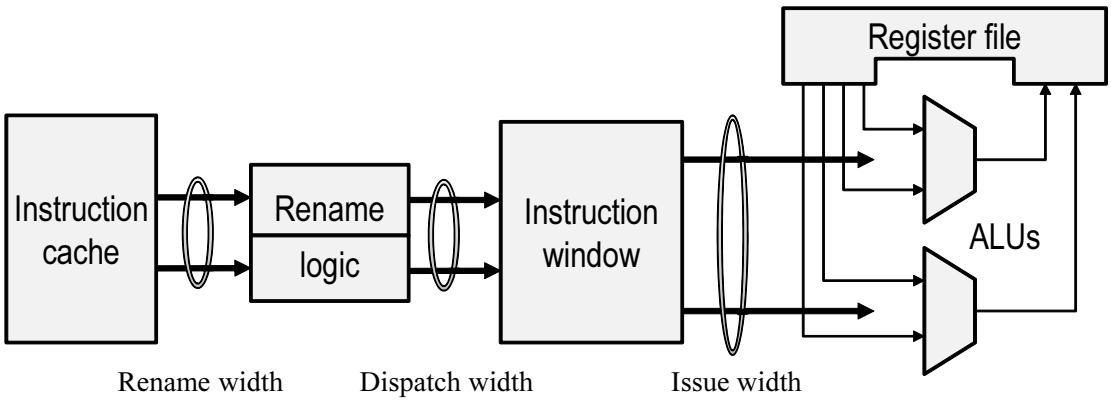


図 2.3: Out-of-Order スーパスカラ・プロセッサにおける処理の幅

2.2.2 主要なコンポーネント

本節では，2.2.1 節の処理幅の議論を基に，主要なコンポーネントに必要なポート数やエントリ数の議論を行う．以下ではこれらのコンポーネントについて，順に説明を行う．

2.2.2.1 命令キャッシュ

命令キャッシュは，主にタグとデータを保持するための RAM からなる．

命令キャッシュのエントリ数は，その読み出しが 2 から 3 サイクル程度で行えるよう制限される．このため，多くのプロセッサでは 8 KB から 64 KB 程度の容量を持つ [19, 21, 18, 20].

2.1.2 節で述べたように，フェッチ・フェーズでは，フェッチ開始アドレスから分岐命令までの連続した命令を一度にフェッチする．このため，命令キャッシュの読み出しポートに関しては，1 ポートであるのが普通である．命令キャッシュに複数

のポートを設けることによって，分岐をまたいでフェッチを行うことも可能であるが，通常は行われない．これは，分岐予測器の複雑化や複数のポートからフェッチした命令の整列に必要な資源の増加が，フェッチのスループットの増加に見合わないためである．

2.2.2.2　RMT

レジスタ・リネーミングを行うための RMT は，RAM か CAM によって構成される．RAM を用いて RMT を構成する場合，そのエントリ数は論理レジスタ数となる．ただし，分岐予測ミスなどからの回復のために，別途なんらかの形で追加のバックアップ用エントリが用意される場合もある [20, 22]．CAM を用いて RMT を構成する場合，そのエントリ数は物理レジスタ数となる [21, 28]．

RMT のポート数については，1 命令あたりにつき，そのオペランド数に比例したポートが必要となる．たとえば 3 オペランド形式の命令セット・アーキテクチャの場合，物理レジスタの解放を行うために追加で必要となる 1 ポートを合わせて，1 命令あたり 4 ポートが必要となる．これは RAM と CAM のどちらで構成する場合も変わらない [29]．この結果，RMT 全体では 発行幅 × 4 のポート数が必要となる．

これらの RMT の構成や動作については，後の 3.1 節でより詳細な説明を行う．

2.2.2.3　ウェイクアップ・ロジック

ウェイクアップ・ロジックは，通常，CAM によって構成される．CAM の内容は，各命令のソース・オペランドを表すタグであり，通常これには物理レジスタ番号が使用される．命令がセレクトされた場合，そのディスティネーション・オペランドをキーとして検索を行う．CAM 内でヒットしたエントリのソース・オペランドを ready とし，全てのソース・オペランドが ready となった場合に，命令をウェイクアップする．この時，1 命令あたりにつきソース・オペランドの数だけ検索を行う必要があるため，命令ウィンドウに格納可能な命令数よりも CAM の容量は大きくなる．

ウェイクアップ・ロジックのエントリ数は，半サイクル内に処理を終えられるよう，通常 16 から 32 程度に制限される．これは，2.1.3.1 節で述べたように，スケジューリングのパイプライン化を行う事が難しいためである．

ウェイクアップ・ロジックのポート数については，発行幅分の検索が同時に行われるため，発行幅の数だけ用意される．

2.2.2.4 セレクト・ロジック

セレクト・ロジックは，他の主要なコンポーネントとは異なり，組み合わせ回路によって構成される．このため，ウェイクアップ・ロジックなどと比べると，その回路は比較的小規模である．セレクト・ロジックは，命令ウィンドウのサイズを WS とすると，おおよそ log_2WS 段程度のダイナミック・ゲート回路によって構成することができる [30]．また，後の 2.3.1 節で述べる，命令ウィンドウの非集中化を行う事により，セレクト・ロジックの入力数や選択数そのものを大きく削減する事も可能である．

2.2.2.5 命令キュー

命令キューは，命令ウィンドウのうち，実際に命令を格納するためのペイロードとなる RAM である．そのエントリ数は，先に述べたウェイクアップ・ロジックのエントリ数と同じである．また，そのポート数については，ディスパッチによる書き込みと発行による読み出しを同時に行うため，ディスパッチ幅と発行幅を足した数が必要となる．

2.2.2.6 レジスタ・ファイル

レジスタ・ファイルは RAM によって構成される．レジスタ・リネーミングを行うため，レジスタ・ファイルは論理レジスタよりも十分に多い数のエントリを持ち，通常は命令ウィンドウ・サイズの倍程度のエントリが用意される．ポート数については，1 命令あたりにつき，そのオペランドの数だけ必要となる．この結果，命令が 3 個のオペランドを持つ場合，レジスタ・ファイルには 発行幅 $\times$ 3 のポート数が必要となる．

2.2.2.7 ロード・ストア・キュー

ロード・ストア・キューの主要なコンポーネントは，実効アドレスとストア時の値を格納するためのバッファ，および実効アドレスの一致検出器からなる．

これらのうち，バッファについては前述した命令キューとほぼ同じものとなる．ただし，ロード・ストア・キューはロード/ストア命令のみを格納すれば良いため，その容量は命令キューよりも小さくなる．

アドレスの一致検出器は，CAM に似た多数の比較器を並べた構造を取る．2.1.4 節で述べたように，ロード命令は，先行する全てのストア命令と実効アドレスの一致比較を行う必要がある．このため，ロード・ストア・キューのサイズを $LSQS$ とすると，1 命令あたり最悪で $LSQS - 1$ 個のアドレス比較を行う必要がある．この結果，ロード・ストア・キュー全体では $LSQS$ の 2 乗に比例した数の比較器が必要となる．

2.2.3 制御部に必要となる回路資源

以上の議論を元に，表 2.1 に，Out-of-Order スーパスカラ・プロセッサの主要なコンポーネントについてまとめる．これまでで説明を行ったように，命令キャッシュを除くコンポーネントのポート数は，リネーム幅，ディスパッチ幅，発行幅のいずれかに比例した数を持つ．

これらのコンポーネントは，チップ内において非常に大きな回路面積を占める．これは，これらのコンポーネントが主に多ポートの RAM や CAM によって構成されるためである．一般に，RAM や CAM の回路面積はポート数の 2 乗に比例する [2, 3] ため，処理幅に応じてポート数を増加させると，その回路面積は急激に増大する．

これらのコンポーネントは，Out-of-Order スーパスカラ・プロセッサの演算器以外の制御部に当たる．これらの制御部のうち，命令キャッシュを除くコンポーネントについては，Out-of-Order スーパスカラ・プロセッサに特有の機構であり，In-Order 実行を行うプロセッサでは通常必要ない．Out-of-Order スーパスカラ・プロセッサが “複雑” であると言われるのは，このように，演算器以外の制御が非常に大きな回路資源を必要とするためである．

図 2.4 に，具体例として Alpha 21464 のフロア・プランを示す[1]．Alpha 21464 は，8 命令同時発行可能な Out-of-Order スーパスカラ・プロセッサである [31]．また，図 2.5 に，このフロア・プランにおける各コンポーネントの回路面積の内訳を示す．

[1] 図 2.4 と図 2.5 は文献 [31] のフロア・プランの図を元に作成した．

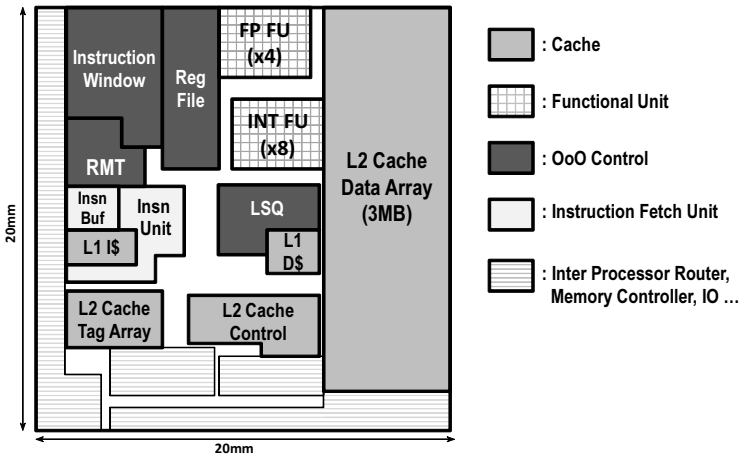


図 2.4: Alpha 21464 のフロア・プラン

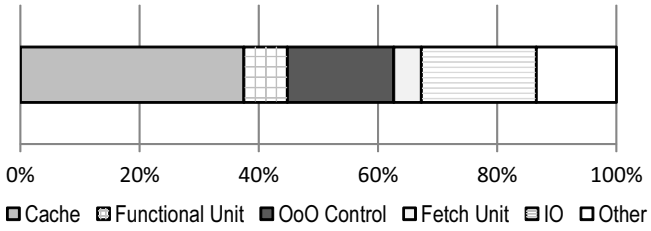


図 2.5: Alpha 21464 の回路面積の内訳

これらの図より，2.2 節で述べた各コンポーネントが実際にチップ内において非常に大きな面積を占めていることがわかる．特に，

- RMT
- 命令ウィンドウ
- レジスタ・ファイル
- ロード・ストア・キュー

の占める面積は非常に大きく，これらのコンポーネントのみでチップ全体の 17.9%もの面積を占める．

2.3　制御部の複雑さを低減する手法

前節までに述べた Out-of-Order スーパスカラ・プロセッサの制御部に対し，その複雑さを低減する手法がいくつか提案されている．本節では，それらの手法として，命令ウィンドウ，ウェイクアップ・ロジック，ロード・ストア・キューの複雑さをそれぞれ低減する手法について述べる．

2.3.1　命令ウィンドウの非集中化

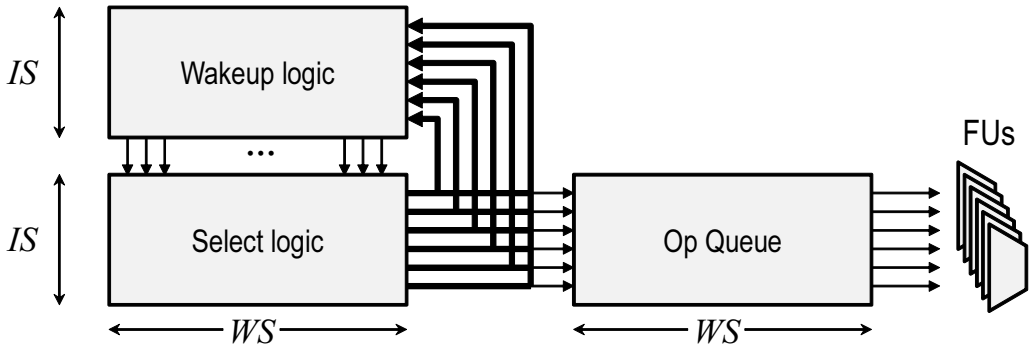
命令ウィンドウは，これまでに述べてきたような集中した単一のロジックとして実装されるのではなく，複数の**サブウィンドウ**に非集中化される場合が多い．命令ウィンドウは，整数，ロード/ストア，浮動小数点と言った命令の系統ごとに，独立したサブウィンドウに分割される [20, 21].

この命令ウィンドウの非集中化により，回路面積を大幅に縮小することができる．ただし，命令ウィンドウを非集中化した場合，ウィンドウの断片化によって性能が低下する可能性がある．命令ウィンドウを系統毎に分割した場合，総容量は足りていても，個々のサブウィンドウの容量が足りないがために，ディスパッチを行えない場合がある．しかし実際には，この断片化の影響は小さく，通常は大きな問題とはならない．これに対し，命令ウィンドウの非集中化は，回路面積の大幅な削減とクリティカル・パスの分離の 2 つの効果がある．以下では，これらの効果について述べる．

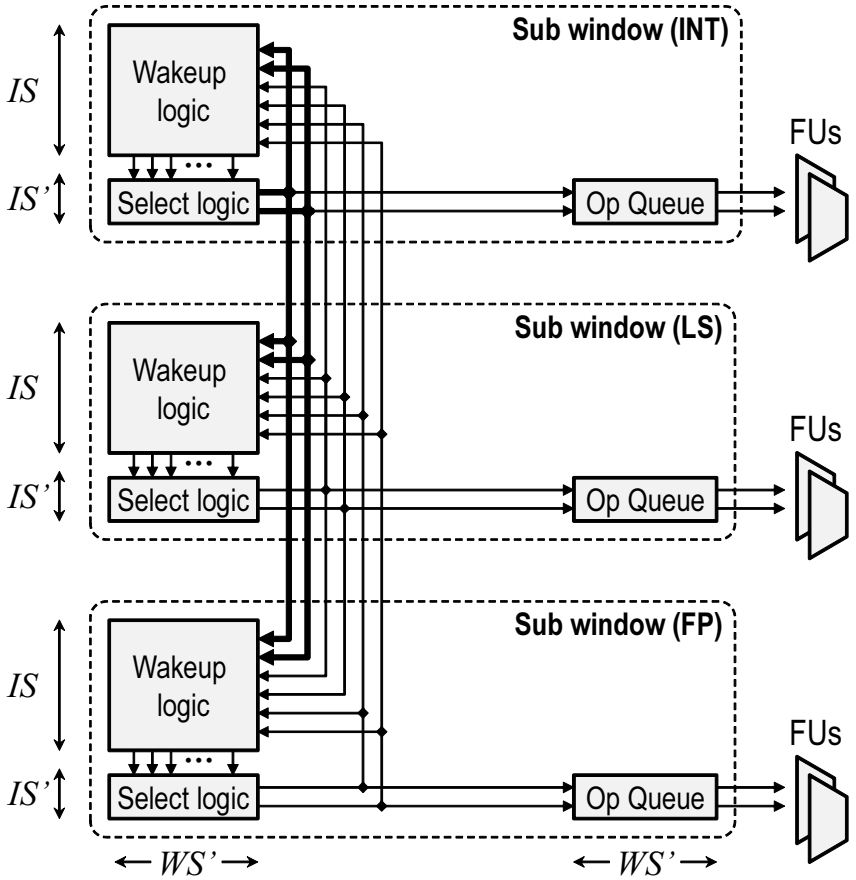
2.3.1.1　回路面積の削減

図 2.6 に命令ウィンドウの非集中化の様子を示す．図 2.6(a) は集中型の命令ウィンドウである．図 2.6(b) は，この集中型のウィンドウを MIPS R10000[20] などの場合と同様にして，整数（INT），ロード/ストア（LS），浮動小数点（FP）の 3 つの系統に応じて非集中化したものである．

同図では，総発行幅を IW，総命令ウィンドウ・サイズを WS としている．また，サブウィンドウに分割した際のそれぞれの発行幅を IS'，命令ウィンドウ・サイズを WS' とする．各サブウィンドウの発行幅と命令ウィンドウ・サイズは等しく，それぞれ $IW' = IW/3 = 2$，$WS' = WS/3$ となっている．



(a) 集中命令ウィンドウ



(b) 非集中命令ウィンドウ

図 2.6: 命令ウィンドウの非集中化

2.2.2 節で述べたように，ウェイクアップ・ロジックやセレクト・ロジック，命令キューは，IS や WS の大きさによってその回路面積が決まる．このため，IS や WS がそれぞれ 1/3 になることによって，それぞれの回路面積は大幅に縮小する．ただし，ウェイクアップ・ロジックに関しては，サブウィンドウ間でウェイクアップを行う必要があるため，そのポート数は IW のままである．このため，個々のウェイクアップ・ロジックは縮小されているものの，回路面積の総量としては変わらない．

2.3.1.2　クリティカル・パスの分離

図 2.6(b) 内の太線で示している，INT から各サブウィンドウへの接続は，2.1.3.1 節で述べた，スケジューリングを 1 サイクルで行う必要のある接続である．これ以外の LS や FP からの接続は，そのスケジューリングをパイプライン化したとしても，2.1.3.1 節で述べたような性能の低下は起きない．これは，INT 以外の LS や FP に属する命令のレイテンシが通常 1 サイクルよりも長いためである．したがって，スケジューリングのクリティカル・パスを考える際は，INT のセレクト・ロジックと，そこから接続されるウェイクアップ・ロジックのことのみを考えればよい．

2.3.2　マトリクス・スケジューラ

2.2.2.3 節で述べたように，ウェイクアップ・ロジックは通常 CAM によって構成される．これに対し，ウェイクアップ・ロジックを RAM と同様の構造を持った依存行列によって構成する，**マトリクス・スケジューラ**が提案されている [12, 13, 14, 15].

図 2.7 にマトリクス・スケジューラの依存行列を示す．依存行列の行と列はそれぞれ命令ウィンドウのエントリに対応しており，列が依存元のプロデューサ命令，行が依存先のコンシューマ命令を表す．依存関係は，依存行列の交点のビットを立てることにより表される．図 2.7 では，I_0 に対して I_1 が，I_1 に対して I_3 が依存しており，それぞれの交点のビットが 1 となっている．

ウェイクアップは，この依存行列を RAM と同様にして読み出すことにより行われる．図 2.7 の場合，行方向が RAM におけるビット・ライン，列方向がワード・ラインに相当する．I_0 のセレクトが行われた場合，図上で上方向の矢印で示され

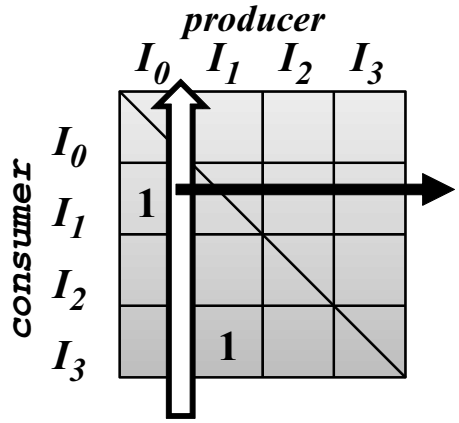


図 2.7: マトリクス・スケジューラ

ている I_0 に対応する列がアサートされる．これにより，交点のビットが読み出され，I_0 に依存する I_1 の行がアサートされる．

マトリクス・スケジューラの依存行列は発行幅に関係無く，1 ポートで構成することができる．複数命令のセレクト時には，それぞれの命令に対応する列を同時にアサートし，読み出しを行えば良い．このため，ウェイクアップ・ロジックを CAM で構成する方式と比べると，マトリクス・スケジューラは大幅に回路面積を削減することができる．

2.3.2.1 プロデューサ・テーブル

マトリクス・スケジューラの問題点は，依存行列中の依存関係がレジスタ番号とは別の方法で表現されているため，追加の変換が必要となる点である．この変換は，レジスタ・リネーミングにおける RMT と同様の**プロデューサ・テーブル**と呼ぶ表によって行う．プロデューサ・テーブルは，論理レジスタごとの，最新の更新を行った命令の位置を内容とするテーブルである．このテーブルを論理レジスタ番号を用いて引く事により，依存元命令の位置を知ることが出来る．

プロデューサ・テーブルは，その内容以外は基本的に RMT と同様の機構であり，リネーム幅に比例した数のポートを必要とする．このため，マトリクス・スケジューラでは，RMT と同程度の回路面積がプロデューサ・テーブルのために追加で必要となる．

2.3.3　ロード再実行

2.2.2.7 節で述べたように，ロード・ストア・キューには多数の比較器が必要となる．これに対し，そのような比較器を必要としない手法として，**ロード再実行**を行う手法が提案されている [32, 16, 17].

これらの手法では，**Store Set 予測器**[33] などを用いて，投機的にロード命令の発行を行う．ロード命令の実行時には，キャッシュから投機的に値を読み出し，それを保存しておく．この時，ロード・ストア・キューで行われるような実効アドレスの一致検出は一切行わない．

投機的に行われたロード命令の正しさの検証は，リタイア時にキャッシュをインオーダにもう一度読み出すことで行われる．この時読み出された値と，保存しておいた投機的に読み出された値の比較を行い，一致している場合に投機が成功していることを検出する．これは，投機的にロードを行った結果が，インオーダにロード/ストアを行った結果と変わらなかったのであれば，実行の結果に矛盾は生じないためである．

これらの手法では，ロード・ストア・キューを無くす事ができるものの，ロード命令の再実行を行うために，キャッシュ・アクセスが増えてしまう問題がある．これに対し，ブルーム・フィルタなどを用いて，必要無い場合にはキャッシュからのロード再実行をフィルタする **NoSQ**（**No Store Queue**）[16] や **Fire-and-Forget**[17] と呼ばれる手法が提案されている．これらの手法では，再実行のフィルタリングに加えて，ストア命令からロード命令への値のフォワーディングをレジスタ・ファイル上で行う手法も合わせて提案している．

2.4　本章のまとめ

本章では，Out-of-Order スーパスカラ・プロセッサの基本的な原理についてまとめ，その制御部に必要な回路資源について検討を行った．2.3 節で述べたように，Out-of-Order スーパスカラ・プロセッサを構成する制御部の中でも，ウェイクアップ・ロジックとロード・ストア・キューに関しては，その回路面積を非常に小さくするか，あるいは完全に省略する手法が提案されている．これに対し，残る RMT やレジスタ・ファイルに関しては，上記の手法ほどには効果的に回路面積を縮小す

る手法は提案されていない．次章以降では，この RMT やレジスタ・ファイルの回路面積を縮小する研究について述べる．

第3章

リネームド・トレース・キャッシュ

Out-of-Order スーパスカラ・プロセッサでは，逆依存や出力依存などの偽の依存を取り除くために，**レジスタ・リネーミング**を行う．このレジスタ・リネーミングのために必要なロジックは，Out-of-Order スーパスカラ・プロセッサの中でも最も高コストなものの 1 つとなっている．

レジスタ・リネーミングでは，論理レジスタ番号から物理レジスタ番号への変換を行う．これは，**レジスタ・マップ・テーブル（Register Map Table : RMT）**と呼ぶテーブルを参照し，更新することにより行われる [20].

RMT は，論理レジスタ番号から物理レジスタ番号への対応を保持する表である．通常，この RMT は多ポートの RAM によって構成される[1]．一般的な 3 オペランド形式の命令セットの場合，RMT にはリネーム幅の 4 倍のポートが必要である．このため，4 命令同時発行可能なプロセッサでは，RMT を構成する RAM のポート数は 16 にもなる．RAM の回路面積はポート数の 2 乗に比例するため [2, 3]，RMT の回路面積は，その容量に対し非常に大きなものとなる．

SMT[34] の普及もまた，RMT を巨大化させる要因の 1 つである．SMT を行うプロセッサでは，スレッドのコンテキストを保持するため，スレッド数に応じた数の RMT が必要となる [35]．4 スレッドの SMT 実行を行う Alpha 21464 プロセッサでは，リネーム・ロジックの回路面積は，L1 データ・キャッシュ（64 KB）よりも大

[1]RMT を多ポートの CAM によって構成する手法 [21, 28] もあるが，RAM の場合と同様の理由により，RMT が高コストなユニットであることには変わらない．

きなものとなっている [31].

巨大な RMT は，遅延や消費電力，熱などの様々な問題を引きおこす.

遅延

RMT が巨大化した場合，その遅延が問題となる．先にも述べたように，RMT は，L1 データ・キャッシュに匹敵する大きさを持つ．したがって，RMT のアクセス・レイテンシもまた，L1 データ・キャッシュに匹敵するほど大きい．このため，L1 データ・キャッシュで行われているのと同様にして，RMT のアクセスもまた，パイプライン化されている．たとえば，Intel Pentium 4 プロセッサ では，RMT へのアクセスには 2 サイクルが割り当てられている [22]．このようにしてパイプラインが深化した場合，予測ミス・ペナルティの増加によって，IPC が低下する．また，命令ウィンドウ・エントリや物理レジスタ・エントリなどの資源の解放が遅れるため，フロントエンドがストールする確率も増える [36, 37].

消費電力や熱

一般に，RAM の消費電力は，その回路面積とアクセスの頻度に比例して大きくなる [2, 3]．通常，ロード/ストア命令は，L1 データ・キャッシュに対してそれぞれ 1 回しかアクセスを行わない．これに対し，ほぼ全ての命令は RMT に対して複数回のアクセスを行う．また，RMT と同様にアクセス頻度の高いレジスタ・ファイルと比較しても，RMT は 2.3 倍ものアクセスを受ける [4]．このため，L1 データ・キャッシュやレジスタ・ファイルと比較して，RMT は面積あたりにおいて，より大きな電力を消費する．この結果，RMT の消費電力は，プロセッサ全体に対して 4%程度を占める [38]．これは，リザベーション・ステーションやグローバル・クロックの消費電力に匹敵するほどの大きさである.

Register Map Cache

このような巨大な RMT の持つ問題の解決を目的として，RMT に小容量の**レジスタ・マップ・キャッシュ（Register Map Cache : RMC）**を追加する手法が提案されている．RMC は，**メイン・レジスタ・マップ・テーブル（Main Register Map Table : MRMT）**の一部を保持するキャッシュである.

Liu らは，RMT のアクセス・レイテンシを短縮するため，RMC を用いる手法を提案している [39]．Liu らの手法では，通常のキャッシュの場合と同様に，RMC が

ヒットした場合にはレイテンシを短縮することができる．キャッシュのミス時には，フロントエンド・パイプラインをストールさせ，その間に MRMT に対してアクセスを行う．Liu らの手法は，後の 4.4.2 節等で詳しく述べるレジスタ・ファイルにキャッシュを追加した場合と同様の理由により，かえって性能が下がってしまう場合がある．

これに対し，三輪らは**ミスを仮定したパイプライン**を用いることにより，RMC による性能の低下を避けながら RMT の面積を削減する手法を提案している [40]．三輪らの手法は，4 章で述べる**非レイテンシ指向レジスタ・キャッシュ・システム**の考え方を RMT に適応したものである[2]．三輪らの手法では，性能をほとんど落とすことなく，大幅に RMT の回路面積を縮小することが可能である．しかし，4 章で述べるレジスタ・ファイルの場合と同様に，三輪らの手法では，MRMT のポート数を削減することはできるが，RMC のポートは削減されない[3]．このため，リネーム幅を増加させた場合には，この RMC の回路面積が再び問題となる．

本章では，この RMT そのものの省略を目的とした，**リネームド・トレース・キャッシュ（Renamed Trace Cache : RTC）**の提案を行う．リネームド・トレース・キャッシュは，真の依存関係（リネーム結果）をキャッシュし，再利用することによって，レジスタ・リネーミングとそれに必要な RMT の省略を行う．リネームを行うのはリネームド・トレース・キャッシュにミスした時のみであり，この時に使う RMT は 1 サイクルあたり 1 命令程度を処理できれば十分である．このため，RMT のポート数を大幅に少なくすることができる．RAM の回路面積はポート数の 2 乗に比例して大きくなるため [2, 3]，このミス時のための RMT の回路面積は非常に小さなものとなる．また，リネームド・トレース・キャッシュからのリネーム結果の読み出しは，リネーム幅とは無関係に 1 ポートで行う事ができる．このためリネームド・トレース・キャッシュでは，回路を巨大化させることなくリネーム幅を広くすることができる．

リネーム済みの結果をキャッシュする手法は，通常の RMT から得られる物理レジスタ番号をキャッシュしただけではうまく機能しない．これは，同じ命令アドレスの命令であっても，

---

[2] 三輪らは文献 [40] 中で，この提案が非レイテンシ指向レジスタ・キャッシュ・システムの考え方を RMT に適用したものであると述べている．

[3] レジスタ・ファイルの場合では，バックエンドのクラスタ化などの手法を組み合わせることにより，回路面積の増加を緩和することができる．しかし，レジスタ・リネーミングはシーケンシャルに行う必要があるため，クラスタ化などの手法を組み合わせることは難しい．

1. 割り当てられる物理レジスタ番号が毎回異なる事と，
2. 経路毎に依存関係が毎回変化する

ためである．リネームド・トレース・キャッシュでは，物理レジスタの参照方式を変更し，経路情報と共にトレース・キャッシュに格納する．これにより，リネームド・トレース・キャッシュでは，リネーム結果を再利用することができる．

本章の構成は，以下の通りである．3.1 節 では背景となるレジスタ・リネーミングについて説明し，通常はリネーミング結果を再利用することが難しいことを説明する．3.2 節と 3.3 節で提案手法について述べたあと，3.4 節で評価結果を示す．

3.1 レジスタ・リネーミング

Out-of-Order スーパスカラ・プロセッサでは，逆依存や出力依存などの偽の依存を取り除くために，**レジスタ・リネーミング**を行う．本節では，背景としてレジスタ・リネーミングについてまとめた後，レジスタ・リネーミングの結果をキャッシュすることが一般に難しいことを説明する．

3.1.1 レジスタ・リネーミングの動作

3.1.1.1 使用するモジュール

レジスタ・リネーミングを実現する方式には，さまざまなものがある [20, 21, 28, 22, 41]．ここでは，それらの中でも最も単純な表を用いる方式について説明する．

表を用いてレジスタ・リネーミングを行う方式では，主に以下のモジュールが用いられる．

1. **物理レジスタ**
   レジスタ・リネーミングを行うプロセッサにおいて，レジスタの値を実際に格納するストレージが物理レジスタ・ファイルである．物理レジスタ・ファイルは，命令毎に動的に割り付けられたレジスタ番号によってアクセスされる．この番号は，命令セット・アーキテクチャによって定義されたレジスタ番号とは異なるものである．これらを区別するため，命令セット・アーキテクチャに

よって定義されたレジスタ番号を**論理レジスタ番号**，物理レジスタ・ファイルにアクセスを行う際に用いるレジスタ番号を**物理レジスタ番号**と呼ぶ．

2. **RMT**
   その時点での，論理レジスタと物理レジスタの対応関係を保持した表がレジスタ・マップ・テーブル（Register Map Table：RMT）である．論理レジスタ番号をインデックスとしてRMTを引く事により，その時点での物理レジスタ番号を得ることができる．

3. **フリーリスト**
   その時点で使用可能な物理レジスタ番号を格納したプールがフリーリストである．新しい物理レジスタが必要な場合はフリーリストから取得を行い，不必要になった場合にはフリーリストに返却を行う．フリーリストは，たとえばFIFOによって構成される．

次節以降では，これらを用いて実際にレジスタ・リネーミングを行う方法について述べる．

3.1.1.2 レジスタ・リネーミングの手順

レジスタ・リネーミングでは，逆依存や出力依存などの，偽の依存の解消をおこなう．以下では，図3.1(a)に示すコードを例として，レジスタ・リネーミングを説明する[4]．同図中のlabelは命令のラベルであり，I_0からI_3は連続する命令を表す．同図左側では，*op*は命令のオプ・コードを，*opD*は命令のディスティネーション・オペランドを，*opL*と*opR*は2つのソース・オペランドをそれぞれ表す．r0からr3は論理レジスタである．

図3.1(a)のコードのデータ・フロー・グラフを図3.2(a)に示す．同図では，ノードI_0からI_3が図3.1(a)上の同じ命令に対応する．ノード間の各エッジのうち，実線はフロー依存を，破線については逆依存と出力依存を表す．エッジの隣にある論理レジスタは，そのレジスタを介して各依存があることを示す．I_2は，I_0に対し，同じレジスタを上書きしていることから，出力依存がある．また，I_2は，I_1のソース・レジスタを上書きしているため，逆依存の関係にある．このため，I_2は，I_0やI_1とは並列に計算を行う事ができない．

[4] これらのコードや図，説明は，一部文献[30]による．

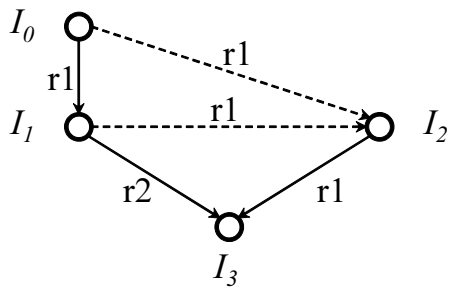
label	*op*	*opD*	*opL*	*opR*
I_0 :	ld	r1	← A	
I_1 :	sll	r2	← r1	<< 1
I_2 :	sll	r1	← r0	<< 2
I_3 :	add	r3	← r1	+ r2

(a) リネーミング前

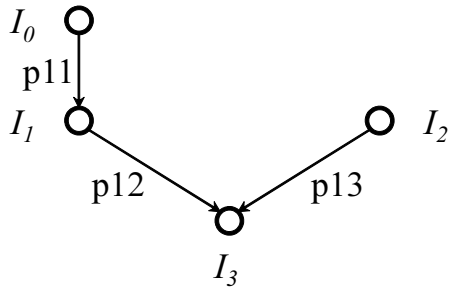
label	*op*	*prD*	*prL*	*prR*	*imm*
I_0 :	ld	p11 ←			A
I_1 :	sll	p12 ←	p11	<<	1
I_2 :	sll	p13 ←	p10	<<	2
I_3 :	add	p14 ←	p13	+ p12	

(b) リネーミング後

図 3.1: レジスタ・リネーミング前と後のコード



(a) リネーミング前のデータ・フロー・グラフ



(b) リネーミング後のデータ・フロー・グラフ

図 3.2: 図 3.1 のデータ・フロー・グラフ

RMT による参照の解決と更新

これらのレジスタに対する上書きを取り除くため，各命令のディスティネーション・オペランドごとに異なる物理レジスタの割り当てを行う．また，各ソース・オペランドでは論理レジスタから物理レジスタへの参照を解決する．

これを行うための具体的な手順は，以下の通りである．

1. **ソース・オペランドの解決**
   命令の各ソース・オペランドの論理レジスタ番号を用いて RMT を引く．これにより，各ソース・オペランドに対応する物理レジスタ番号を得る．
2. **物理レジスタの確保**
   命令のディスティネーション・オペランドに対し，フリーリストから新しい物理レジスタ番号を確保して割り当てる．
3. **RMT の更新**
   割り当てを行った物理レジスタ番号によって，ディスティネーション・オペランドに対応する RMT のエントリを更新する．

なお，**“1. ソース・オペランドの解決”** は，**“3. RMT の更新”** に先立って行う必要があることに注意されたい．これは，命令が，自身のディスティネーションに割り当てられた物理レジスタを，自身のソースとして参照してしまうのを防ぐためである．

以上が，レジスタ・リネーミングの基本的な手順である．なお，同一サイクルにおいて複数の命令のレジスタ・リネーミングを行う場合，これらの操作を命令毎にシーケンシャルに行った場合と同じ結果となるよう，追加の制御を行う必要がある [20, 21].

レジスタ・リネーミングの例

図 3.1(a) のコードに対してレジスタ･リネーミングを行った場合のコードを図 3.1(b) に示す．prD は命令のディスティネーション・オペランドを，prL と prR は 2 つのソース・オペランドを表す．p11 から p14 はリネーミング済みの物理レジスタである．imm は，即値オペランドを表す．その他のラベルについては，図 3.1(a) と同じ意味を表す.

1. まず，I_0 では，ディスティネーション・オペランドに対して，フリーリストか

らp11が割り当てられる．RMTのr1に対応するエントリはp11によって更新される．

2. 次にI_1では，ソース・オペランドのr1を用いてRMTを引く．この時，I_0によってr1に対応するエントリはp11に更新されているため，I_1のソース・オペランドはp11となる．
3. その後，I_1のディスティネーション・オペランドに対して，フリーリストからp12が割り当てられる．RMTのr2に対応するエントリはp12によって更新される．
4. 以下，I_2とI_3についても同様の操作を行った結果が図3.1(b)である．

図3.1(b)のコードのデータ・フロー・グラフを図3.2(b)に示す．エッジやラベルなどの意味は，図3.2(a)の場合と同様である．図3.2(b)では，図3.2(a)にあったI_2とI_0，I_2とI_1間の偽の依存がなくっている．

3.1.1.3 物理レジスタの解放

命令への物理レジスタの割り当ては，比較的容易な操作により実現できる．これは，フリーリストから物理レジスタ番号を取り出すだけでよい．

これに対し，不要になった物理レジスタの解放を行う事は難しい．これは，現在物理レジスタを参照している命令が存在する期間だけではなく，将来にわたって使用される可能性がある間は解放を行う事ができないからである．

ある論理レジスタを上書きする命令が**完了**した場合，その上書きされた論理レジスタ（とそれに対応する物理レジスタ）が以降で参照されることはない．たとえば図3.1(a)の場合，I_0のディスティネーション・オペランドであるr1は，I_2によって上書きされている．この場合，I_2の完了後にはI_0のディスティネーション・オペランドを参照する命令は現れない．このため，図3.1(b)で，I_0に割り当てられているp11の解放をI_2の完了時に行う事ができる．MIPS R10000では，この議論に基づいた手法を実際に実装している[20].

これを実現するため，命令はレジスタ・リネーミング時に，自身が上書きを行う論理レジスタに割り当てられている物理レジスタを取得する必要がある[5]．これは

[5]この時取得した物理レジスタ番号は，アクティブ・リストと呼ばれるリストに登録され，命令の完了時や投機予測ミス時に参照される[20].

単純にRMTをディスティネーション・オペランドによって参照することにより実現できる．しかしこのため，RMTにはさらなる追加のポートが必要となる．3オペランドの命令セット・アーキテクチャにおいて，RMTに1命令あたり4ポートが必要とされるのは，このためである．

3.1.2 リネーミング結果がキャッシュ不能な理由

3節で述べたように，RMTは非常に高コストなコンポーネントである．RMTには通常，1命令あたり4ポートが必要である．RMTを構成するRAMの面積はポート数の2乗に比例するため[2, 3]，RMTの回路面積は，その容量に対し非常に大きなものとなる．

本稿で提案するリネームド・トレース・キャッシュは，レジスタ・リネーミングの結果をキャッシュすることによって，このRMTの規模の縮小を図るものである．このレジスタ・リネーミングの結果のキャッシュは，そのままでは行えない．これは主に

1. 命令に割り当てられる物理レジスタは毎回異なることと，
2. 実行の経路毎に命令の依存関係が毎回変化すること

による．以下では，これらの点について説明を行う．

物理レジスタ番号の割り当ての非再現性

3.1.1節で述べたように，物理レジスタ番号はフリーリストによって管理される．物理レジスタ番号は，命令のリネーム時や完了時にその確保や解放を行う．このため，同一のアドレスにある命令であっても，そこに割り当てられる物理レジスタ番号は毎回異なる．

経路毎の依存関係の変化

命令の依存関係は，それまでの実行経路に依存して変化する．このため，ソース・オペランドとして参照する物理レジスタは，経路毎に異なる．

図3.3に示すコードを例として，このことの説明を行う．各ラベルの意味については，基本的に3.1.1節内の図3.1(a)のものと同じである．I_1は条件分岐命令となっており，分岐成立時にはI_3へ制御を移す．このコードに対し，レジスタ・リ

label	*op*	*opD*	*opL*	*opR*
I_0 :	ld	r1 ←	A	
I_1 :	beq		r1	I_3
I_2 :	add	r1 ←	r1 +	1
I_3 :	sll	r2 ←	r1 <<	1

図 3.3: 分岐を含んだコード

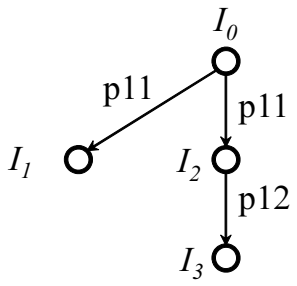
label	*op*	*prD*	*prL*	*prR*	*imm*
I_0 :	ld	p11 ←			A
I_1 :	beq		p11		I_3
I_2 :	add	p12 ←	p11 +		1
I_3 :	add	p13 ←	p12 <<		1

(a) Untaken

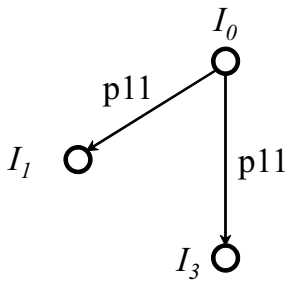
label	*op*	*prD*	*prL*	*prR*	*imm*
I_0 :	ld	p11 ←			A
I_1 :	beq		p11		I_3
I_3 :	add	p12 ←	p11 <<		1

(b) Ttaken

図 3.4: 分岐を含む場合の実行フロー（レジスタ・リネーミング後）



(a) Untaken



(b) Taken

図 3.5: 図 3.4 のデータ・フロー・グラフ

ネーミングを行った時の実行フローを図 3.4 に示す．各ラベルの意味については，図 3.1(b) のものと同じである．図 3.4(a) は I_1 の分岐が不成立の場合，図 3.4(b) は分岐が成立した場合である．また，これらの実行フローに対するデータ・フロー・グラフを図 3.5 に示す．図 3.5(a) と図 3.5(b) は，それぞれ図 3.4(a) と図 3.4(b) に対応する．

これらの図では，分岐の成立，不成立によって同じ命令アドレスにある I_3 の依存関係が異なっている．I_3 は，図 3.5(a) では，p12 を介して I_2 に依存しているのに対し，図 3.5(b) では p11 を介して I_0 に依存している．

3.2 リネームド・トレース・キャッシュ

本節では，**リネームド・トレース・キャッシュ**の提案を行う．リネームド・トレース・キャッシュでは，レジスタ・リネーミング済みの命令をトレース・キャッシュを用いてキャッシュする．レジスタ・リネーミングはキャッシュ・ミス時のトレース生成時にのみ行われるため，そのためのリネーム・ロジックを大幅に縮小することができる．

3.1 節で述べたように，一般に，レジスタ・リネーミングの結果をキャッシュすることは困難である．これは，

1. 命令に割り当てられる物理レジスタは毎回変化することと，
2. 実行の経路毎に命令の依存関係が毎回変化すること

による．これに対し，リネームド・トレース・キャッシュは，

1. ソース・オペランドを命令間の変位で指定し，
2. 経路毎にトレース・キャッシュにキャッシュすること

によって，レジスタ・リネーミング結果のキャッシュを可能にしている．

以下では，3.2.1 節で，変換するトレースのモデルについて述べた後，3.2.2 節で，それを経路毎にキャッシュする方法について説明する．

3.2.1 リネームド Op

リネームド・トレース・キャッシュでは，命令セット・アーキテクチャによって定義された命令をレジスタ・リネーミング済みの命令に変換する．以下では，こ

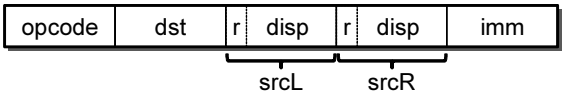


図 3.6: リネームド Op のフォーマット

のレジスタ・リネーミング済みの命令を**リネームド Op** と呼ぶことにする．本節では，このリネームド Op のフォーマットと，レジスタのモデルについて述べた後，具体的な実行の様子を説明する．

3.2.1.1 リネームド Op のフォーマット

リネームド Op では，基本的にソース・オペランドを，n 命令前の実行結果と言う形で命令間の**変位**で指定する．また，依存元命令がリタイアしている事が保証される場合には，命令セット・アーキテクチャの定める論理レジスタ番号でソース・オペランドを指定する．

図 3.6 にリネームド Op のフォーマットの例を示す．各フィールドの意味は以下の通りである．

opcode

命令の opcode が格納される．

dst

命令のディスティネーション・オペランド．論理レジスタ番号によって指定される．

srcL/srcR

命令のソース・オペランドが格納され，disp と r のサブフィールドを持つ．r が 0 の場合，disp は依存元命令への変位を示す．r が 1 の場合，disp は依存する論理レジスタ番号を示す．

imm

命令の即値フィールドが格納される．

label	*op*	*opD*	*opL*	*opR*
I_0 :	mov	r1	← A	
I_1 :	sll	r2	← r1	<< 1
I_2 :	sll	r3	← r1	<< 2
I_3 :	add	r4	← r2	+ r3
I_4 :	add	r3	← r1	+ r4

(a) リネーミング前

label	*op*	*dst*	*srcL*	*srcR*	*imm*
I_0 :	mov	r1	← A		
I_1 :	sll	r2	← [-1]	<<	1
I_2 :	sll	r3	← [-1]	<<	2
I_3 :	add	r4	← [-2]	+ [-1]	
I_4 :	add	r3	← r1	+ [-1]	

(b) リネーミング後

図 3.7: シフトと加算により，7 倍を計算するコード

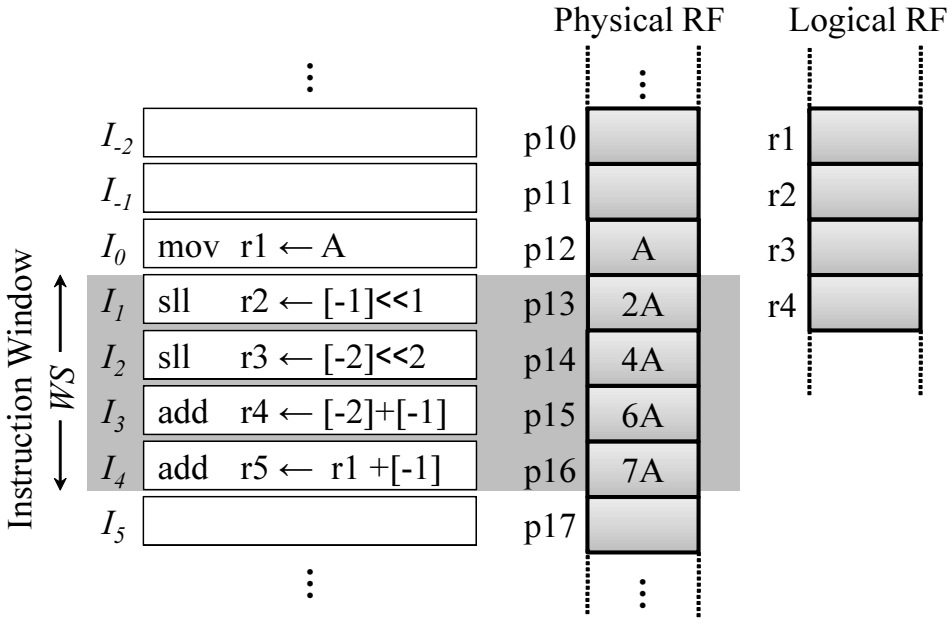


図 3.8: リネームド・トレース・キャッシュのレジスタ・モデルと実行例

3.2.1.2 レジスタのモデル

3.1.1 節で述べたように，通常のレジスタ・リネーミングでは，フリーリストを用いて未使用の物理レジスタが命令に割り付けられる．これに対し，リネームド・トレース・キャッシュでは物理レジスタはリング状の構成を取り，命令に対してシーケンシャルに割り当てられる．

図 3.8 に，リネームド・トレース・キャッシュのレジスタ・モデルを示す．同図で I_{-2} から I_5 は連続する命令を，p10 から p17 は物理レジスタ番号を，r1 から r4 は論理レジスタ番号をそれぞれ表す．また，同図の I_0 から I_4 の命令は，図 3.7 に示すコードと同一のものである．このコードはシフトと加算を組み合わせることにより，定数 A の 7 倍を計算するコードである．

リネームド・トレース・キャッシュのレジスタ・モデルでは，物理レジスタ・ファイルと論理レジスタ・ファイルがある．

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．この物理レジスタ・ファイルの各エントリは，プログラム・オーダ上での実行命令列の各ディスティネーション・オペランドと 1 対 1 に対応しており，命令に対してシーケンシャルに割り当てられる．図 3.8 の場合，I_{-2} から I_5 までの命令に対し，それぞれの右側にある物理レジスタ・ファイルのエントリが割り当てられる．たとえば，I_0 のディスティネーション・オペランドには p12 が，I_1 のディスティネーション・オペランドには p13 が対応する．前述したように，各命令のソース・オペランドは命令間の変位の形で指定されており，自身のインデックスにこの変位を加算することによって，目的の物理レジスタのインデックスを求める．

この物理レジスタ・ファイルのモデルでは Out-of-Order 実行が想定されており，**命令ウィンドウ**が導入されている．図 3.8 では，影となっている部分がサイズ WS の命令ウィンドウとなっている．物理レジスタ・ファイルはウィンドウ・サイズ WS の 2 倍の容量を持つ．

論理レジスタ・ファイルは，この命令ウィンドウからリタイアした命令の実行結果を保持する．命令は，リタイア時にその実行結果を論理レジスタ・ファイルに対してインオーダに書き込む．この時書き込みを行うレジスタ番号は，論理レジスタ番号をそのまま用いる．ソース・オペランドを供給する命令が WS 命令以上離れており，自身が命令ウィンドウに入る際に依存元命令がリタイアしている事が保証される場合には，論理レジスタ・ファイルから値を読み出す．

3.2.1.3 実行例

引き続き，図 3.8 を用いてリネームド・トレース・キャッシュの具体的な実行例を示す．同図では I_0 の実行が完了し，I_1 から I_4 までの実行が終了した状態となっている．同図上の I_0 から I_4 までのコードは，以下のように実行される．

1. 最初，命令ウィンドウ内には I_0 から I_3 がディスパッチされている．I_0 は即値を生成する命令であり，p11 に定数 A を書き込む．
2. 図 3.7 では，I_1 は，1 命令前の I_0 のディスティネーションである r1 を読んでいる．このため，I_1 のリネームド Op ではそれらの変位である -1 がソース・オペランドとなる．I_1 は，自身に割り当てられた物理レジスタ番号 p13 に変位 -1 を加算し，依存元の p12 を参照する．I_1 は左シフトの結果である 2A を p13 に書き込む．
3. I_2 と I_3 についても I_1 と同様に実行を行う．それぞれの物理レジスタ番号に依存元への変位を加算することによってソース・オペランドを決定し，演算結果を自身の物理レジスタに書き込む．
4. I_0 の実行が終了したため，リタイヤさせる．命令ウィンドウが図上で下方向に 1 命令分スライドし，図 3.8 の状態になる．この時，同時に以下の I_4 のディスパッチが行われる．
5. I_4 のディスパッチを行う．I_4 のソース・オペランドのうち，r4 についてはこれまでと同様にして命令間の変位を用いてレジスタ・アクセスを行う．もう 1 つのソース・オペランドである r1 に関しては，それを生成する I_0 が既にリタイアしているため，論理レジスタから値を取得する．図 3.8 からも明らかなように，ある命令の依存元命令がリタイアしているかどうかは，依存元命令がその命令から WS 命令以上離れているかどうかによって判定可能である．I_4 は I_0 からちょうど WS 命令離れており，I_0 の実行結果を論理レジスタを介して受け取る．
6. I_4 はディスパッチ後，実行を行い，結果を p16 に書き込む．

3.2.1.4 物理レジスタの再利用タイミング

物理レジスタはリング状になっており，使い終わった物理レジスタはシーケンシャルに再利用される．この再利用は，割り当てを行った命令の完了直後に行ってはいけない．たとえば図 3.8 の場合，I_0 の完了直後に p12 を再利用してはならない．これは，図からも明らかなように，p12 を参照する命令はまだ命令ウィンドウ内に存在するためである．

物理レジスタの再利用のタイミングは，ウィンドウ・サイズ WS によって決定される．先にも述べたように，依存元命令が WS 命令以上離れていた場合には，命令は論理レジスタを介して値を受け取る．これは逆に言うと，命令が完了してから WS 命令間は，割り当てられた物理レススタには参照の可能性があり，再利用してはいけない事を意味する．このため，物理レジスタ・ファイルには命令ウィンドウそのものの大きさに加え，リタイヤ後 WS 命令分のエントリが必要となる．物理レジスタの容量が WS の 2 倍必要なのは，このためである．

3.2.1.5 リネームド Op への変換

リネームド Op への変換は，通常のレジスタ・リネーミングと同様にして行う事ができる．通常の RMT が論理レジスタ番号と物理レジスタ番号の対応を保持しているのに対し，同様の表によって論理レジスタ番号と命令の位置を保持すればよい．以下，この表を **Dependency Map Table**（**DMT**）と呼ぶ．

以下にこれを実現するナイーブなアルゴリズムを示す．

1. フェッチした各命令に対し，プログラム・オーダ順の連続した通し番号を振る．
2. 命令のディスティネーション・オペランド（論理レジスタ番号）に対応する DMT のエントリを，この通し番号で更新する．
3. ソース・オペランドの論理レジスタ番号で DMT を引くことにより，最後にその論理レジスタを更新した命令の通し番号が得られる．これを自身の通し番号から引く事により，命令間の距離を得る．
4. 命令間距離が WS 以上であった場合，論理レジスタへアクセスを行う．

上記で用いる DMT は RMT と同様の物理構成を持つ．このため，変換幅の 2 乗に比例してその回路規模が増大する．

提案手法では，変換済みのリネームドOpをキャッシュし，上記の変換はキャッシュ・ミス時にのみ行う．したがって，変換幅を縮小しても性能には大きな影響を与えない．変換幅の縮小は，変換に必要なDMTの回路規模を大幅に縮小させる．

なお，上記のアルゴリズムでは，通し番号の再利用が考えられておらず，そのままでは実現に際していくつかの問題がある．DMTの現実的な構成方法については，後の3.3節で詳しく述べる．

3.2.1.6 リネームドOpの効果

以上のように，リネームドOpでは，物理レジスタへの参照は命令間の変位によって示される．この変位は，依存関係にある命令間の命令列が同じであれば常に一定である．このため，1度行った変換結果の変位を再利用することが可能であり，物理レジスタの絶対位置とは関係無く常に正しいソース・オペランドを指すことができる．

実際のプログラムでは分岐命令があるため，命令の依存関係はその実行経路によって変化する．次節では，リネームドOpを実行の経路毎にトレース・キャッシュに格納する方法について説明する．

3.2.2 トレースのキャッシュ

リネームドOpでは，命令間の変位を用いてソース・オペランドを指定する．この変位は，実行の経路が同じであれば常に一定である．しかし，依存関係にある命令間の経路が異なる場合には，その変位もまた変化してしまう．

これに対し，リネームド・トレース・キャッシュではリネームドOpの**トレース**を経路毎にキャッシュすることにより，異なる経路でも正しいリネームドOpを取得できるようにする．ここでトレースとは，実行時の動的な順に並べられた命令列のことを指す．このキャッシュには**トレース・キャッシュ**[42]と同様の仕組みを用いる．

以下では，実行経路毎にリネームドOpの変位が変化することを説明した後，リネームドOpのトレースをキャッシュする方法について述べる．

label	*op*	*opD*	*opL*	*opR*
I_0 :	ld	r1	← A	
I_1 :	beq		r1	I_3
I_2 :	add	r1	← r1 +	1
I_3 :	sll	r2	← r1 <<	1

図 3.9: 分岐を含んだコード

label	*op*	*dst*	*srcL*	*srcR*	*imm*
I_0 :	ld	r1 ←			A
I_1 :	beq		[-1]		I_3
I_3 :	add	r2	← [-2] <<		1

(a) Taken

label	*op*	*dst*	*srcL*	*srcR*	*imm*
I_0 :	ld	r1 ←			A
I_1 :	beq		[-1]		I_3
I_2 :	add	r1	← [-2] +		1
I_3 :	add	r2	← [-1] <<		1

(b) Untaken

図 3.10: 図 3.9 の実行フロー

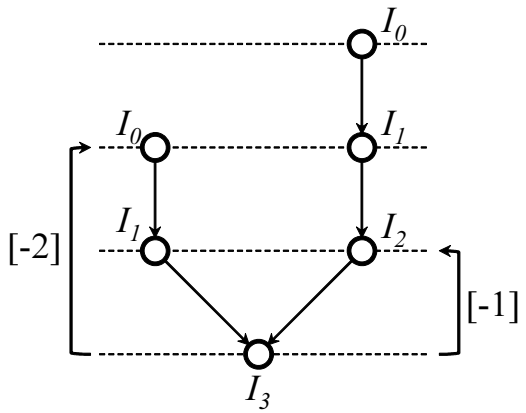


図 3.11: 図 3.10 の制御フロー・グラフ

3.2.2.1 経路毎の依存関係の変化

図 3.9 に示すコードを例として，依存関係の変位が経路毎に変化することを説明する．同図は途中に分岐命令を含んだコードの例である．図中のラベルは 3.1.2 節内のものと同様である．この図のコードをリネームド Op に変換した場合の実行フローが図 3.10 である．図 3.10(b) は I_1 の分岐が成立の場合，図 3.10(a) は分岐が不成立の場合である．図内のラベルの意味は 3.2.1.1 節で述べた，リネームド Op のフォーマットのものと同じである．そして，図 3.10 の制御フロー・グラフを図 3.11 に示す．同図では，各ノードは 1 つの命令を表す．また，開始点が同じ命令であっても実行経路が異なる場合は違うノードとして表してある．

これらの図では，I_1 の分岐命令の成立，不成立によって I_3 のソース・オペランドが異なっている．図 3.11 の左側のパスでは，分岐が成立したことによって，I_3 の依存元が 2 命令前の I_0 となっている．これに対し，右側のパスでは分岐が不成立であったために I_3 の依存元が 1 命令前の I_2 となっている．

3.2.2.2 パス

リネームド・トレース・キャッシュでは，実行経路を識別し，経路毎にリネームド Op のトレースをキャッシュする．この実行経路を表す情報を**パス**と呼ぶ．提案手法では，このパスを用いてトレースを一意に識別する．

パスの表現

パスの最もナイーブな表現は，経路上にある全命令アドレスの列である．しかし，長さ PL の命令アドレス列をそのままパスとして用いた場合，そのサイズが問題となる．たとえば命令アドレスのサイズが 64 ビットであり，PL が 32 命令であった場合，パスのサイズは $8 \times 32 = 256$ バイトにもなる．これは命令幅が通常 4 バイト程度であることを考えると，非常に大きい．

これに対し，提案手法では経路の先頭命令アドレスと経路内の分岐命令の制御情報を組み合わせることによって，パスをよりコンパクトに表現する．

以下ではこれを命令の種類毎に分けて説明する．一般に，ある命令の次の命令は，以下の様にして決定することができる．

1. **条件分岐命令**

条件分岐命令の次に実行される命令は，分岐の成立/不成立によって一意に決まる．

2. **間接分岐命令**
   間接分岐命令の次に実行される命令は，その分岐ターゲットによって一意に決まる．

3. **無条件分岐を含むそれ以外の全ての命令**
   無条件分岐を含むそれ以外の命令の場合，次に実行される命令は常に一意に決まる．

以上より，ある命令の次に実行される命令を決定するためには，条件分岐命令の分岐方向と間接分岐命令のターゲットがわかれば良い．したがって，これらの情報と経路の先頭の命令アドレスがわかれば，経路の先頭から逐次的に，次に実行される命令を決定できる．このため，これらの組み合わせは経路内の命令のアドレス列と等価な情報となる．

図 3.12 に具体例を示す．同図は制御フロー・グラフであり，ノードは命令を表す．ノードのうち，中が斜線によって塗りつぶされたものは分岐命令，白く塗りつぶされたものはそれ以外の命令を表す．同図では，命令 I_{t_start} に到達する経路は複数存在する．これらのうち，I_{p_start} から I_{t_start} に至る経路は，その間のパスによって一意に識別できる．この時のパスは，具体的には I_{p_start} の命令アドレスと，分岐命令である I_{br} の分岐方向によって表される．

パスの長さ

トレースを識別するために必要なパスの長さ PL は，命令ウィンドウ・サイズ WS（3.2.1.2 節）と，識別対象となるトレースの長さを TL を用いると，

$$PL = (WS-1)+(TL-1) \tag{3.1}$$

$$= WS+TL-2. \tag{3.2}$$

となる．以下では図 3.12 の例を用いてこれを説明する．同図において，破線で囲まれたトレースを識別することを考える．

1. **トレースの先頭への経路**
   式 3.1 の第 1 項は，トレースの先頭にある I_{t_start} への経路を識別するのに必要

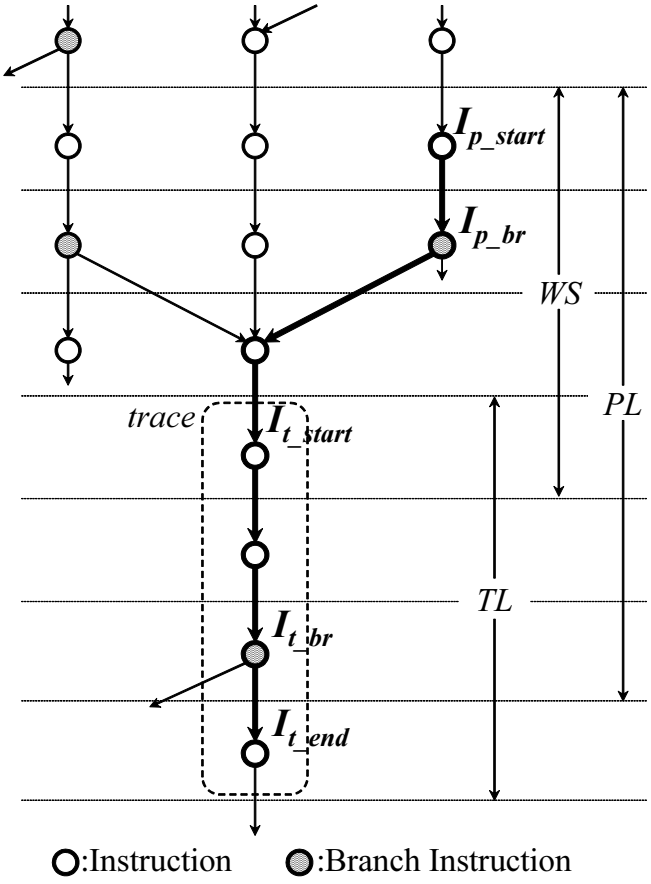


図 3.12: パスの例

な命令数である．この時，$WS-1$ 命令分のパスを見る必要がある．これは，ソース・オペランドの変位は最大で $WS-1$ 命令となるためである[6]．これ以上過去に実行された命令の実行結果は論理レジスタを介して受け取るため，トレースを識別する際に経路の一致を確かめる必要はない．

2. **トレース内の経路**

式 3.1 の第 2 項は，トレース内の経路を識別するために必要な命令数である．これは通常のトレース・キャッシュで用いられる，トレース内の経路情報と同様のものである．トレースの先頭にある I_{t_start} から最後尾にある I_{t_end} までの経路を識別するためには，$TL-1$ 命令が必要である．

[6]命令ウィンドウ中の最も後続にある命令が，命令ウィンドウ中の最も先行している命令に依存している場合がこれに該当する

以上より，これらを合計した結果である式 3.2 の命令数がパスには必要となる．

パスの情報量

以下では，パスを表現するために必要な情報量について，命令毎に分けて議論する．

1. **先頭の命令アドレス**
   パスの先頭の命令アドレスを表現するためには，命令アドレスのビット数が必要である．
2. **条件分岐命令**
   条件分岐の分岐方向を表現するために必要な情報は 1 bit である．パス内には最大で PL 命令の条件分岐命令が含まれるため，これを表すために PL bit が必要となる．
3. **間接分岐命令**
   分岐ターゲットを表現するためには，命令アドレスのビット数が必要である．このため，パス内の全ての命令について分岐ターゲットを保持すると，全命令の命令アドレスをそのまま保持しているのと変わらなくなってしまう．
   そこで，パスに含まれる間接分岐命令の個数を IJ_{max} 個に制限する．IJ_{max} 個を超える個数の間接分岐命令がパス内にある場合，キャッシュを行わずに毎回リネームド Op への変換を行うようにする．一般に，命令中に現れる間接分岐命令の数は多くないため，パス中の間接分岐命令の数を制限したとしても，大きな性能低下には繋がらない（3.4 節）．

3.2.2.3 パスの生成

本項では，パスの生成方法について説明する．図 3.13 にパスを生成する機構を示す．この機構は，主に分岐方向履歴，PC 履歴，間接分岐ターゲット履歴からなる．

分岐方向履歴は，g-share 予測器などで用いられるグローバル分岐履歴と同様のものである [23]．分岐方向履歴はシフト・レジスタら成り，n ビット目に $n+1$ 命令前の分岐方向の履歴を保持する．各要素は 1 の時に分岐成立，0 の時に分岐不成立を表す．また，分岐命令以外の命令の場合も 0 によって表す．m 命令をフェッチした場合，シフト・レジスタを m 命令をシフトし，そこにフェッチした命令の分岐

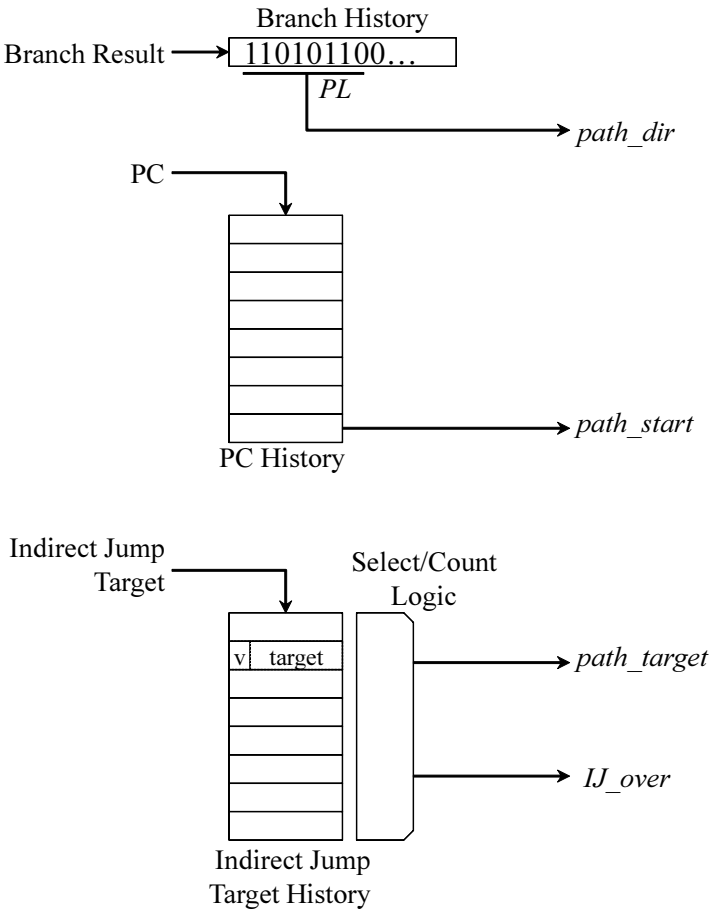


図 3.13: パスの生成

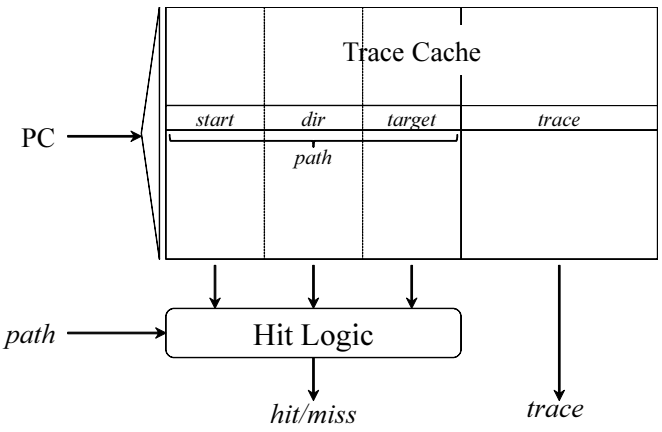


図 3.14: トレース・キャッシュの構成

情報を更新する．このシフト・レジスタの下位 PL ビットを取りだし，$path\_dir$ にパスの分岐方向履歴を出力する．

PC 履歴はシフト・レジスタから成り，n 要素目に $n+1$ 命令前の PC の履歴を保持する．このシフト・レジスタの $PL-1$ 要素目を取りだし，$path\_start$ にパスの先頭アドレスとして出力する．

間接分岐ターゲット履歴はシフト・レジスタとセレクタから成る．シフト・レジスタは，n 要素目に $n+1$ 命令前の間接分岐ターゲットの履歴を保持する．この表は各要素に valid ビットを備えており，この valid ビットが 1 の場合は，対応する命令は間接分岐命令であることを示す．セレクタは，シフト・レジスタの中から有効な間接分岐ターゲットを IJ_{max} 個選択し，$path_target$ 出力する．IJ_{max} は，前述したパス中に含むことが出来る間接分岐ターゲットの最大個数である．また，シフト・レジスタ中に IJ_{max} 個を超える間接分岐が存在した場合は，該当トレースはキャッシュ不能であることを IJ_over に出力する．

上記で説明した $path\_dir$，$path\_start$，$path_target$ を合わせたものをパスとして用いる．次項では，こうして生成したパスを用いて提案するトレース・キャッシュにアクセスを行う方法について説明する．

3.2.2.4 リネームド・トレース・キャッシュへのアクセス

図 3.14 に，提案するリネームド・トレース・キャッシュの機構を示す．通常のトレース・キャッシュでは，タグとしてトレースの先頭命令アドレスの一部と，分岐予測方向などを含む．これに対し，リネームド・トレース・キャッシュでは，3.2.2.3 節で生成したパスをタグとして使用する．通常のトレース・キャッシュと異なるのは，

1. トレースの先頭命令アドレスではなく，パスの先頭の命令アドレスを用いていること，
2. トレース内の分岐方向だけではなく，パス内の分岐方向も含んでいること，
3. パス内の間接分岐ターゲットを含んでいること

である．

リネームド・トレース・キャッシュの各エントリは，以下のフィールドからなる．

$path$:

3.2.2.3 節で述べた，パスを保持する．

trace :

リネームドOpのトレースを保持する.

トレース・キャッシュからの読み出しの際は，通常のトレース・キャッシュと同様にしてPCの一部をインデックスとし，キャッシュのセットを取得する．分岐予測器からの入力を元に作成したパスと，キャッシュから読み出されたタグが一致した場合，読み出されたトレースは有効である.

3.2.3 リネームド・トレース・キャッシュの効果

リネームド・トレース・キャッシュでは，ヒットしている限りにおいてはリネーム済みのリネームドOpがフェッチできるため，リネーミングとそのための機構を省略できる．これによる効果は以下の通りである.

1. **RMTの規模の縮小**
   通常の命令からリネームドOpへの変換はミス時にのみ行われるため，そのためのRMTの変換幅を削減することが出来る．変換幅の削減は，その回路規模と消費電力の大幅な削減に繋がる（3.2.1.5節）.
2. **パイプラインの短縮による性能向上**
   リネーム・ステージを省略することができるため，その分各種の予測ミス・ペナルティを削減することが出来る．また，パイプラインが短くなっただけ各種資源の解放も早くなるため，資源不足によるストールの発生確率も減少する.
3. **フェッチ幅の増大**
   通常のRMTではフェッチ幅に比例した数のポートが必要であるのに対し，リネームド・トレース・キャッシュはフェッチ幅に関わらず常に1ポートで構成できる．このため，リネーミングのための資源の増加を伴わずにフェッチ幅を増加させることができる．RMTを構成するRAMの回路面積はポート数の2乗に比例して大きくなる．このため，フェッチ幅を8命令以上にするためにはL1データ・キャッシュ以上の大きさのRMTが必要となり，現実的ではない[31].
   しかし，フェッチ幅と無関係に1 ポートで済むリネームド・トレース・キャッシュでは，これは十分実現可能である.
   また，通常のRMTではプログラム・オーダに従ってアクセスを行う必要があるが，リネームド・トレース・キャッシュへのアクセスはプログラム・オーダ

に従う必要はない．このため，フェッチをOut-of-Orderに行うなどの，さらなる工夫の余地がある．

これらの利点に対し，リネームド・トレース・キャッシュの問題点は以下の通りである．

1. **トレース・キャッシュの面積の増加**
   パスをタグに格納するため，その分だけ回路面積が増加する．ただし，3.4節の評価で示すように，この回路面積の増加は，RMTの縮小によって十分相殺可能である．
2. **キャッシュ・ヒット率の低下**
   パス毎にトレースをキャッシュするため，キャッシュ・ヒット率が低下してしまう場合がある．3.4節の評価で示すように，このヒット率低下による性能低下はそれほど大きなものではなく，リネーミング・ステージを省略することによる性能向上で相殺できる程度である．また，次節以降で述べる工夫により，このヒット率低下の影響を緩和することが出来る．

3.3 リネームド・トレース・キャッシュの詳細

本節では，リネームド・トレース・キャッシュについて詳細な議論を行う．3.3.1節ではリネームドOpへの変換について，それを実現するハードウェアの構成方法を説明する．3.3.2節では，リネームド・トレース・キャッシュのヒット率を向上させるいくつかの方法についてまとめる．

3.3.1 DMTの構成方法

3.2.1.5節で述べたナイーブなDMTの実装では，リネーム幅に比例した書き込みポートが必要である．これはリネームド・トレース・キャッシュにミスした場合に変換を再開するため，リネームド・トレース・キャッシュから取れてきたトレースを用いて，DMTの更新を常に行う必要があるためである．このため，通常のRMTと比較すると，読み出しポートは大幅に削減されているものの，書き込みポート数は変わらないことになる．

これに対し，以下ではフェッチ幅と無関係にDMTの書き込みポートを1ポートにまで削減する方法について述べる．

提案する方法は，CAM方式によるRMT[21, 28]と同様の構成を取る．これらのRMTでは，論理レジスタ番号をその内容とし，物理レジスタ数の容量を持つ．論理レジスタ番号をキーとして検索することにより，ヒットしたエントリの位置から物理レジスタ番号を得る．

提案するDMTもまた，論理レジスタ番号をその内容とし，物理レジスタ数の容量を持つ．以下，これを**CAM式DMT**と呼ぶ．また，区別のため，3.2.1.5節で述べたRAMを用いるナイーブな実装をRAM式DMTと呼ぶ．

CAM式DMTと，前述したRMTの主な違いは，CAM式DMTが命令ウィンドウを意識して優先順位付き検索を行う点である．このため，CAM式DMTは命令ウィンドウを表現するための$head$と$tail$の2つのポインタを持つ．$head$は命令ウィンドウ内の最も先行している命令を，$tail$は命令ウィンドウ内の最も後続にある命令をそれぞれ表す．

3.3.1.1 CAM式DMTの動作

図3.16に示す例を用いて，CAM式DMTの動作を説明する．同図は図3.15のコードによるDMTの更新の様子である．同図では，左から右に向かってI_1からI_4による更新が行われている．p2からp6は，それぞれI_1からI_4にシーケンシャルに割り当てられた物理レジスタである．また，影となっている部分は命令ウィンドウを表す．

CAM式DMTの更新

DMTの更新は，命令に割り当てられた物理レジスタに対応するエントリに，その命令のディスティネーション・オペランドである論理レジスタ番号を書き込む事によって行う．具体的には，$tail$をインクリメントした後，それが指す位置に論理レジスタ番号を書き込む．また，この時同時に$head$のインクリメントも行う．

図3.16の場合，左端にある初期状態では，$tail$はp2を指している．I_1による更新では，$tail$をインクリメントして得られたp3に対し，r2が書き込まれる．ここでr2はI_1のディスティネーション・オペランドである．

label	*op*	*opD*	*opL*	*opR*
I_0 :	mov	r1	← A	
I_1 :	sll	r2	← r1	<< 1
I_2 :	sll	r3	← r1	<< 2
I_3 :	add	r4	← r2	+ r3
I_4 :	add	r3	← r1	+ r4

(a) リネーミング前

label	*op*	*dst*	*srcL*	*srcR*	*imm*
I_0 :	mov	r1	← A		
I_1 :	sll	r2	← [-1]	<<	1
I_2 :	sll	r3	← [-1]	<<	2
I_3 :	add	r4	← [-2]	+ [-1]	
I_4 :	add	r3	← r1	+ [-1]	

(b) リネーミング後

図 3.15: シフトと加算により，7 倍を計算するコード（図 3.7 の再掲）

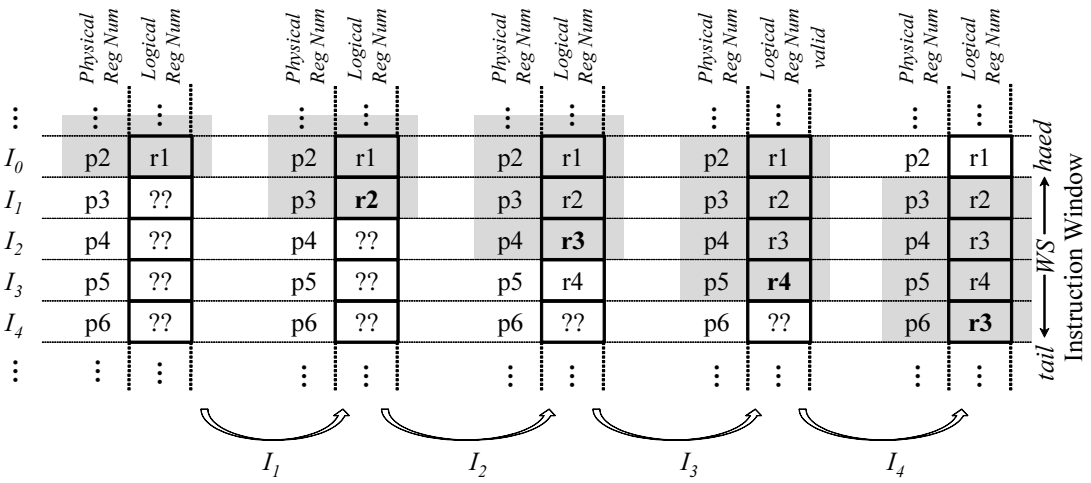


図 3.16: CAM 式 DMT の更新

CAM 式 DMT の参照

DMT の参照は，ソース・オペランドの論理レジスタ番号をキーとして，DMT を優先順位付き検索することによって行われる．この検索は $head$ と $tail$ 間にある命令についてのみ行われ，優先順位は $tail$ に近い命令ほど高く与えられる．$tail$ から検索結果の物理レジスタ番号を引く事により，最終的な命令間の変位を得る．なお，論理レジスタ番号が DMT にヒットしなかった場合，対応するレジスタは物理レジスタ中ではなく論理レジスタにある事を意味する．

以下では，図 3.16 の右端にある状態を使って参照の例を説明する．この状態は I_4 によって p6 に r3 が書き込まれた直後の状態である．$head$ は p3 を，$tail$ は p6 を指しており，この間の領域が命令ウィンドウである．

- r3 を検索した場合，命令ウィンドウ内では p4 と p6 がそれぞれ該当する．この場合，$tail$ により近い p6 が選ばれる．これは，論理レジスタ r3 を最も最近に上書きした I_4 を選択している事を意味する．
- r1 を検索した場合，命令ウィンドウ内には有効なエントリは存在しない．このため，r1 に対応するレジスタは論理レジスタ内にあることがわかる．

優先順位付き検索のロジック

優先順位付き検索に必要となるセレクト・ロジックについては，たとえば優先順位をサイクリックに変更可能なよう拡張した prefix-sum 方式 [43] を用いる．同方式では，自身より優先順位が高いエントリのヒット数を数えることにより，自身が選択されるかどうかを判定する．

このセレクト・ロジックのために必要な回路は，命令スケジューリングに用いられるセレクト・ロジックと同等か，より小さなものとなる．命令スケジューリングに用いられるセレクト・ロジックでは，命令ウィンドウ内から発行幅分の命令を選択するのに対し，DMT に用いるセレクト・ロジックでは 1 つを選択すればよいためである．また，DMT の場合はリネーム幅とは無関係に，1 命令当たりのソース・オペランドの数だけセレクト・ロジックがあればよい．このため，その総数もたかだか 2 個程度である．

3.3.1.2 CAM 式 DMT の効果

CAM 式の DMT には，以下のような効果がある．

1. 書き込みのマルチバンク化によりポート数を削減できる点
2. チェックポインティングを簡略化できる点
3. 変位の計算が簡略化できる点

以下では，これらについて説明する．

マルチバンク化による書き込みポート数の削減

前述したように，3.2.1.5 節で述べた RAM 式 DMT にはリネーム幅分の書き込みポートが必要となる．これに対し，CAM 式 DMT ではマルチバンク化を行う事によって，書き込みポート数減らす事が出来る．

一般に，複数エントリへのシーケンシャルなアクセスは，マルチバンク化によってポートを増やす事無く実現できる．リネームド・トレース・キャッシュでは，シーケンシャルに新しい物理レジスタを割り当て，書き込みを行う（3.2.1.2 節）．このため，DMT をフェッチ幅の分だけマルチバンク化することにより，フェッチ幅の増加に対して書き込みポート数を 1 に保つ事が出来る．

図 3.17 に 2 バンクにマルチバンク化を行った DMT の例を示す．同図の見方は基本的に図 3.16 と同じである．同図では偶数物理レジスタを Bank 0 に，奇数物理レジスタを Bank 1 に割り振っている．同図の状態から新たに 2 命令分の更新を行う場合，p6 と p7 が割り当てられ，書き込まれる．このようにして連続した番号に書き込まれる場合，同じバンクに対する書き込みは生じない．

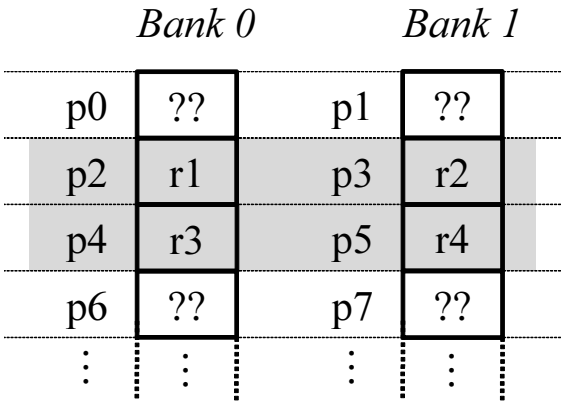


図 3.17: CAM 式 DMT のマルチバンク化

チェックポインティングの簡略化

通常の RMT では，分岐予測ミス時の回復のため，投機開始直前の状態を保存する**チェックポインティング**を行う．RMT に RAM を用いる方式では，RAM 全体を保存する必要があるため，多くの状態を保存するためには非常に高いコストが必要となる [20]．RMT に CAM を用いる方式では，RAM 全体を保存する必要はなく，各エントリにある valid ビットのみを保存すればよい [28, 21, 29]．

これに対し，CAM 式 DMT では投機開始直前の $head/tail$ ポインタを保存するだけでチェックポインティングを実現できる．投機ミスからの回復時には，ポインタを投機直前の状態に戻すだけでよい．これは，前述した RMT の場合と比べると非常に単純である．ポインタの保存領域以外の追加のハードウェアは必要なく，投機の回復も数 bit 程度のポインタを書き換えるだけですむため，非常に高速に実現することができる．

変位の計算の簡略化

RAM 式 DMT では，各エントリに，その論理レジスタに最後に上書きを行った命令の ID を格納する（3.2.1.5 節）．実際のハードウェアでは，各エントリは有限の幅を持つため，そのままでは ID のラップアラウンドが生じる．このため，RAM 式 DMT では，一定以上古い ID のエントリを定期的にクリアするなどの工夫が必要となる．

これに対し，CAM 式 DMT では上記の様な問題は発生しない．3.3.1.1 節で述べたように，リタイア済みの命令に対応するエントリは $head/tail$ ポインタの制御によって適切に検索から除外さる．また，依存する値が論理レジスタ内にあるかどうかの判定も，CAM 式 DMT へのヒット・ミス判定により自然に実現できる．

3.3.2 リネームド・トレース・キャッシュのヒット率の向上

本項ではリネームド・トレース・キャッシュのヒット率を向上させる方法として，パスの最小化と，インデックス生成方法の工夫による 2 つの方法を説明する．

3.3.2.1 パスの最小化によるトレース数の削減

リネームド・トレース・キャッシュでは，同じ命令アドレスであっても実行経路毎に異なるリネームド Op が生成されるため，キャッシュのヒット率を低下させてしまう．本節では，パスの長さをトレース毎に必要な長さまで短縮することにより，このヒット率の低下を緩和させる方法について説明する．

3.2.2.2 節や 3.2.2.3 節で述べた方法では，常に一定の長さのパスを用いてリネームド・トレース・キャッシュへのアクセスを行っていた．3.2.2.2 節で述べたように，この長さ PL_{max} は，リネームド Op において依存元として参照しうる最大の変位を元に決定される．

しかし，実際には PL_{max} までパスが一致している必要はない．パスは，トレース中のソース・オペランドの中で最大の変位を持つものが参照する距離まで一致していればよい．

図 3.18 に例を示す．同図は制御フロー・グラフであり，各ノードやラベルは 3.2.2.2 節内の図と同じ意味を示す．命令 I_t を先頭とするトレース内で，最も遠くにある依存元命令は I_p である．この場合，パスの一致判定は，PL_{max} 分行う必要はなく，PL_{min} 分行えば良い．これは，I_p よりも過去にどのような経路を経たとしても，トレース内と I_p 以降の命令間の依存関係には関係を及ぼさないためである．

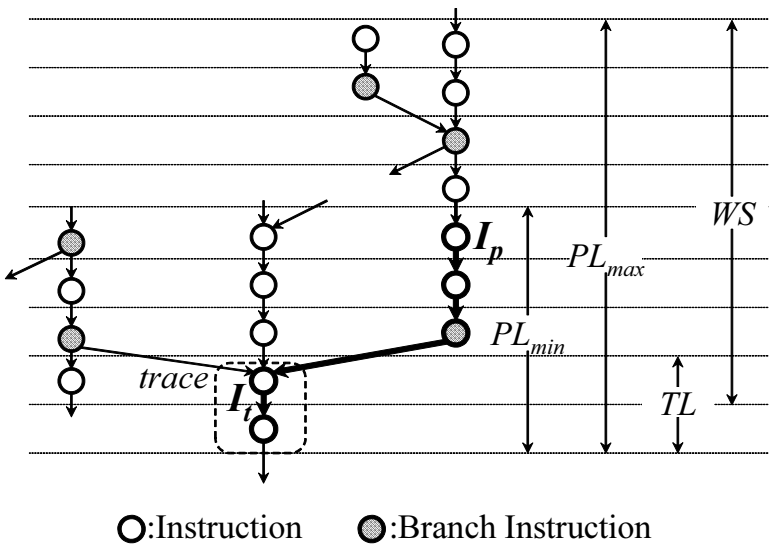


図 3.18: パスの最小化

なお，トレースの中に論理レジスタへの参照があった場合，上記とは別に扱う．論理レジスタへの参照の場合，依存元命令への経路が一致している必要はなく，WS 命令以上前に該当論理レジスタへの最後の書き込みが行われていることのみを確かめればよい．

上記の様にして一致比較を行うパスを最小化することにより，経路毎に生成されるトレースの数を大幅に減らすことが出来る．

3.3.2.2 リネームド・トレース・キャッシュ・アクセス時のインデックス

3.2.2.4 節で述べた実装では，キャッシュの一部のセットにトレースが集中してしまい，セットの競合ミスが増加する場合がある．これは主に，実行経路の合流点にある命令アドレスにおいて起きる．

以下では，図 3.18 の場合を例に挙げて，これを説明する．同図の見方は 3.3.2.1 節で説明したとおりである．同図では，I_t には 3 つの経路が合流している．このため，I_t を先頭とするトレースは経路に応じて 3 種類以上生成される．以下では簡単のため，ここで生成されるトレースは 3 種類であるとする．

3.2.2.4 節で述べた実装では，トレースの先頭にある I_t の命令アドレスの一部をキャッシュ・アクセス時のインデックスとして用いる．このため，I_t を先頭とする複数のトレースが全て同じセットに集中してしまう．

これを避けるため，実行経路上で n 命令前にある命令のアドレスを用いてインデックスを生成する．合流前の経路上にある命令を用いてインデックスとすることにより，トレースが配置されるセットを分散させることができる．たとえば図 3.18 の場合，I_t の 1 命令前の命令は全て異なる経路上にある．このため，1 命令前のアドレスをインデックスとすると，I_t を先頭とする 3 種類のトレースをそれぞれ別のセットに配置できる．

このインデックスの生成方法の変更は，リネームド・トレース・キャッシュのエントリ内容やその他のヒット/ミス判定部分の変更を伴わない．通常のセットアソシアティブ・キャッシュでは，アドレスの一部をインデックスとし，残りの部分をタグとして用いる．このため，インデックスの生成方法を変えた場合，ヒット/ミス判定を正しく行うために，タグにアドレス全体を格納するなどの変更をあわせて行う必要がある．リネームド・トレース・キャッシュの場合，そもそもトレースの先頭アドレスをエントリ内に含んでいないため（3.2.2.4 節），このような変更を

行う必要はない．キャッシュのヒット/ミス判定は，エントリに含まれるパスの先頭のアドレスと，そこからの分岐の制御情報によってのみ行われる．

3.4 評価

3.4.1 評価環境

鬼斬弐[44] にリネームド・トレース・キャッシュを実装し，評価を行った．鬼斬弐はサイクル・アキュレートなプロセッサ・シミュレータであり，プロセッサ・アーキテクチャの研究で広く用いられている SimpleScalar Tool Set[45] よりも高い精度でシミュレーションを行う事ができる．SimpleScalar Tool Set とは異なり，鬼斬弐は命令の実行を実際の実行ステージに正確なタイミングで行う．このため，SimpleScalar Tool Set では再現不能であるアドレス一致不一致予測のようなデータ予測の挙動を鬼斬弐では再現することができる．

評価には SPECCPU CINT2000 [46] に含まれる全 12 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 *train* を用い，最初の 1G 命令をスキップして，続く 100M 命令の評価を行った．

表 3.1 に，シミュレーションを行った際の構成を示す．同構成のキャッシュ容量や分岐予測器などのパラメータについては，最近のプロセッサのものに合わせてある．このベースラインとなるプロセッサにリネームド・トレース・キャッシュを実装して評価を行った．

3.4.1.1 評価モデル

以下のモデルについて評価を行った．

BASE:

評価のベースラインとなる，通常のトレース・キャッシュを持つモデル．通常の命令キャッシュへは，トレース・キャッシュにミスした時のみアクセスを行う．トレース・キャッシュのヒット/ミスが判明するのは，3 ステージあるフェッチ・ステージのうち，2 ステージ目とした．その他のパラメータについては表 3.1 にある通りである．

表 3.1: シミュレーション環境

Name	Parameter
pipeline stages	fetch:3, rename:2, dispatch:2, issue:2
fetch width	4 inst.
execution unit	int:2, fp:2, mem:2.
instruction window	32 entries
ROB	128 entries
branch predictor	8 KB g-share
branch miss penalty	9∼11 cycles
BTB	2 K entries, 4 way
RAS	8 entries
L1IC	32 KB, 4 way set assoc, 64 B/line, 3 cycles
L1DC	32 KB, 4 way set assoc, 64 B/line, 3 cycles
L2C	4 MB, 8 way set assoc, 64 B/line, 10 cycles
main memory	200 cycles
trace cache	2K entries, 8 way, trace:4 inst., branch:1 inst.

RTC:

リネームド・トレース・キャッシュを持つモデル．このモデルではリネームド・トレース・キャッシュの基本的な実装に加え，3.3.2 節で述べた改良も実装している．インデックスの生成時にさかのぼる命令数に関しては，11 命令とした．また，トレースあたりに含むことのできる間接分岐の数については，2 個とした．

リネームド・トレース・キャッシュでは，そのヒット/ミス判定が複雑となるため（3.2.2.4 節と 3.3.2 節），ヒット/ミスが判明するのは，フェッチ・ステージ中の最後である 3 ステージ目とした．

このモデルではリネームド・トレース・キャッシュにヒットし続ける限りは，レジスタ・リネーミングを行う必要がない．このため，表 3.1 にある rename ステージは，このモデルでは存在しない．リネームド・トレース・キャッシュのミス時にはリネームド Op への変換を行うが，この時の変換幅については，1 サイクルあたり 1 命令としている．

3.4.2 パス中に含む間接分岐ターゲットの数

タグの容量を削減するため，パスに含むことのできる間接分岐ターゲットの数を IJ_{max} 個に制限する（3.2.2.2 節）．IJ_{max} 個を超える間接分岐がパスにあった場合，リネームド・トレース・キャッシュから命令をフェッチするのではなく，DMT を用いてリネームド Op への変換を行う．DMT の変換はサイクルあたり 1 命令に制限されているため，この場合は性能が低下する．

本項では，この IJ_{max} を変化させた場合の，性能に対する影響について述べる．リネームド・トレース・キャッシュのその他の要因による影響を排除するため，本項の評価では，その容量を 64k エントリ（1MB）と十分に大きなものとした．

図 3.19 に，パス中の間接分岐数が IJ_{max} を超えたサイクルの，全実行サイクルに占める割合を示す．このグラフは，全ベンチマークにおける割合の平均をとったものである．グラフの数値は，IJ_{max} を 0 から 3 に振った場合のものである．ここで，IJ_{max} が 0 個と言うのは，パス中に間接分岐がある場合は，トレースのキャッシュを行なわないことを意味する．

また，図 3.20 に，IJ_{max} を 0 から 3 に振った場合の平均相対 IPC を示す．同図の IPC は，各ベンチマークにおける IPC を BASE の IPC によって正規化し，平均をとったものである．

IJ_{max} が 0 の場合，間接分岐が現れるたびにリネームド Op への変換が発生するため，全実行サイクルの 40%もの期間において変換が発生している．この結果，BASE に対して IPC は 23.7%も低下している．

これに対し，IJ_{max} が 1 以上の場合の変換の発生率は大幅に低く，1 の場合で 6.8%，2 の場合で 1.4%，3 の場合で 0.3%である．変換の発生率が下がった結果，IJ_{max} が 1 以上の場合は BASE と比較して IPC が若干向上している．これは，3.2.3 節で述べたパイプライン短縮の効果によるものである．

以上の結果より，IJ_{max} が 2 程度あれば，間接分岐数の超過による性能低下はほとんど発生しないと言える．このため以降では，IJ_{max} を 2 に固定して評価を行う．

3.4.3 リネームド・トレース・キャッシュのミス率と IPC

3.2.3 節で述べたように，リネームド・トレース・キャッシュではリネームド Op をパス毎に格納するため，キャッシュ・ミス率が増加する可能性がある．この影響

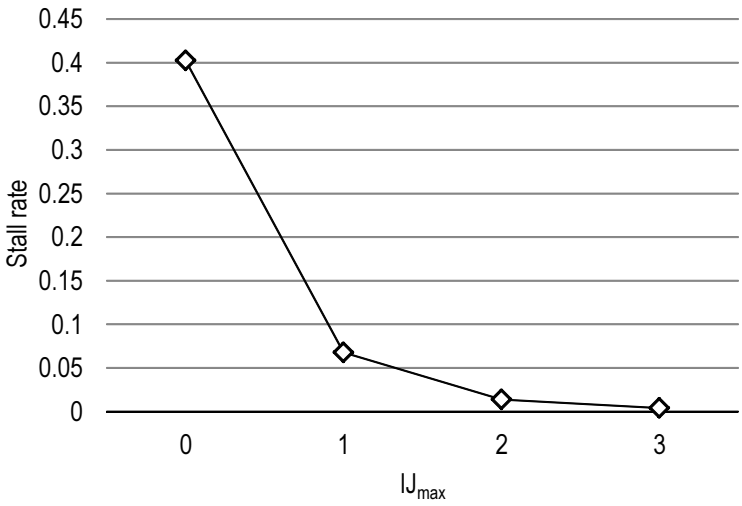


図 3.19: IJ_{max} 毎の間接分岐数超過によるリネームド Op 変換発生率

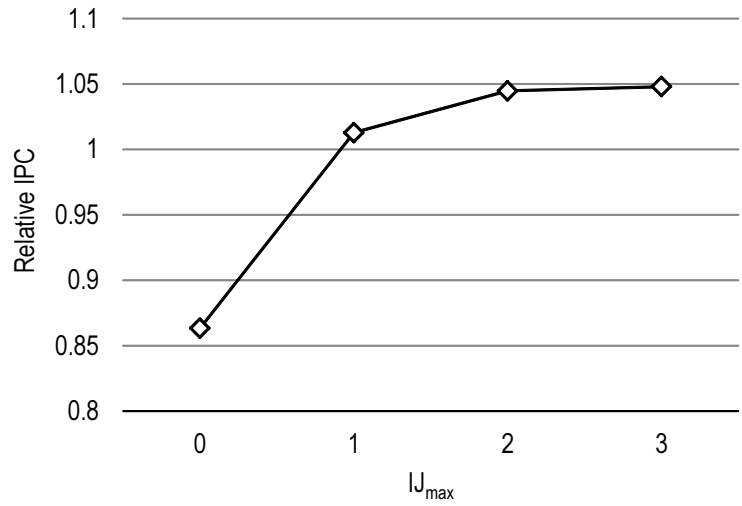


図 3.20: IJ_{max} 毎の平均相対 IPC

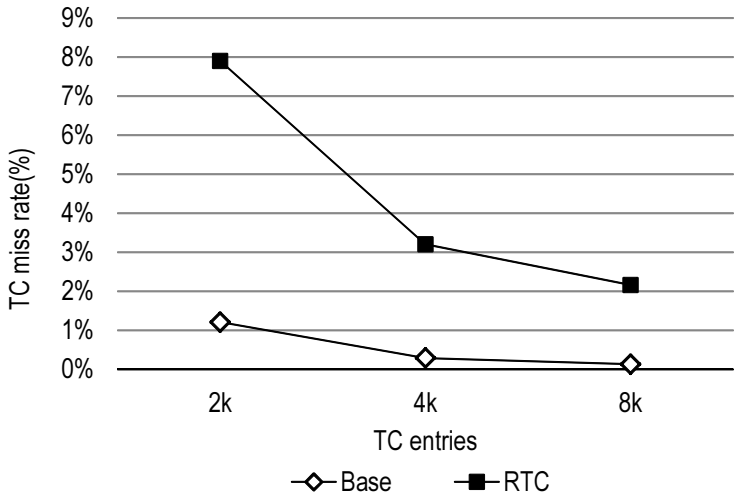


図 3.21: キャッシュのミス率

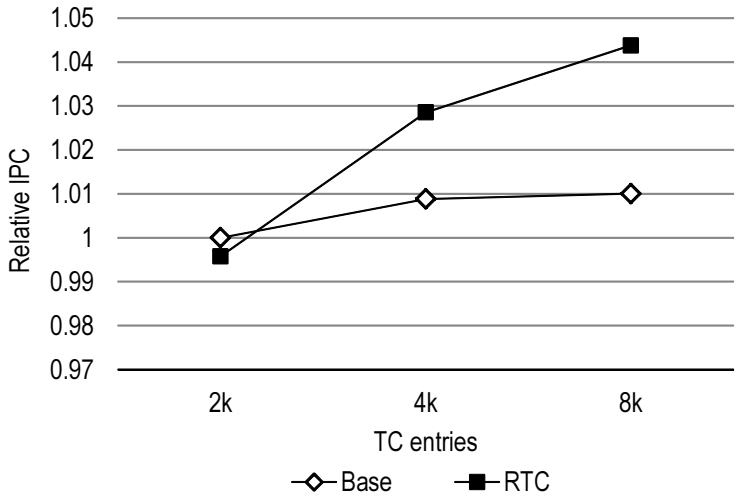


図 3.22: 平均相対 IPC

を測るため，リネームド・トレース・キャッシュの容量を変化させて評価を行った．

図 3.21 に，キャッシュのエントリ数を 2k から 8k に振った場合の，BASE と RTC のキャッシュ・ミス率を示す．RTC のミス率は，2k，4k，8k エントリのキャッシュの場合で，それぞれ 7.9%，3.2%，2.2%である．これは BASE のミス率がそれぞれ 1.2%，0.29%，0.13%であることと比べると，大幅に高い．

次に図 3.22 に,，キャッシュのエントリ数を 2k から 8k に振った場合の，BASE と RTC の平均相対 IPC を示す．同図は，各モデルの BASE（2k エントリ）に対する相対 IPC である．2k エントリのキャッシュを持つ場合，RTC の相対 IPC は 1.0 であり，BASE とほぼ同じ性能を持つ．これはキャッシュ・ミスによる性能低下が，パイプライン短縮による性能向上で相殺されたためである．4k エントリ以上の容量を持つ場合，RTC は 4k エントリの場合で 2.9%，8k エントリの場合で 4.4 の性能向上を見せている．

以上より，2k エントリ程度のリネームド・トレース・キャッシュであれば，キャッシュ・ミス率の増加による性能低下は，パイプライン短縮による性能向上で十分相殺可能であると言える．

3.4.4 回路面積と消費電力

CACTI 5.3 モデル [3] を用いて，各モデルにおける回路面積と消費電力を評価した．評価は，ITRS の 45 nm と 32 nm テクノロジ・ノード [47] を用いて行ったが，両者はほぼ同じ傾向を示したため，以下では 32nm のものについてのみ示す．

BASE のリネーム・ロジックは CAM 方式 [28, 21, 29] によって構成し，CAM 部分のみを評価した[7]．RTC については，3.3.1 節で述べた CAM 式 DMT のうち，同じく CAM 部分のみを評価した．

3.4.4.1 回路面積

図 3.23 にトレース・キャッシュとリネーム・ロジックの相対回路面積を示す．グラフ中の各値は，BASE のトレース・キャッシュとリネーム・ロジックの回路面積の合計に対する相対値となっている．グラフ中の “trace cache” は，BASE ではトレー

[7]RAM 方式では特殊なセルなどを用いてチェックポインティングを行うが [20]，使用した CACTI ではそのようなモデルは取り扱えないため，ここでは CAM 方式を評価した．

ス・キャッシュの面積を，RTC ではリネームド・トレース・キャッシュの回路面積をそれぞれ表す．また，“mapping table” は，BASE では RMT の面積を，RTC では DMT の回路面積をそれぞれ表す．グラフでは，BASE は 2k エントリのトレース・キャッシュについて，RTC は 2k，4k，8k エントリのリネームド・トレース・キャッシュについての結果を載せている．

RTC の DMT は，BASE の RMT と比較して 5.1% にまで縮小されている．これは BASE の RMT が 16 ポートであるのに対し，RTC の DMT は 3 ポートであるためである．RAM の回路面積はポート数の 2 乗に比例するため，結果として回路面積は大幅に縮小される．$(3/16)^2 \approx 3.5\%$ であり，実際の縮小率である 5.1% におおよそ一致する．

RTC による 2k エントリのリネームド・トレース・キャッシュと DMT の合計回路面積は，BASE によるトレース・キャッシュと RMT の合計回路面積と比べて 73.1% にまで縮小されている．回路面積が RMT のみの場合と比べて縮小されていないのは，リネームド・トレース・キャッシュの回路面積が増加しているためである．RTC の 2k エントリのリネームド・トレース・キャッシュは，BASE のトレース・キャッシュと比較して 48.1% 回路面積が増加している．これは，パスを格納するために追加の回路が必要なためである（3.2.3 節）．この回路面積の増加は RMT の回路面積の縮小分よりも小さいため，結果として，2k エントリの場合は全体で回路面積

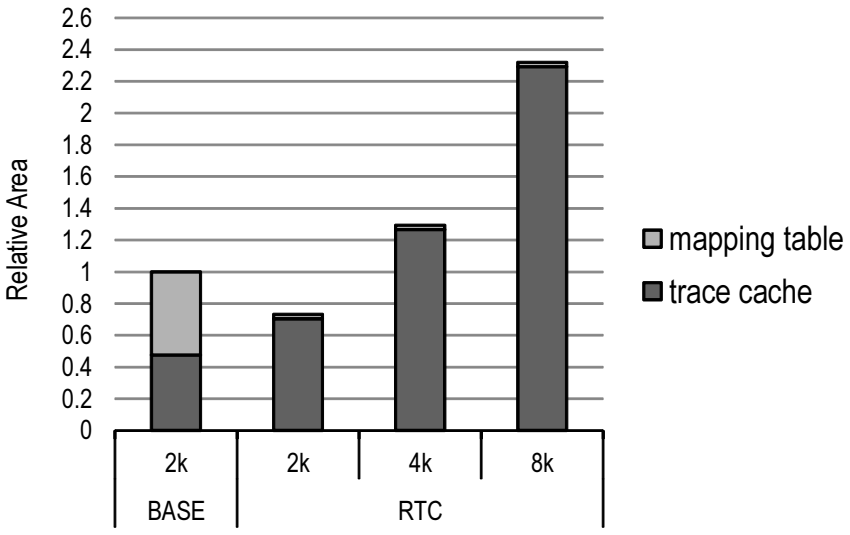


図 3.23: 相対回路面積

は縮小されている．

4k と 8k エントリのリネームド・トレース・キャッシュの場合，BASE と比較して回路面積はそれぞれ 26.6%と 129%増加してしまっている．これは 4k と 8k エントリのリネームド・トレース・キャッシュが 2k エントリのトレース・キャッシュと比較して，それぞれ 166%と 382%も回路面積が増加しているためである．

3.4.4.2 消費電力

図 3.24 に，各モデルの BASE モデルに対する相対消費電力を示す．グラフ中の各ラベルは図 3.23 の場合と同じ意味を示す．グラフ中の各値は，BASE のトレース・キャッシュとリネーム・ロジックの回路面積の合計に対する相対値となっている．これらの消費電力はプログラムごとに評価を行ったものを，全体で平均したものである．

2k と 4k エントリの RTC では，BASE と比較して，それぞれ 46.5%と 7.4%消費電力が削減されている．これらの消費電力の削減は，回路面積の削減よりも大きなものとなっている．これは，3 節で述べたように，RMT が非常に多くのアクセスを受けるためである．トレース・キャッシュや RMT を構成する RAM や CAM の消費電力は，そのアクセス数に比例する．通常，トレース・キャッシュ が 1 サイク

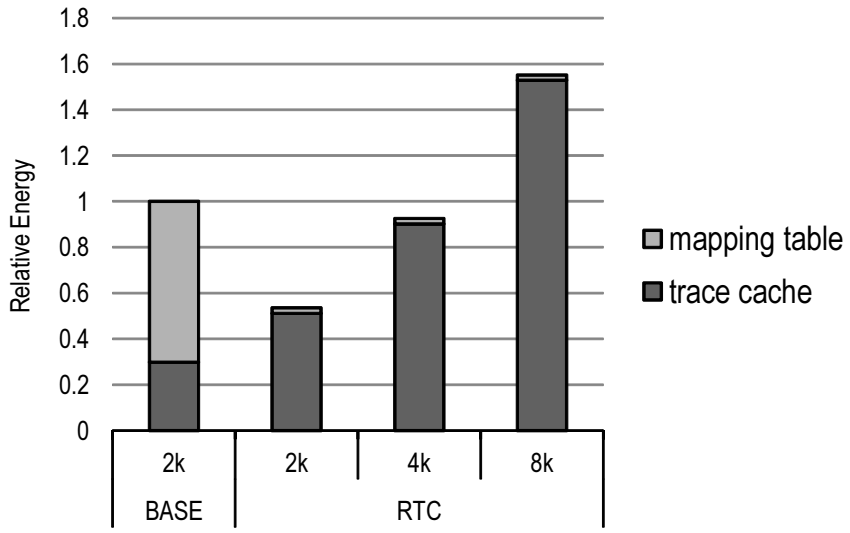


図 3.24: 相対消費電力

ル当たりにつき1回しかアクセスを受けないのに対し，RMTはリネームを行う全命令のオペランド数に応じたアクセスを受ける．このため，RMTの消費電力はトレース・キャッシュと比べると相対的に大きくなるのである．

3.5 関連研究

レジスタ・リネーミングをそもそも行わない命令セット・アーキテクチャ として，**Dualflow アーキテクチャ**が提案されている [48, 49]．Dualflow アーキテクチャでは，通常の制御駆動型の命令セット・アーキテクチャのようなレジスタを定義しない．命令間のデータの授受は，制御駆動のようにレジスタを介して間接的に行われるのではなく，データ駆動型アーキテクチャ[50, 51, 52] のように命令間で直接的に行われる．この Dualflow アーキテクチャの命令は通常コンパイラによって静的に生成される．

Dualflow アーキテクチャは，元々は命令スケジューリング・ロジックを簡略化するために提案されたものである．しかし，データの授受を命令間で直接的に行うため，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．この点において，Dualflow アーキテクチャの命令はいわば，レジスタ・リネーミング済みであると考えることができる．

Dualflow アーキテクチャの問題は，命令の配置に関して強い制約があることである．Dualflow アーキテクチャでは，データ駆動型アーキテクチャのように，依存先命令を n 命令後というように命令間の変位で指定する．そのため，たとえば分岐命令を超えてデータを受け渡すためには，分岐が成立した側と不成立であった側で依存先の命令の位置を揃えなければならない．また，命令中の変位を指定するフィールドの幅が有限であることから，それによって表現できるよりも遠い命令にデータを送信する場合には中継命令を挟む必要がある．

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．これらの追加される命令のため，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，Dualflow アーキテクチャでは，命令数が大幅に増加してしまっていた．また，データの送信先を指定する命令セット・アーキテクチャであるため，送信先がディスティネーション・オペランドの個数に制限されてしまう問題もある．送信先が不足する場合は中継命令を挿入する必要があるが，これもまた命令数を増大させる要因となっている．

リネームド・トレース・キャッシュは，このDualflowアーキテクチャに着想を得たアーキテクチャである．Dualflowアーキテクチャとリネームド・トレース・キャッシュの最も大きいな違いは，命令の授受を指定するための変位の方向の違いにある．

Dualflowアーキテクチャでは，ディスティネーション・オペランドにおいてn命令後と言う形でデータの送信先を指定している．このため，先に述べたような命令の配置に関する問題がある．一方，リネームド・トレース・キャッシュでは，ソース・オペランドにおいてn命令前という形でデータの参照元を指定している．このため，Dualflowアーキテクチャにあるような命令の配置に関する制約は無い．また，論理レジスタを導入することにより，変位が大きい場合の転送命令の必要を無くしている．

リネームド・トレース・キャッシュでは，通常の制御駆動型の命令からリネームドOpへの変換を行っている．これに対し，このような命令の動的な変換をDualflowアーキテクチャで行う事は難しい．これはDualflowアーキテクチャの命令が，データ授受の際にn命令後と言う形で未来に実行される命令を指定しているためである．また，中継命令やNOP命令の挿入により，分岐パス毎にデータ送信先命令の位置を揃えることも行わなければならない．このため，命令の後続にどのような制御構造があるかを事前に把握する必要がある．この変換は，コンパイラ上であればコード構造の静的な解析によって，十分実現可能である．しかし，プロセッサにおいて動的にコードの制御構造を解析することは非常に困難である．これに対し，リネームド・トレース・キャッシュでは，いわば過去に実行された命令を指定してデータの授受を行うため，複雑な制御の解析は不要である．このため，3.2.1.5節で述べた様な現実的なハードウェアで動的な変換を実現できる．

3.6 本章のまとめ

レジスタ・リネーミング のために必要なロジックは近年巨大化する傾向にあり，Out-of-Orderスーパスカラ・プロセッサ・コアの中でも最も高コストなものの1つとなっている．

本章では，レジスタ・リネーミングの結果をトレース・キャッシュにキャッシュするリネームド・トレース・キャッシュの提案を行った．リネームド・トレース・キャッシュでは，キャッシュにヒットし続ける限り，レジスタ・リネーミングを行う必要はない．レジスタ・リネーミングを行うのはリネームド・トレース・キャッ

シュにミスした時のみであり，この時に使うDMTは1サイクルあたり1命令程度を処理できれば十分である．このため，通常のRMTと比べると，そのポート数を大幅に少なくすることができる．また，リネームド・トレース・キャッシュからのリネーム結果の読み出しは，リネーム幅とは無関係に1ポートで行う事ができる．このためリネームド・トレース・キャッシュでは，リネーム幅に比例したポートが必要となる通常のRMTとは異なり，回路を巨大化させることなくリネーム幅を広くすることができる．

評価の結果，リネームド・トレース・キャッシュは0.4%の性能低下で，リネーム・ロジックの回路面積を5.1%にまで縮小できることを示した．ただし，リネームド・トレース・キャッシュにパスを格納するために追加の回路が必要となるため，リネーム・ロジックとトレース・キャッシュを合わせた面積の縮小率については73.1%に留まる．

今後の課題は，パスの格納のためにリネームド・トレース・キャッシュに追加される回路の縮小である．通常，実行命令中に含まれる間接分岐命令は比較的少数であるため，全てのトレースのパスにこれを含む必要はない．このため，現在，パス情報の大部分を占める間接分岐ターゲット・アドレスを圧縮することを考えている．

第4章

非レイテンシ指向レジスタ・キャッシュ・システム

物理レジスタ・ファイルとそのバイパス・ネットワークは，最近のスーパスカラ・プロセッサの構成要素の中でも最も高コストなものとなっている．

この要因として，まず，物理レジスタ・ファイルの容量の増加がある．通常，物理レジスタ・ファイルにはインフライト命令数に応じた容量が必要となる．より高い命令レベル並列性を抽出するため，近年ではスーパスカラ・プロセッサのインフライト命令数は増加する傾向にあり，それにあわせて物理レジスタ・ファイルの容量も増大している．また，SMT をサポートするコアでは，スレッド数に比例した容量の物理レジスタ・ファイルが必要となるため [34, 35]，このことも容量の増大に寄与している．

次に物理レジスタ・ファイルのポート数の増加がある．一般に，4 命令同時発行可能なスーパスカラ・プロセッサでは，レジスタ・ファイルに 8 つのリード・ポートと 4 つのライト・ポートが必要となる．レジスタ・ファイルは通常，多ポートの RAM によって構成されるが，RAM の回路面積はポート数の 2 乗に比例して大きくなるため [2, 3]，その回路面積は容量に対して非常に大きなものとなる．図 4.1 に Intel Pentium 4 プロセッサのチップ写真を示す[1]．Intel Pentium 4 プロセッサでは，レジスタ・ファイルの容量は，L1 データ・キャッシュのそれと比較して 1 桁以上小さい．それにも関わらず，同図ではレジスタ・ファイルの回路面積は L1 デー

[1]Intel Pentium 4 プロセッサ は 6-read/3-write のレジスタ・ファイル を倍速で動作させることにより，12-read/6-write のレジスタ・ファイルとして用いている [5].

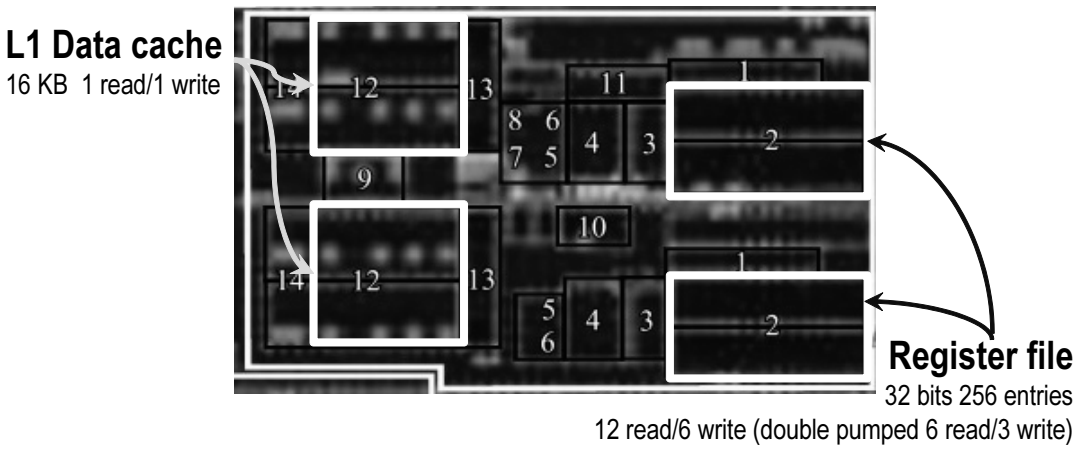


図 4.1: Intel Pentium 4 プロセッサ の整数実行コア [1].

タ・キャッシュのそれに匹敵するほど大きいことがわかる．このことは，最近市場に投入されたプロセッサであっても変わらない．

巨大なレジスタ・ファイルは，IPC の低下や複雑さの増大，消費電力，熱などの様々な問題を引きおこす．

前述したように，レジスタ・ファイルの回路面積は，L1 データ・キャッシュのそれに匹敵するほど大きい．このため，レジスタ・ファイルのアクセス・レイテンシもまた，L1 データ・キャッシュに匹敵するほど大きなものとなる．したがって，近年では L1 データ・キャッシュで行われているのと同様にして，レジスタ・ファイルのアクセスもまたパイプライン化されており，2 から 3 ステージ程度をそのアクセスに割り当てるのが普通である [5, 53, 22]．しかし，このレジスタ・ファイル・アクセスのパイプライン化は以下の様な問題を引きおこす．

IPC の低下

まず，パイプラインが深化することの一般的な悪影響として，IPC の低下が挙げられる．パイプラインが深くなればその分だけ，分岐予測をはじめとする各種予測ミス・ペナルティが増加する．また，命令ウィンドウ・エントリや物理レジスタ・エントリ自体の解放が遅れるため，資源不足によってフロントエンドがストールする確率も増える [36, 37]．しかし，これらによる性能低下はあくまで間接的なものであり，深刻なものではない．パイプライン 1 段あたりの性能低下は，最悪の場合

でもたかだか2% 以下である [54].

複雑さや消費電力，熱の増大

第 2 の問題はレジスタ・ファイルのレイテンシに固有の問題である．この問題は，第 1 の問題である IPC の低下よりも深刻である．一般に，レジスタ・ファイル・アクセスのパイプライン化はバイパス・ネットワークの複雑化を引きおこす [55]．レジスタ・ファイルのレイテンシが l サイクルであった場合，命令は $2l$ サイクルの間，バイパス・ネットワークを介して値を受け取らなければならない．このバイパス・ネットワークは通常，長い配線と非常に幅の広いマルチプレクサによって構成される．このため，バイパス・ネットワークはプロセッサ内でも最もクリティカルな部分の一つであり，プロセッサの動作周波数を制限する．また，その消費電力も非常に大きい．

バイパス・ネットワークの消費電力に加え，レジスタ・ファイルそのものによって消費される電力もまた，非常に大きな問題である．一般に，RAM の消費電力は，その回路面積とアクセスの頻度に比例して大きくなる [2, 3]．ロード/ストア命令が L1 データ・キャッシュに対してそれぞれ 1 回しかアクセスを行わないのに対し，ほぼ全ての命令は通常，レジスタ・ファイルに対して複数回のアクセスを行う．このため，面積が同程度の L1 データ・キャッシュと比較して，レジスタ・ファイルはより大きな電力を消費する．また，SMT プロセッサでは命令の実行スループットを増大させるため，アクセスの頻度もより高くなる．

消費電力とそれによって発生する熱は，最近のプロセッサ・コアにおける問題の中でも，もっとも深刻なものの一つである．レジスタ・ファイルとバイパス・ネットワークを含む領域は，プロセッサ・コア内の**ホット・スポット**であり，その動作周波数を制限する．

複雑さを下げる既存手法

レジスタ・ファイルやバイパス・ネットワークの複雑さを緩和するため，プロセッサ資源をナイーブな方法で削減する手法がいくつか提案されている．

バイパス・ネットワークに対しては，**インコンプリート・バイパス**を行う手法が提案されている [56, 7, 8]．インコンプリート・バイパスでは，演算直後の $m < 2l$ となる期間のみ演算結果のバイパスを行う．依存元命令の実行から m サイクル経過後は，レジスタ・ファイルから値を取得できるまで依存先命令の発行を見合わせ

る必要がある．

レジスタ・ファイルのポート数を単純に削減した場合，回路面積の縮小によりレジスタ・ファイルのレイテンシが削減されるため，バイパス・ネットワークの複雑さもまた削減される．この場合，レジスタ・ファイルのポートが不足した場合にプロセッサのパイプラインは乱れることになる．

これらの手法では，複雑さとIPCはトレードオフの関係にある．後の4.5節の評価で示すように，これらの手法では，IPCは最悪の場合20%以上低下する．

レジスタ・キャッシュ

これらの問題を解決するよりよい手法として，**レジスタ・キャッシュ**が提案されている[6, 7, 8]．レジスタ・キャッシュはその名の通りメイン・レジスタ・ファイルのキャッシュである．通常，レジスタ・キャッシュは1サイクルでアクセスが可能なよう，その容量を制限する．レジスタ・キャッシュがヒットし続ける限り，レジスタ・キャッシュは1サイクルでアクセス可能なレジスタ・ファイルと等価となる．

このレジスタ・キャッシュには，以下の2つの効果がある．

まず，レジスタ・キャッシュは1サイクルでアクセス可能なレジスタ・ファイルと等価であるため，アクセスのパイプライン化によるIPCの低下を回避することができる．しかし，この効果はあまり大きなものではない．なぜなら，先にも述べたように，アクセスのパイプライン化によるIPCの低下は通常十分小さいためである．

次に，レジスタ・キャッシュは以下のように，レジスタ・ファイルの複雑さや消費電力，そして熱を削減することができる．

1. **バイパス・ネットワークの簡略化:**
   バイパス・ネットワークはレジスタ・キャッシュにのみ必要であり，メイン・レジスタ・ファイルには必要ない．このため，バイパス・ネットワークは1サイクルでアクセス可能なレジスタ・ファイルのそれと同程度にまで縮小される（4.1節）．
2. **メイン・レジスタ・ファイルのポート数の削減:**
   レジスタ・キャッシュにミスしたオペランドのみがメイン・レジスタ・ファイルにアクセスを行うため，メイン・レジスタ・ファイルのポートを削減することできる．4.5節で示すように，レジスタ・キャッシュを用いることによって

メイン・レジスタ・ファイルのポート数を 12 から 4 程度にまで減らすことができる．

このように，理想的な状況下ではレジスタ・キャッシュはレジスタ・ファイルによる問題を解決することができる．しかし，レジスタ・キャッシュでは，先に述べたのとは別の理由により，大きな IPC の低下を引きおこしてしまう．後の 4.2 節で述べるように，レジスタ・キャッシュ・ミス時のメイン・レジスタ・ファイルのレイテンシは Out-of-Order スケジューリングでは隠蔽することができない．また，実際にパイプラインが乱れる確率である**実効ミス率**は，レジスタ・キャッシュそのもののミス率よりも大幅に大きい．プロセッサの命令パイプラインは，同時にレジスタ・キャッシュにアクセスを行うオペランドのうち，1 つ以上がミスを起こした場合に乱れる．たとえば，4.5 節で示すように，SPEC CPU 2006 の 456.hmmer では，レジスタ・キャッシュのヒット率は 94.2% と十分高い．しかし，1 サイクルあたりにレジスタ・キャッシュにアクセスを行う平均オペランド数は 2.49 個あることから，実効ミス率の理論値は $1 - 0.942^{2.49} \approx 13.9\%$ にもなってしまい，結果として 10.2% も性能が低下してしまう．

非レイテンシ指向レジスタ・キャッシュ・システム

本章では，**非レイテンシ指向レジスタ・キャッシュ・システム**（**NORCS**：**Non-Latency-Oriented Register Cache System**）を提案する．ここで，レジスタ・キャッシュの**システム**とは，レジスタ・キャッシュとメイン・レジスタ・ファイル，そしてそれらにアクセスを行う命令パイプライン・ステージまでを含めた系のことを指す．

議論の簡単のため，以降では既存のレジスタ・キャッシュ・システム を**レイテンシ指向レジスタ・キャッシュ・システム**（**LORCS**：**Latency-Oriented Register Cache System**）と呼ぶことにする．NORCS は，その名のとおり，レイテンシの短縮を目的としないレジスタ・キャッシュ・システムである．NORCS と LORCS の間では，レジスタ・キャッシュとメイン・レジスタ・ファイルそのものには大きな違いはない．NORCS の特徴は，その命令パイプラインにある．

LORCS は**ヒットを仮定したパイプライン**を持つ．LORCS のパイプラインでは，レジスタ・キャッシュへのアクセス・ステージは存在するが，メイン・レジスタ・ファイルへのアクセス・ステージは存在しない．これは，通常の L1 データ・キャッ

シュと同様の構成である．一般的な命令パイプラインでは，L1 データ・キャッシュへのアクセス・ステージは存在するが，L2 データ・キャッシュへのアクセス・ステージは存在しない．L1 キャッシュ・ミス時には，L2 キャッシュへのアクセスを行うために，パイプラインをストールかフラッシュする必要がある．これは，パイプラインに発行された後では，命令の再スケジュールをおこなうことができないためである．このことは，LORCS の場合でも同様である．LORCS ではレイテンシの短縮を行ったがために，結果としてパイプラインの乱れによって非常に大きな性能低下をこうむってしまう．

これに対し，NORCS は**ミスを仮定したパイプライン**を持つ．NORCS の命令パイプラインはメイン・レジスタ・ファイルへのアクセス・ステージを持ち，全ての命令はレジスタ・キャッシュのヒット・ミスに関わらずメイン・レジスタ・ファイルへのアクセス・ステージを通過する．レジスタ・キャッシュのミス時には，メイン・レジスタ・ファイルはそのアクセス・ステージの最後で値を供給する．レジスタ・キャッシュのヒット時には，ミス時と同様にメイン・レジスタ・ファイルのアクセス・ステージの最後にレジスタ・キャッシュが値を供給する．このように，NORCS ではレジスタ・ファイルのレイテンシを短縮していないため，そもそも IPC は向上しない．その代わり，NORCS では LORCS で問題となるパイプラインの乱れも発生しない．これはメイン・レジスタ・ファイルへのアクセスに必要な時間が，パイプライン中に組み込まれているアクセス・ステージによって常に得られるためである．つまり，NORCS では，IPC は必ずしも向上しないが，逆に大きく下がる事もないのである．NORCS では，このようにして性能を保ったまま，LORCS で行っているのと同様にして複雑さを緩和することができる．

レイテンシを短縮しないキャッシュというアイデアは，一見奇妙に聞こえるかもしれない．通常の場合，L2 キャッシュと同じレイテンシを持つ L1 キャッシュと言うものは，全く無意味である．ある教科書では，キャッシュを“最近アクセスされたコードやデータを保持する，小さくて**速い**メモリ”と定義している [57]．この定義に従えば，NORCS はもはやキャッシュではない．なぜなら，NORCS は**速くない**からである．レイテンシを短縮しないキャッシュ・システムはレジスタ・ファイルにのみ適用可能であり，NORCS はレイテンシを短縮しない唯一のキャッシュ・システムである．あとの 4.4 節では，なぜレイテンシを短縮しないキャッシュ・システ

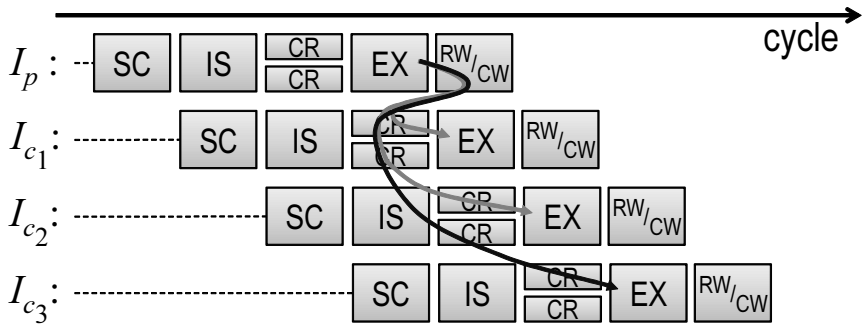


図 4.2: レジスタ・キャッシュ・ヒット時の LORCS のバックエンド・パイプライン

ムがレジスタ・ファイルにのみ適応可能であるのかについて，詳しく説明を行う．

本章の構成は，以下の通りである．4.1 節 と 4.2 節 では，背景となる LORCS の説明を行う．4.3 節で NORCS の詳細な構成について述べた後，4.4 節 では詳細な議論を行う．その後 4.5 節で評価結果を示す．

4.1 LORCS

本節では，LORCS（レイテンシ指向レジスタ・キャッシュ・システム）のうち，主にレジスタ・キャッシュ・ヒット時の挙動とその効果について説明する．巨大なレジスタ・ファイルによる問題のうち，特にレジスタ・ファイルやバイパス・ネットワークの複雑さや消費電力，熱の増大がどのようにして解決されるかについて述べる [6, 7, 8]．なお，レジスタ・キャッシュ・ミス時の挙動については，次節で詳しく述べる．

4.1.1 LORCS の命令パイプライン

図 4.2 に LORCS のバックエンド・パイプラインの挙動を示す．同図の LORCS では，レジスタ・キャッシュとメイン・レジスタ・ファイルのレイテンシはそれぞれ 1 サイクルとなっている．各ステージのラベルの意味については，それぞれ以下の通りである．

SC 命令スケジューリング

IS 発行

EX 実行

CR レジスタ・キャッシュからの読み出し

CW レジスタ・キャッシュへの書き込み

RW メイン・レジスタ・ファイルへの書き込み

RW/CW ステージについては，メイン・レジスタ・ファイルに対する書き込みとレジスタ・キャッシュに対する書き込みが同時に行われる事を表す．CR ステージが縦に 2 段に分かれているのは，各命令の 2 つのソース・オペランドが同時にレジスタ・キャッシュに対してアクセスを行う事を表す．

レジスタ・キャッシュにヒットし続ける限り，LORCS は 1 サイクルでアクセス可能なレジスタ・ファイルを持つシステムと等価に振る舞う．この時，図 4.2 のように，メイン・レジスタ・ファイルからの読み出しサイクルは現れない．L1 データ・キャッシュの場合と同様に，命令パイプラインはレジスタ・キャッシュ・ミス時に乱れる．このレジスタ・キャッシュ・ミス時の挙動については，次節で詳しく述べる．

4.1.2 レジスタ書き込み

実行結果の書き込みは，メイン・レジスタ・ファイルに書き込まれる前に一時的に**ライト・バッファ**に保持される．

このライト・バッファを介したメイン・レジスタ・ファイルへの書き込みは**ライトスルー・ポリシ**に従って行われる．すなわち，命令の実行結果は，実行ステージの直後にある RW/CW ステージにおいてレジスタ・キャッシュとライト・バッファの双方に同時に書き込まれる．

通常のデータ・キャッシュの場合とは異なり，**ライトバック・ポリシ**を用いても，メイン・レジスタ・ファイルへの書き込みを減らす事はできない．通常のデータ・キャッシュの場合，同じアドレスに対するストアは一般によく生じる．このため，ストアそのものの回数に比べてライトバックの回数を十分少なくすることができ

る．これに対し，レジスタ・キャッシュ・システムの場合では，同一のレジスタに対する上書きはめったに発生しない．これは Out-of-Order スーパスカラ・プロセッサの行うレジスタ・リネーミングにより，そのような同一のレジスタに対する上書きがあらかじめ取り除かれているからである．

レジスタ・キャッシュ・システムにおけるレジスタの上書きは，分岐予測ミス時やキャッシュ・ミス時のフラッシュなどによって命令が再実行された場合にのみ生じる．分岐予測ミス時は，フラッシュされた命令の物理レジスタがフリー・リストへ返却され，それが再利用されるために上書きが起きる場合がある．キャッシュ・ミス時もまた，フラッシュ後のリプレイによって同じ命令が再実行されるため，レジスタへの上書きが起きる場合がある．

これらが起きる可能性は非常に低いものの，生じた場合には動作の正しさを保つための制御を行う必要がある．そこで，ライト・ヒットを検出した際はレジスタ・キャッシュの全エントリをフラッシュする．4.5 節の評価で述べるように，通常，このライト・ヒットはほとんど全く起きないため，このような簡単な実装としても性能はほとんど全く変化しない．

4.1.3 バイパス・ネットワーク の簡略化

LORCS では，パイプライン化されたレジスタ・ファイルの場合と比較してバイパス・ネットワーク の規模を縮小することができる．これは，レジスタ・キャッシュのレイテンシが短縮されていることと，メイン・レジスタ・ファイル（ライト・バッファ）のためのバイパス・ネットワーク が必要ないことによる．

図 4.2 では，命令 I_{c_1} から I_{c_3} は I_p に依存している．これらのうち，I_{c_3} は，レジスタ・キャッシュを介して実行結果を受け取ることができる．これは，I_{c_3} のレジスタ・キャッシュ読み出しが I_p のレジスタ・キャッシュ書き込みよりも後にあるためである．同図では，I_{c_1} と I_{c_2} のみがバイパス・ネットワークを介して実行結果を受け取る必要がある．すなわち，バイパス・ネットワークは実行が行われた直後の 2 サイクル間のみ，値の供給を行えばよい．これは，1 サイクルのレイテンシを持つレジスタ・ファイルの場合と同様である．

LORCS では，メイン・レジスタ・ファイルとライト・バッファのためのバイパス・ネットワークは通常必要ない．前述したように，命令の実行結果は，RW/CW ステージにおいてレジスタ・キャッシュとメイン・レジスタ・ファイル（ライト・

バッファ）の双方に同時に書き込まれる．したがって，メイン・レジスタ・ファイルとライト・バッファのバイパスの対象となる命令の実行結果は，ほとんどの場合レジスタ・キャッシュ上にある事が期待できる．

4.1.4 メイン・レジスタ・ファイルのポート数の削減

LORCS では，レジスタ・キャッシュにミスしたオペランドのみがメイン・レジスタ・ファイルにアクセスを行うため，メイン・レジスタ・ファイルの読み出しポートを削減することができる．また，レジスタ・キャッシュとライト・バッファを組み合わせて用いることにより，追加のフォワーディング・パスを増やすことなく，メイン・レジスタ・ファイルの書き込みポートを減らす事ができる．

ライト・バッファは，メイン・レジスタ・ファイルの書き込みポートを命令の平均実行スループットにまで下げる．この手法は，レジスタの書き込みにのみ有効であり，読み出しに対しては適用できない．レジスタの書き込みは命令の実行に対してクリティカルではないため，通常のメモリにおけるストア・バッファと同様にして，書き込みを遅延させることができる．これに対し，レジスタの読み出しは命令の実行に対してクリティカルであるため，バッファを用いて読み出しを遅延させることはできないのである．

4.5 節の評価で示すように，メイン・レジスタ・ファイルには 2 read/2 write 程度のポートがあれば十分である．

このポート数の削減により，メイン・レジスタ・ファイルの回路面積は大幅に削減される．これは，メイン・レジスタ・ファイルを構成する RAM の回路面積がポート数の 2 乗に比例するためである [2, 3]．4.5 節の評価では，メイン・レジスタ・ファイルの回路面積はレジスタ・キャッシュのそれとほぼ同程度であり，フル・ポートのレジスタ・ファイルと比較して回路面積は 30.7% にまで削減される．

この回路面積の削減は，結果としてメイン・レジスタ・ファイルのレイテンシの削減につながる．メイン・レジスタ・ファイルの回路面積はレジスタ・キャッシュのそれとほぼ同程度であるため，メイン・レジスタ・ファイルもまた 1 サイクルでアクセス可能である．

4.2 LORCS におけるレジスタ・キャッシュ・ミス

Butts らが指摘したように，レジスタ・キャッシュ・ミス時のメイン・レジスタ・ファイルからの読み出しを行う現実的な方法は**ストール**と**フラッシュ**の 2 つしかない [8].

4.2.1 ストールとフラッシュ

図 4.3 に，レジスタ・キャッシュ・ミス時にストールを行うモデルによるバックエンド・パイプラインの挙動を示す．同図のモデルでは，メイン・レジスタ・ファイルのレイテンシは 1 サイクルであり，ストールによって命令の実行は 1 サイクル遅延する．

図 4.4 に，レジスタ・キャッシュ・ミス時にフラッシュを行うモデルによるバックエンド・パイプラインの挙動を示す．フラッシュを行うモデルでは，レジスタ・キャッシュ・ミスを起こした命令と同時か，それより後に発行された全ての命令をフラッシュし，再発行を行う．この場合，命令の実行は**発行レイテンシ**の分だけ遅延する．発行レイテンシは，スケジューリング・ステージからフラッシュを行う直前のステージまでのステージ数によって与えられる．これは，フラッシュされた命令はスケジューリング・ステージからリプレイされるためである．図 4.4 の場合，スケジューリング，発行，レジスタ・キャッシュ読み出しにそれぞれ 1 サイクルが割り当てられているため，発行レイテンシは $3 - 1 = 2$ サイクルとなる．

通常，メイン・レジスタ・ファイルのレイテンシはこの発行レイテンシよりも短い．このため，4.5 節で示すように，通常ストールを行うモデルの方が，フラッシュを行うモデルよりも性能はよくなる[2].

ここで，レジスタ・キャッシュ・ミス時の現実的な対処方法はストールとフラッシュの 2 つしかないことを強調しておきたい [8]．これは，**選択的ストール**や**ヒット・ミス予測**を用いることで，LORCS のペナルティを軽減できるのではないかと言う指摘をこれまでに多数受けたためである．指摘者らがこれらの手法が一見容易に実現可能であると考えるのに対し，実際には次節以降で述べるように**資源競合**による副作用によって，これらを実現することは非常に困難である．

[2]一般に，L1 データ・キャッシュではフラッシュを行ったほうが良い．L1 データ・キャッシュの場合，ストールのペナルティは L2 キャッシュのレイテンシとなるが，これはフラッシュのペナルティである発行レイテンシよりも通常ずっと長いためである．

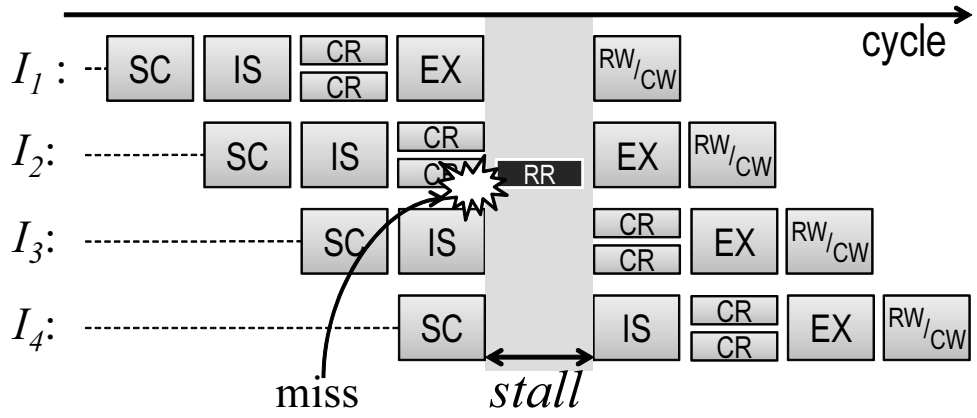


図 4.3: レジスタ・キャッシュ・ミス時の LORCS のバックエンド・パイプライン（ストール）

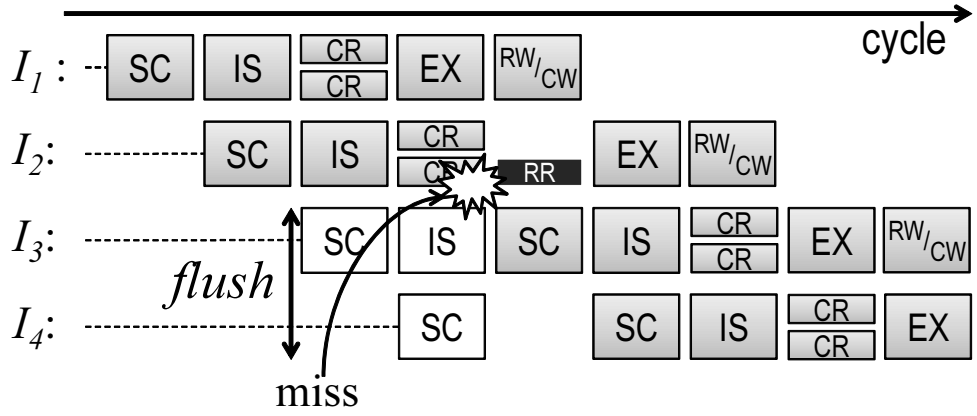


図 4.4: レジスタ・キャッシュ・ミス時の LORCS のバックエンド・パイプライン（フラッシュ）

4.2.2 選択的ストール

通常，Out-of-Order スーパスカラ・プロセッサでは長いレイテンシを持つ命令の実行を選択的に遅らせることができる．このため，レジスタ・キャッシュ・ミス時のメイン・レジスタ・ファイルのレイテンシもまた，同様にして隠蔽できると考えがちである．しかし，実際にこれを行うことは非常に難しい．これは，Out-of-Order スーパスカラ・プロセッサのバックエンド・パイプラインが命令の再スケジューリング能力を持たないためである．

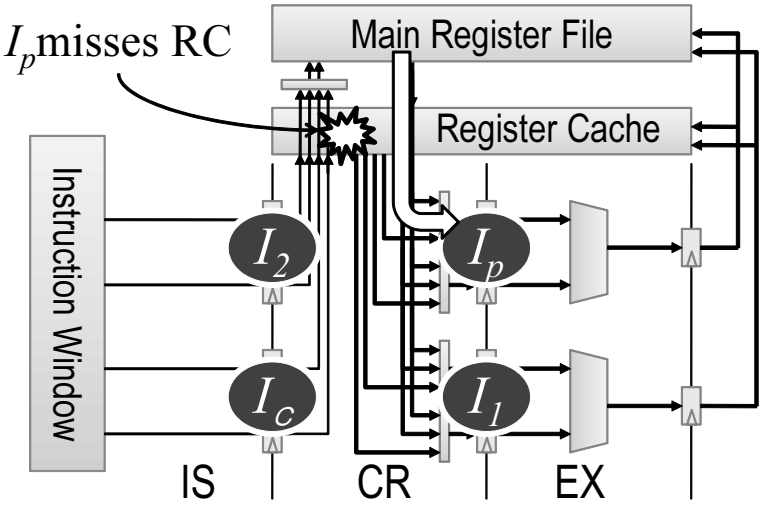
図 4.5(a) と図 4.5(b) に LORCS のブロック・ダイアグラムとパイプライン・チャートを示す．これらの図では，命令 I_p，I_1，I_2，I_c がバックエンド・パイプラインに対して発行されている．また，I_c は I_p に依存している．図 4.5(a) は，(b) 内の下向き矢印で示す瞬間のスナップ・ショットとなっており，この時 I_p はレジスタ・キャッシュ・ミスを起こしている．

図 4.5(b) だけを見た場合，I_p と I_c の**選択的ストール**を行い，後続の I_1 と I_2 の実行を継続する事が一見可能なようにみえる．これはしかし，パイプライン・チャートのような図が，資源競合を視覚的に表現することをあまり得意としていないためである．図 4.5(b) では，I_2 は同じパイプラインの**レーン**上で先行している I_p にブロックされており，実際にはこれ以上進むことができない．また，I_c は I_p に依存しているため，I_p に合わせてストールを行う必要がある．このため，もう片方のレーンも I_c によって命令がブロックされ，命令はパイプライン上を進むことができない．結果として，I_p が動き始めるまで，このパイプラインでは一切命令の発行を行うことはできないのである．この状況でストールを免れうるのは，実際には I_1 のみである．

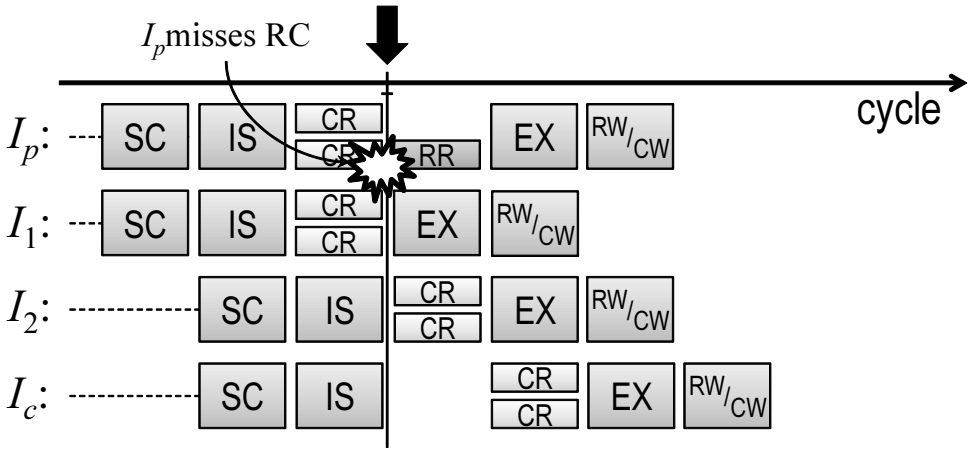
ある命令がパイプライン上を先に進むためには，

1. **その命令がキャッシュ・ミスを起こした命令に依存していない**
   事に加え，
2. **キャッシュ・ミスを起こした命令とそれに依存した命令が同じレーンの先に居ない**

ことが必要である．これを実現するためには，バックエンド・パイプライン中にある命令の中から，ミスを起こした命令に依存している命令を再帰的に毎サイクル探索する必要がある．この動作は，通常の命令スケジューリングにおいて行うウェ



(a) ブロック・ダイアグラム



(b) パイプライン・チャート

図 4.5: 選択的ストール

イクアップの動作よりも遥かに複雑であり，到底現実的ではない．

パイプラインの再スケジューリング能力の欠如

結局のところ，Out-of-Order スーパスカラ・プロセッサのバックエンドもまたパイプラインであるため，**選択的ストール**は困難なのである．パイプラインは，その中で命令の再スケジューリングを行うことはできない．命令は，パイプライン上を流れることができるよう，パイプライン間にあるバッファ（命令ウィンドウ）上で前もってスケジューリングを行っておく必要がある．

4.2.3 ヒット・ミス予測

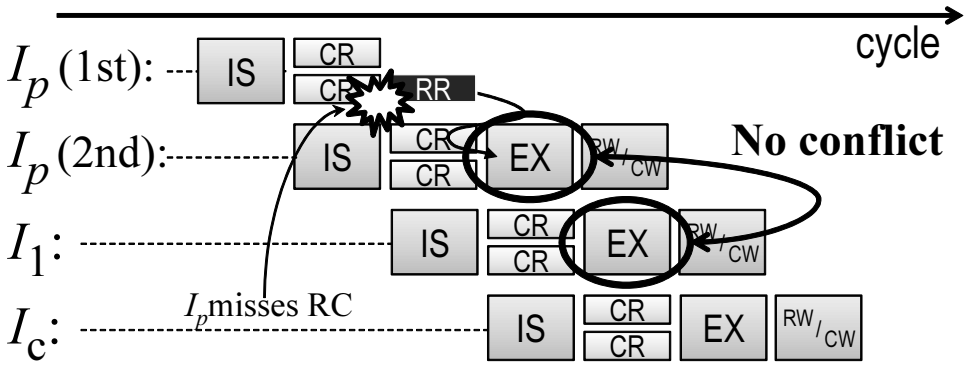
通常のデータ・キャッシュに対して**ヒット・ミス予測**を行う手法が提案されている [21]．これらの手法では，依存元命令がキャッシュ・ミスを起こすと予測した場合，依存先命令の発行をそれに合わせて遅らせる事によってパイプラインの乱れを回避している．

ヒット・ミス予測をレジスタ・キャッシュに対して適用する場合，データ・キャッシュの場合とは異なる部分がある．図 4.6(a) と図 4.6(b) にヒット・ミス予測を行った場合の LORCS のパイプライン・チャートを示す．図中の各ラベルについては，図 4.2 と同じ意味を持つ．同図の中では，命令 I_c は I_p に依存している．

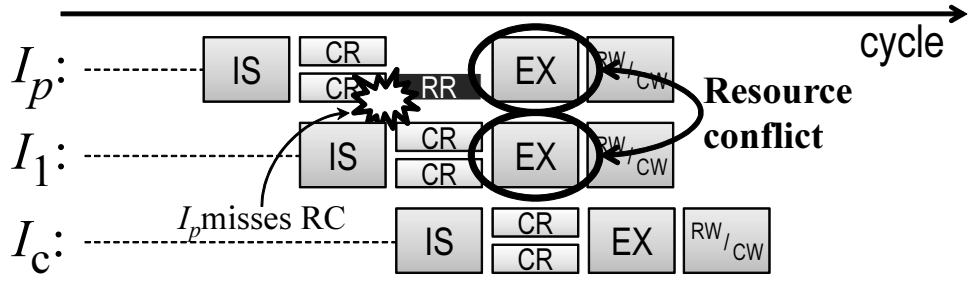
2 度の発行を行う場合

図 4.6(a) で示すように，レジスタ・キャッシュにミスすると予測された命令は，以下の様に 2 回の発行を行う必要がある．

1. **1 度目の発行:**
   I_p がレジスタ・キャッシュ・ミスを起こすと予測され，通常のタイミングで発行される．その後，I_p は予測通りレジスタ・キャッシュ・ミスを起こし，通常の場合と同様にしてメイン・レジスタ・ファイルへのアクセスを開始する．
2. **2 度目の発行:**
   I_p はメイン・レジスタ・ファイルのレイテンシの経過後，再度発行される．2 度目に発行された I_p は，実行のちょうど直前にメイン・レジスタ・ファイルから値が得られるため，それを用いて実行を行う．



(a) 2 回の発行を行った場合



(b) 2 回の発行を行わなかった場合

図 4.6: 2 回の発行を行った場合と行わなかった場合のヒット・ミス予測の動作

1 度目の発行の目的はメイン・レジスタ・ファイルへのアクセスを開始することである．メイン・レジスタ・ファイルへのアクセスを開始するためにはソース・オペランドの物理レジスタ番号が必要である．物理レジスタ番号を得る最も適当な方法は，命令ウィンドウから発行を行うことである．

2 度目の発行の目的は，得られた値を用いて実際に実行を行う事と，資源競合を回避することである．

2 度の発行を行わなかった場合

図 4.6(b) は，上記の 2 回の発行を行わなかった場合のパイプライン・チャートである．レジスタ・キャッシュ・ミスを起こすと予測された I_p は，通常のタイミングで発行され，メイン・レジスタ・ファイルのアクセスを開始する．ここまでは，図 4.6(a) の場合と同様である．その後，なんらかの方法を用いて I_p はその場でメイン・レジスタ・ファイルから値が得られるまで待つ．この場合，図で示されているように，I_p と I_1 は同じ演算器に対して資源競合を起こす（I_p と I_1 が同じタイプの演算器を使用する場合）．

これらの資源競合を回避するための最も適切な方法は命令を 2 度発行することである．図 4.6(a) では，スケジューラが I_p を 2 度目の発行のためにセレクトするため，結果として I_1 の発行が次のサイクルに持ち越される．それにより，I_p と I_1 の資源競合は回避される．

通常のデータ・キャッシュにおけるヒット・ミス予測では，この 2 回の発行を回避することができる．今，ロード命令 I_m が L1 データ・キャッシュにミスすると予測され，別のロード命令 I_h がその後続で発行されるとする．データ・キャッシュの場合，これらの I_m と I_h は同じユニットを使用しない．I_m が L2 データ・キャッシュから値を取得するのに対し，I_h は L1 データ・キャッシュから値の取得を行うためである．

これらの双方の命令によって得られた値はレジスタ・ファイルに対して書き込みを行う必要がある．このため，レジスタ・ファイルの書き込みポートについてこれらの命令間で資源競合が発生する．また，これらの値はバイパス・ネットワークを介して後続のコンシューマ命令に値を渡す必要がある．このため，バイパス・ネットワークの入力ポートに対してもやはり資源競合が発生する．これらの資源競合

は，レジスタ・ファイルの書き込みポートとバイパス・ネットワークの入力ポートをL2データ・キャッシュからの入力のために増設することによって回避することができる [21].

この方法は，しかし，レジスタ・キャッシュに対して適用することは困難である．全てのミスを起こすと予測された命令のために演算器やポートを増設することは非現実的である．

結果として，レジスタ・キャッシュにおいてヒット・ミス予測を実現する唯一の現実的な実装は，発行を2回行うことであると言える．しかし，発行幅を2倍消費することによる性能低下は大きく，4.5節で述べるように，単純にストールを行うモデルよりも性能は悪くなってしまう．

4.3 NORCS

本稿では，**非レイテンシ指向レジスタ・キャッシュ・システム**（**NORCS**：**Non-Latency-Oriented Register Cache System**）を提案する．NORCSではレジスタ・ファイルのアクセス・レイテンシを短縮しないため，そもそもIPCは向上しない．そのかわり，NORCSではLORCSで発生するパイプラインの乱れもまた発生しない．言い換えると，NORCSでは，IPCは必ずしも向上しないが，大きくも下がらないのである．NORCSはIPCを保ったまま，LORCSと同様にレジスタ・ファイルの複雑さを下げることができる．

本節では，まずこのNORCSの物理構成と命令パイプラインについて述べた後，LORCSとの違いについて述べる．

4.3.1 NORCSの物理構成

図4.7と図4.8にLORCSとNORCSのブロック・ダイアグラムを示す．これらの図では，2ステージがメイン・レジスタ・ファイルへのアクセスに対して割り当てられている．同図からもわかるように，NORCSはLORCSとほぼ同じ構成を持つ．NORCSのレジスタ・キャッシュやメイン・レジスタ・ファイルそのものは，LORCSのそれらと大きくは違わない．

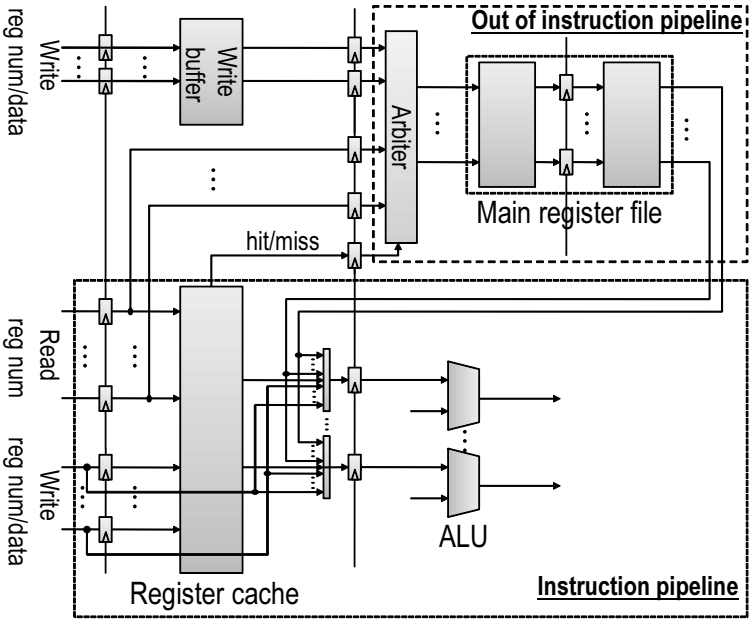


図 4.7: LORCS のブロック・ダイアグラム

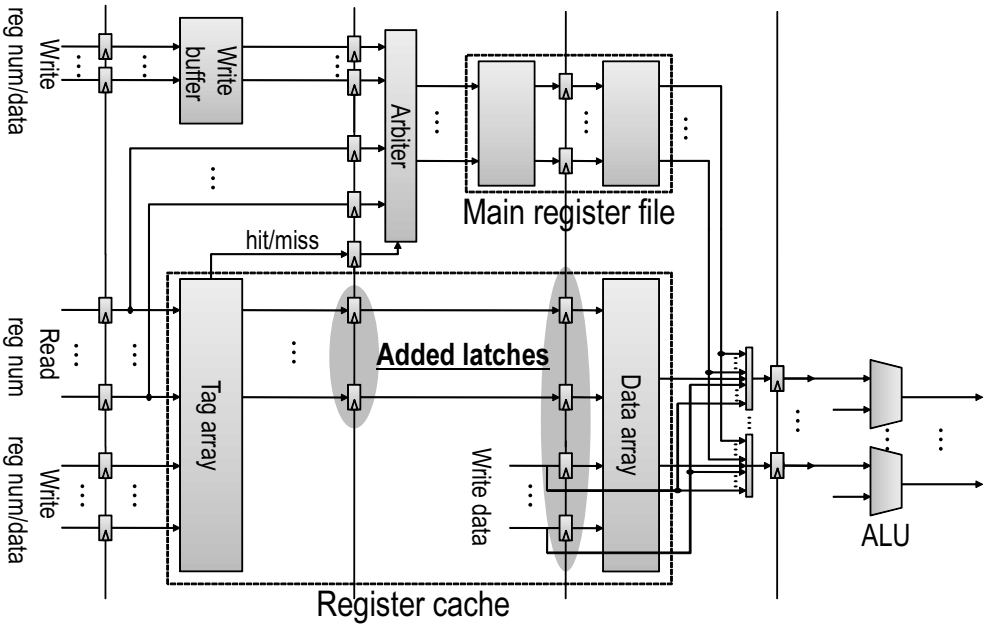


図 4.8: NORCS のブロック・ダイアグラム

図 4.8 内で強調されている，LORCS から追加されたパイプライン・ラッチ[3]は，小さいながらも最も重要な違いである．これらのラッチはレジスタ・キャッシュのタグ・アレイとデータ・アレイの間に置かれ，タグ・アレイとデータ・アレイはそれぞれ異なるステージでアクセスされる．

これらの図が示すように，NORCS と LORCS の間のロジックの違いはとても小さい．しかし，以降の節で述べるように，この小さな違いがとても大きな性能の違いを生むのである．

4.3.2 NORCS の命令パイプライン

NORCS は**ミスを仮定したパイプライン**を持つ．全ての命令は，レジスタ・キャッシュのヒット/ミスに関わらず，メイン・レジスタ・ファイルの読み出しステージを通過する．

図 4.9 と図 4.10 に LORCS と NORCS のパイプライン・チャートを示す．図 4.9 は，図 4.3 で示したものと同じものである．図 4.10 では，レジスタ・キャッシュのタグ・アレイとデータ・アレイはそれぞれ 1 サイクルのレイテンシを持つ．図中の RS（***Register Scheduling***）は，レジスタ・キャッシュのヒット・ミス判定を行うステージを示す．また，その他のラベルについては，先の図 4.2 内のものと同じである．

同図では，I_2 の**下側**のオペランドのみが，レジスタ・キャッシュ・ミスを起こしている．RR/CR ステージでは，RS ステージのヒット・ミス判定の結果に従い，メイン・レジスタ・ファイルかレジスタ・キャッシュのデータ・アレイにアクセスを行う．図中の RR/CR ステージの内，塗りつぶされているもの（I_2 の下側のオペランド）は，レジスタ・キャッシュ・ミスが発生したためにメイン・レジスタ・ファイルからの読み出しが発生していることを示す．これに対し，その他の塗りつぶされていない RR/CR については，メイン・レジスタ・ファイルはアクセスされず，その代わりにレジスタ・キャッシュのデータ・アレイから値が読み出されていることを示す．

発行されてきた命令は，RS においてレジスタ・キャッシュのタグ・アレイにアクセスを行い，ヒット/ミス判定のみを行う．この時，レジスタ・キャッシュのデータ・アレイへはアクセスしない．オペランドがレジスタ・キャッシュにヒットした

[3]図中で Added latches と書かれている部分である．

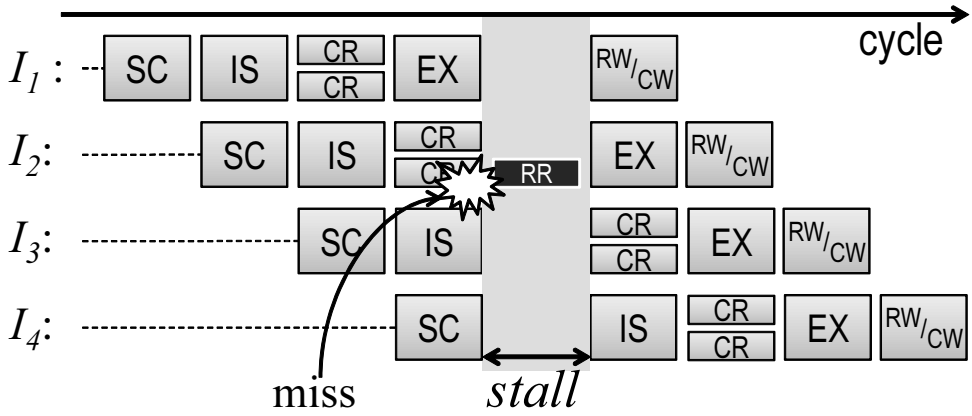


図 4.9: LORCS のバックエンド・パイプライン

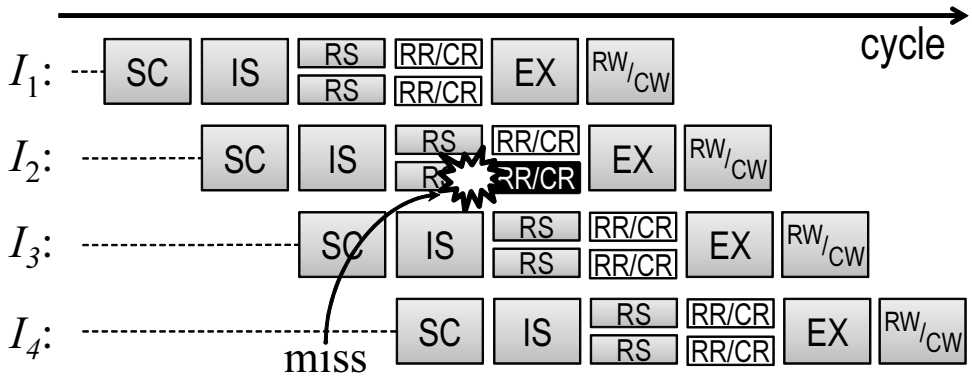


図 4.10: NORCS のバックエンド・パイプライン

場合（I_1 など），命令は単純にタグ・マッチングの結果を後続のステージに送る．実行ステージの直前にある，塗りつぶされていない RR/CR では，送信されてきたタグ・マッチングの情報を元に，レジスタ・キャッシュのデータ・アレイから値を読み出す．レジスタ・キャッシュ内にオペランドが見つからなかった場合は（I_2 の下側のオペランド），図上で塗りつぶされている RR/CR ステージに置いてメイン・レジスタ・ファイルの読み出しを行う．

パイプラインのストール

NORCS のパイプラインは，メイン・レジスタ・ファイルへのアクセスを行う RR ステージを持つ．このため，レジスタ・キャッシュ・ミスの発生のみではパイプラインは乱れない．

パイプラインは，メイン・レジスタ・ファイルの読み出しポートが不足した場合に乱れる．この読み出しポートの不足とは，メイン・レジスタ・ファイルの読み出しポートの数よりも多い数のレジスタ・キャッシュ・ミスが 1 サイクル内に発生した場合のことを指す．この場合，パイプラインをストールさせることにより，メイン・レジスタ・ファイルから値を読み出すのに必要な時間を用意しなければならない．

LORCS と NORCS のパイプラインの乱れが生じる条件をまとめると，以下の様になる．

- **LORCS:** 1 つ以上のレジスタ・キャッシュ・ミスが 1 サイクル内に発生した場合．
- **NORCS:** メイン・レジスタ・ファイルのポート数よりも多い数のレジスタ・キャッシュ・ミスが 1 サイクル内に発生した場合．

4.5 節の評価で示すように，メイン・レジスタ・ファイルの読み出しポート数の不足や，それによるパイプラインの乱れを防ぐためには，読み出しポートが 2 本ほどあれば十分である．

4.3.3 レジスタ書き込み

NORCS におけるレジスタ・キャッシュやメイン・レジスタ・ファイルへの書き込みは，基本的には 4.1.2 節で述べた LORCS の場合と同じである．メイン・レジ

スタ・ファイルへはライト・バッファを介し，ライトスルー・ポリシに基づいて書き込みを行う．レジスタの上書きが生じた場合の対処についてもLORCSの場合と同じである．

ポート共有時の書き込み

メイン・レジスタ・ファイルのポートを読み書きで共有することにより，そのポート数の合計を減らす事ができる．この場合，NORCSではライト・バッファがより重要となる．

4.1.2節で述べたように，LORCSのライト・バッファでは，メイン・レジスタ・ファイルの書き込みのために必要なポート数を，命令の平均実効スループットに比例した数まで減らすことができる．NORCSではこれに加え，読み出しのために書き込みを遅らせる効果が重要となる．

LORCSのメイン・レジスタ・ファイルからの読み出しは，バックエンド・パイプラインをストールさせることによって作った追加の時間中に行われる．このため，ポートを読み書きで共有したとしても，通常は読み出しと書き込みが同時に起きることはない．この場合のライト・バッファの効果は，書き込みのために必要なポート数を減らすことである．

NORCSのメイン・レジスタ・ファイルからの読み出しは，基本的にはバックエンド・パイプラインのストールを伴わない．このためLORCSとは異なり，メイン・レジスタ・ファイルからの読み出しと書き込みは同時に起きる．この場合，書き込みよりも読み出しを優先して行う方がよい．読み出しはクリティカルであるのに対し，書き込みは遅らせる事ができるためである．NORCSではこのため，ライト・バッファは書き込みポート数を減らすことに加え，読み出しを優先するために書き込みを遅らせる効果を持つ．

4.3.4 バイパス・ネットワークの簡略化

4.1節で述べたように，LORCSはバイパスの簡略化を行うことができる．この簡略化は，レジスタ・キャッシュがヒットし続ける限りにおいて，LORCSは1サイクルでアクセス可能なレジスタ・ファイルを持つシステムと等価になるためである．

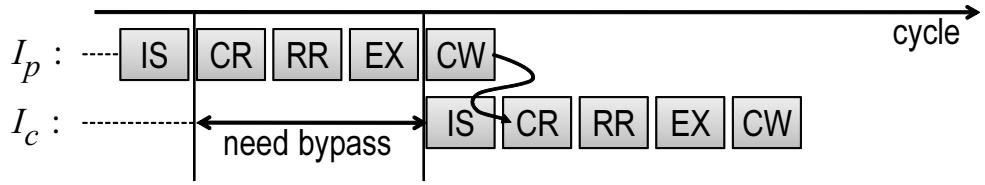


図 4.11: NORCS のバイパス (ナイーブな実装の場合)

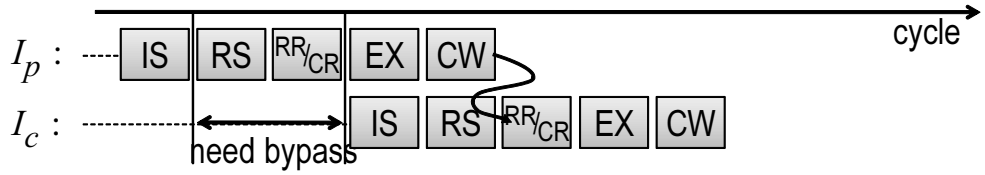


図 4.12: NORCS のバイパス

NORCS は，LORCS と同じ方法ではバイパス・ネットワークの簡略化を行うことができない．NORCS ではレジスタ・ファイルのレイテンシを短縮していないからである．このため，NORCS では，LORCS とは異なる方法によってこれを実現する．

NORCS では，レジスタ・キャッシュのデータ・アレイのアクセスをメイン・レジスタ・ファイルのアクセス・ステージの最後にまで遅らせる事によってバイパス・ネットワークの簡略化を行う．以下ではこの簡略化について，ナイーブに NORCS を実装した場合と比べながら説明する．

ナイーブな実装の場合

通常のキャッシュの実装では，高速化のため，タグ・アレイとデータ・アレイを並列にアクセスする．図 4.11 は，NORCS をそのようにしてナイーブに実装した場合のパイプライン・チャートである．同図では CR ステージにおいて，レジスタ・キャッシュのタグ・アレイとデータ・アレイは並列にアクセスされる．図中のその他のラベルについては図 4.11 のものと同じである．また，図中の I_c は，I_p に依存している．

I_c は，レジスタ・キャッシュを介して I_p の実行結果を受け取る．これは，I_c の CR ステージが I_p の CW ステージよりも後にあるためである．もし，I_c がこれよ

りも早いサイクルで発行された場合，バイパス・ネットワークを介して I_p の実行結果を受け取らなければならない．結果として，同図の場合では，バイパス・ネットワークは I_p の実行結果をその発行後 3 サイクル間供給し続ける必要がある．

提案手法の場合

図 4.12 は，NORCS の提案手法による実装を行った場合のパイプライン・チャートである．同図では，レジスタ・キャッシュのデータ・アレイ・アクセスを，メイン・レジスタ・ファイルのアクセス・ステージの最後にまで遅らせて行う．実行ステージの直前にある RR/CR ステージでは，RS ステージで行われたタグ・マッチの結果に従い，レジスタ・キャッシュのデータ・アレイにアクセスを行う．

同図の場合，バイパス・ネットワークは I_p の実行結果をその発行後 2 サイクル間のみ供給すればよい．I_c の RR/CR ステージが I_p の CW ステージよりも後にあるためである．これは，先に示したナイーブな実装の場合よりも 1 サイクル短く，1 サイクルのレイテンシを持つレジスタ・ファイルのバイパス・ネットワークの場合と同じである．

4.4 NORCS の考慮

レイテンシの短縮を行わないキャッシュのアイデアは，一見奇妙に映ることがある．本節では，読者が疑問に思いがちな点に焦点をおいて，NORCS の議論を行う．

レイテンシの短縮を行わないキャッシュ・システムは，レジスタ・キャッシュのみで働き，通常のキャッシュでは全く意味がない．4.4.1 節ではこの点について説明を行う．

LORCS よりも NORCS の方が良い性能を示すことは，より難解である．単純にメイン・レジスタ・ファイルのレイテンシのみに着目すると，

- LORCS では，レジスタ・キャッシュにミスした命令のみがメイン・レジスタ・ファイルのレイテンシを待つのに対し，
- NORCS ではヒット/ミスに関わらず常にメイン・レジスタ・ファイルのレイテンシを待つ

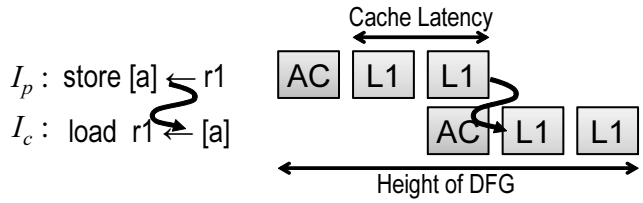
ためである．4.4.2 節では，両者の比較を行いながら，NORCS の方が速い理由について述べる．

4.4.1 レイテンシを短縮しないキャッシュ・システム

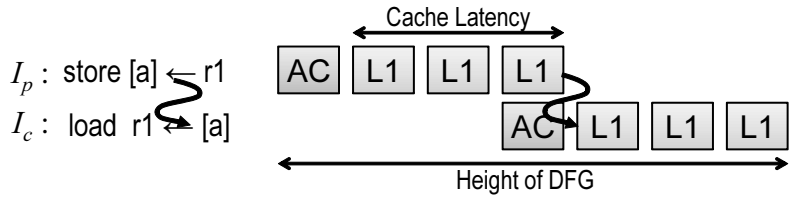
レイテンシを短縮しないパイプラインはレジスタ・キャッシュでのみ働き，通常のデータ・キャッシュでは全く無意味である．以下ではこの点について，データの依存関係と**データ・フロー・グラフ**（**DFG** : **Data Flow Graph**）の高さに着目して説明を行う．レイテンシの長さがデータ・フロー・グラフの高さに影響を与えないのであれば，レイテンシを短縮しないキャッシュ・システムは有効に働くと言える．

データの依存関係は，“***value–value***” 依存 と “***via–index***” 依存 に分ける事ができる．データ・キャッシュの場合，*index* とはメモリのアドレスの事を指す．また，レジスタ・キャッシュの場合，*index* はレジスタ番号を指す．これらの依存関係とデータ・フロー・グラフの高さは，以下の様にまとめられる．

- **データ・キャッシュ**
  - ***Value–value***:
    Value–value 依存は，ストア命令と，それに依存するロード命令の間にある依存関係である．この依存関係にある命令間では，レイテンシの増加はデータ・フロー・グラフの高さの増加につながる．データ・キャッシュは通常，値のバイパスを行う事ができないからである．図 4.13(a) と図 4.13(b) にデータ・キャッシュの場合の *Value–value* 依存の例を示す．それぞれの図は，キャッシュのレイテンシが 2 サイクルの場合と 3 サイクルの場合についてのものである．各図では，同一のメモリ・アドレス a に対するストア命令 I_p とロード命令 I_c がこの依存関係にある．これらの図では I_p によるキャッシュへの書き込みが終了するまで I_c を開始することができない．このため，図 4.13(a) と図 4.13(b) では，キャッシュのレイテンシが 1 サイクル増加することにより，その分データ・フロー・グラフの高さが伸びている．
  - ***Via–index***:
    Via–index 依存は，ロード命令と，それに依存する任意の種類の命令の間にある依存関係である．この場合でも，レイテンシの増加はデータ・フロー・グラフの高さを増加させる．リンクド・リストにおけるポインタ・チェイシングは *Via–index* 依存の典型的な例である．図 4.15 に，このポインタ・チェイシングに含まれる，ポインタの更新コードを取りだした物を示す．同図では，ロード命令 I_p と，それに依存したロード命令 I_c がこの依存関係にある．I_c は，I_p の結果が得られるまでメモリ・アクセスを開始できないため，

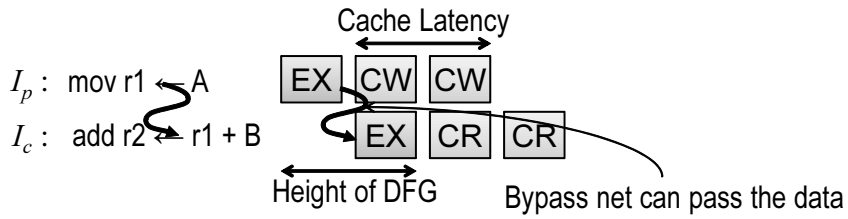


(a) レイテンシが 2 サイクルの場合

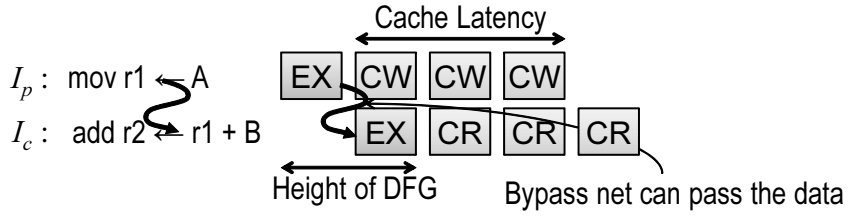


(b) レイテンシが 3 サイクルの場合

図 4.13: データ・キャッシュにおける *Value–value* 依存



(a) レイテンシが 2 サイクルの場合



(b) レイテンシが 3 サイクルの場合

図 4.14: レジスタ・キャッシュにおける *Value–value* 依存

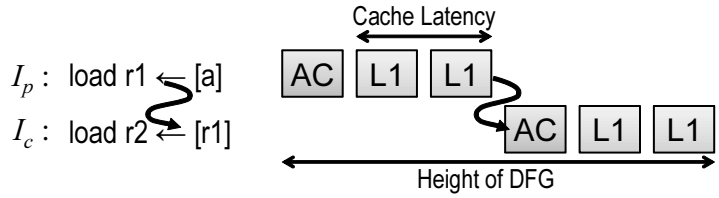


図 4.15: データ・キャッシュにおける *Via–index* 依存

キャッシュのレイテンシが伸びるとデータ・フロー・グラフの高さも伸びてしまう.

- **Register Cache**
  - ***Value–value***:
    実行結果はバイパス・ネットワークによって即座に渡されるため，レジスタ・キャッシュのレイテンシはデータ・フロー・グラフの高さとは無関係である. 図 4.14(a) と図 4.14(b) にレジスタ・キャッシュの場合の *Value–value* 依存の例を示す. それぞれの図は，レジスタ・キャッシュのレイテンシが 2 サイクルの場合と 3 サイクルの場合についてのものである. 各図では，同一のレジスタ $r1$ を介して依存している I_p と I_c がこの依存関係にある. これらの図では，I_p の実行結果はバイパス・ネットワークを介して次のサイクルに I_c に渡されるため，レジスタ・キャッシュのレイテンシが伸びてもデータ・フロー・グラフの高さは伸びていない.
  - ***Via–index***
    index である命令のレジスタ番号は命令のデコード/リネーム時に静的に決定されているため，他の命令の結果に依存して変化することはない. したがって，レジスタ・キャッシュの場合，*value–index* 依存はそもそも存在しない.

以上で述べたように，データ・キャッシュの場合ではレイテンシはデータ・フロー・グラフの高さに影響を与えるが，レジスタ・キャッシュの場合ではレイテンシはデータ・フロー・グラフの高さに影響を与えない. このため，レイテンシを短縮しないキャッシュ・システムはレジスタ・キャッシュにのみ有効なのである.

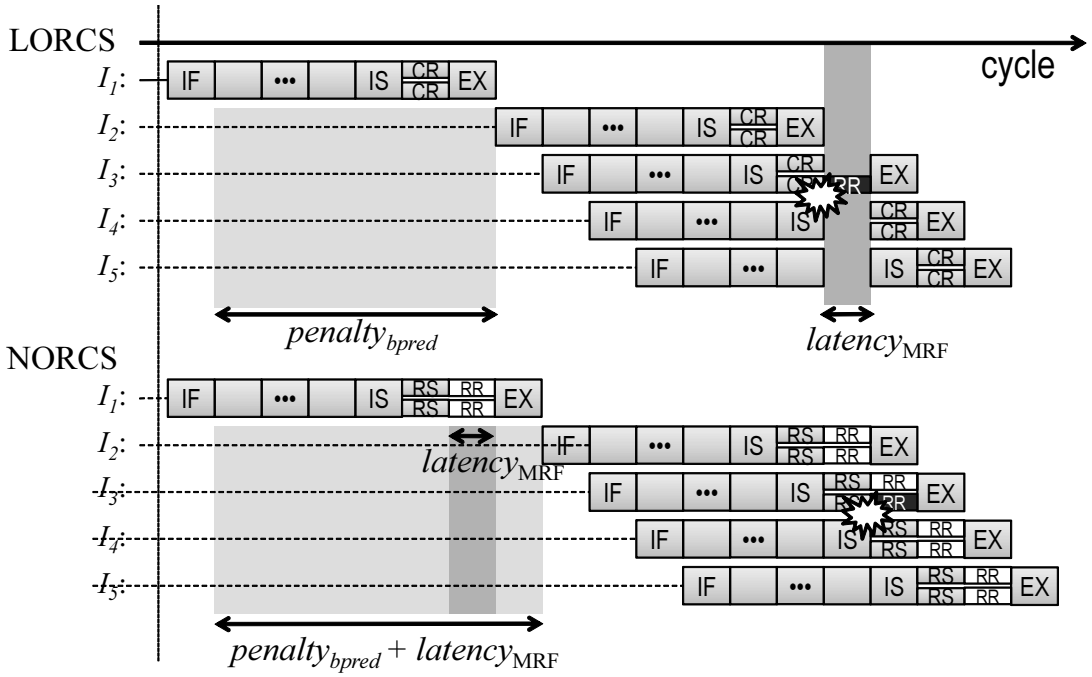


図 4.16: LORCS と NORCS のパイプライン

4.4.2 ミスを仮定したパイプライン

LORCS ではレジスタ・キャッシュにミスした命令のみがメイン・レジスタ・ファイルのレイテンシを待つのに対し，NORCS では全ての命令がメイン・レジスタ・ファイルのレイテンシを待つ．それにも関わらず，NORCS は LORCS よりも良い性能を示す．この理由は，パイプラインには再スケジューリングがない事による．

投機を行う近代的な命令パイプラインでは，レイテンシの増加は，ミス・ペナルティを介してのみ間接的に IPC を下げる．もしミス・ペナルティやパイプラインの乱れが一切ない場合は，レイテンシを伸ばしても性能は一切低下しない．

以降では簡単のため，分岐予測ミスのみを取り挙げて説明を行うが，他の予測ミスについても同じことが成り立つ．

分岐予測ミスが起きた場合，実行サイクル数は（おおよそ）分岐予測ミス・ペナルティの分だけ伸びる．レジスタ・キャッシュ・ミスが起きた場合も同様に，実行サイクル数は（おおよそ）レジスタ・キャッシュのミス・ペナルティの分だけ伸びる．

LORCS のミス・ペナルティの合計は

$$penalty_{bpred} \times \beta_{bpred} + latency_{\mathrm{MRF}} \times \beta_{\mathrm{RC}}, \tag{4.1}$$

となる．ここで，$penalty_{bpred}$ は分岐予測ミス・ペナルティ，$latency_{\mathrm{RC}}$ と $latency_{\mathrm{MRF}}$ はレジスタ・キャッシュとメイン・レジスタ・ファイルのレイテンシをそれぞれ表す．β_{bpred} と β_{RC} は分岐予測とレジスタ・キャッシュの**実効ミス率**をそれぞれ表す[4]．なお，β_{bpred} は分岐命令毎の“分岐予測ミス率”**ではない**事を注意しておきたい．β_{bpred}，すなわち実効ミス率は，各サイクルごとに分岐予測ミスが発生する確率である．これは，β_{RC} についても同様である．

図 4.16 の上半分は LORCS のレジスタ・キャッシュ・ミス時の挙動を表す．図中のステージを表すラベルについては図 4.2 や 図 4.10 中のものと同じである．同図では，I_3 の**下側**のオペランドがレジスタ・キャッシュにミスを起こしている．図 4.16 の上半分で影となっている部分のうち，薄い影は式 4.1 中の $latency_{\mathrm{MRF}}$，濃い影は同式中の $penalty_{bpred}$ をそれぞれ表す．

NORCS の分岐予測ミス・ペナルティは LORCS のそれよりもメイン・レジスタ・ファイルのレイテンシの分だけ長い．このため，NORCS のミス・ペナルティの合計は，

$$(penalty_{bpred} + latency_{\mathrm{MRF}}) \times \beta_{bpred}. \tag{4.2}$$

となる．ここで，NORCS においては，メイン・レジスタ・ファイルのポート数が不足した際に起きるパイプラインの乱れの影響は，とても小さい事を注記しておく．

図 4.16 の下半分において，薄い影は式 4.2 中の $latency_{\mathrm{MRF}}$，それに囲まれている濃い影は同式中の $penalty_{bpred}$ をそれぞれ表す．これらの結果，LORCS と NORCS の合計ミス・ペナルティの差は

$$\begin{aligned}(4.1) - (4.2) &= (penalty_{bpred} \times \beta_{bpred} + latency_{\mathrm{MRF}} \times \beta_{\mathrm{RC}}) - \\ &\quad ((penalty_{bpred} + latency_{\mathrm{MRF}}) \times \beta_{bpred}) \\ &= latency_{\mathrm{MRF}} \times \beta_{\mathrm{RC}} - latency_{\mathrm{MRF}} \times \beta_{bpred} \\ &= latency_{\mathrm{MRF}}(\beta_{\mathrm{RC}} - \beta_{bpred}).\end{aligned}$$

となる．したがって，$\beta_{\mathrm{RC}} > \beta_{bpred}$ の場合に NORCS は LORCS よりも良い性能を示す．この式が意味するところは，NORCS は LORCS に対し，

[4]フラッシュを行うモデルでは，レジスタ・キャッシュ・ミス・ペナルティは発行のレイテンシとなる．これは通常 $latency_{\mathrm{MRF}}$ よりも数サイクル程度長い．

1. レジスタ・キャッシュのミス・ペナルティ（$latency_{\mathrm{MRF}}$）を
2. 分岐予測ミス・ペナルティ

に転化している，と言うことである．

4 節で述べた様に，レジスタ・キャッシュの実効ミス率（β_{RC}）は，オペランドのアクセス毎におけるレジスタ・キャッシュのミス率そのものよりも遥かに高い．たとえば SPEC CPU 2006 の 456.hmmer では，β_{RC} は 0.139 にもなる．これに対し，分岐予測の実効ミス率（β_{bpred}）は，いわゆる分岐命令毎の “分岐予測ミス率” を分岐命令の平均ラン・レングスで割ったもので与えられる．このため，β_{RC} は通常 0.01 よりも小さい程度である．これらの結果，レジスタ・キャッシュの実効ミス率（β_{RC}）は分岐予測の実効ミス率（β_{bpred}）よりも遥かに高いものとなる．NORCS が LORCS よりも良い性能を示すのは，このためである．

4.5 評価

4.5.1 評価環境

鬼斬弐[44] に LORCS と NORCS を実装し，評価を行った．3.4.1 節で述べたように，鬼斬弐はサイクル・アキュレートなプロセッサ・シミュレータである．

評価では SPEC CPU2006 [58] に含まれる全 29 本のベンチマーク・プログラムを用いた．ベンチマーク・プログラムのコンパイルには gcc 4.2.2 を用い，コンパイル・オプションには “−O3”を指定した．入力データ・セットには ref を用い，プログラムの先頭 1 G 命令をスキップして，続く 100 M 命令の評価を行った．

表 4.1 に，シミュレーションを行った際の構成を示す．左側の列にある “Baseline” がベースラインとなるプロセッサの構成である．この構成は MIPS R 10000 Out-of-Order スーパスカラ・プロセッサ と同様の 6 命令同時発行可能なプロセッサをベースとしている．また，一部分岐予測器などについては，最近のプロセッサのものに合わせている．このベースラインとなるプロセッサに LORCS と NORCS を実装して評価を行った．

4.5.1.1 評価モデル

以下のモデルについて評価を行った．

表 4.1: シミュレーション環境

Name	Baseline	Ultra-wide
ISA	Alpha	←
pipeline stages	fetch:3, rename:2, dispatch:2, issue:2	fetch:4, rename:5, dispatch:2, issue:1
fetch width	4 inst.	8 inst.
execution unit	int:2, fp:2, mem:2.	int:6, fp:4, mem:2.
instruction window	int:32 entries, fp:16 entries, mem:16 entries	unified:128 entries
ROB	128 entries	512 entries
branch predictor	8 KB g-share	16 KB g-share
branch miss penalty	11∼12 cycles	14∼15 cycles
BTB	2 K entries, 4 way	4 K entries, 4 way
RAS	8entries	64entries
L1C	32 KB, 4 way set assoc, 64 B/line, 3 cycles	←
L2C	4 MB, 8 way set assoc, 64 B/line, 10 cycles	←
main memory	200 cycles	←

PRF:

評価のベースラインとなる，パイプライン化されたレジスタ・ファイルを持つモデル．このモデルでは完全バイパス・ネットワークを備える．

PRF-IB:

パイプライン化されたレジスタ・ファイルを持つモデル．PRF モデルとの違いは，このモデルではインコンプリート・バイパスを行うことである．命令の実行結果は，実行後の 2 サイクル間のみバイパス・ネットワークに送られる．それよりも後に，その実行結果を得ようとした場合，レジスタ・ファイルから値が得られるようになるまでバックエンド・パイプラインをストールさせる [56]．このモデルによるバイパス・ネットワークは，LORCS や NORCS，1 サイクルでアクセス可能なレジスタ・ファイルのためのバイパス・ネット

ワークと同じ規模を持つ．

NORCS:

NORCS を実装したモデル．レジスタ・キャッシュは，4，8，16，32，64 エントリのものと “infinite” エントリのものについて評価を行った．ここで “infinite” モデルとは，レジスタ・ファイルと同じ容量を持ったレジスタ・キャッシュを備えたモデルである．このモデルでは，レジスタ・キャッシュ・ミスは一切発生しない．レジスタ・キャッシュのリプレースメント・ポリシについては，LRU，USE-B，POPT について実装を行い，評価した．USE-B と POPT 置き換えについては後で述べる．

LORCS:

LORCS を実装したモデル．レジスタ・キャッシュそのものの構成については，NORCS の評価に用いたものと同じである．レジスタ・キャッシュ・ミス時の挙動については，ストールに加えフラッシュなどの複数のモデルを実装し，評価した．これらについては後で詳しく述べる．

表 4.2 内の左側の列にある “Baseline” に，各モデルのその他のレジスタ・ファイルに関する構成をまとめる．表内の “PRF”，“RC”，“MRF’ は，それぞれパイプライン化されたレジスタ・ファイル，レジスタ・キャッシュ，メイン・レジスタ・ファイルを表す．

4.5.1.2 レジスタ・キャッシュのリプレースメント・ポリシ

レジスタ・キャッシュのリプレースメント・ポリシとして，LRU に加えて **USE-B** モデルの評価を行った．このモデルは，Butts らの提案した *use-based* 置き換えアルゴリズムを実装したものである [8]．Butts らは *Use-predictor* と呼ぶ，レジスタの参照回数を予測する機構により，この置き換えアルゴリズムを実装している [59]．レジスタ・キャッシュの各エントリは *use-counter* と呼ぶカウンタを持ち，それらは Use-predictor によって予測された参照回数により初期化される．各カウンタは参照のたびにデクリメントされ，リプレース時には，最も小さなカウンタ値を持つエントリが置き換えの対象となる．評価で用いた Use-predictor のパラメータについては，表 4.2 に示す通りである．これらのパラメータについては，Butts らが評価で用いたものと同じものである．

表 4.2: レジスタ・ファイル・システムの構成

Name	Baseline	Ultra-wide
PRF latency	2 cycles	←
PRF capacity	int:128, fp:128	int:512, fp:512
PRF ports	12 ports	24 ports
RC latency	1 cycle	←
RC capacity	4, 8, 16, 32, 64 entries	16, 32, 64 entries
RC associativity	full	2-way
MRF latency	1 cycle	←
MRF capacity	int:128, fp:128	int:512, fp:512
MRF ports	read:1-3	read:4
	write:1-3 ports	write:4 ports
write buffer	8 entries	←
use predictor	4 K entries, 4 way,	←
	4 bits/pred,	
	2 bits/conf,	
	6 bits/tag,	
	6 bits/future control	

Butts らは，セット・アソシアティブ構成を取る レジスタ・キャッシュのために，Use-predictor を用いたインデクシングについても提案している．通常のレジスタ・キャッシュのインデクシングでは，物理レジスタ番号の下位ビットがインデックスとして用いられる．これに対し，Butts らの提案した手法では，インデックスは物理レジスタ番号から独立しており，大きなカウンタ値を持つエントリを上書きしないようにインデクスが割り当てられる．以降の評価では，**USE-B** モデルのレジスタ・キャッシュのうち，セット・アソシアティブ構成を取るものについてはこのインデクシングも合わせて実装して評価を行っている．

4.5.1.3 LORCS のミス時のモデル

LORCS のミス時のモデルとして，バックエンド・パイプラインのストールを行う STALL モデルと，フラッシュを行う FLUSH モデルを実装して評価した．これ

らの動作については，4.2.1 節で述べた通りである．また，STALL と FLUSH に加え，以下の理想化されたモデルについても評価を行った．

SELECTIVE-FLUSH:

このモデルでは，レジスタ・キャッシュにミスした命令と，それに依存した命令のみをフラッシュし，リプレイを行う．このモデルでは，4.2.2 節で述べたのと同じ理由により，ミスを起こした命令に依存する命令の再帰的な探索が必要である．4.2.2 節で述べたように，このような再帰的な探索を実際に実装することは非常に困難である．

このモデルでは，FLUSH とは異なり，ミスを起こした命令に依存した命令だけでなく，ミスを起こした命令そのもののフラッシュも必要である．これは，ミスを起こした命令そのものと，フラッシュされなかった命令間での資源競合を避けるためである．FLUSH の場合，ミスを起こした命令の後続にある命令は全てフラッシュされるため，このような資源競合は発生しない．

PRED-PERFECT:

このモデルは，4.2.3 節で述べたヒット・ミス予測を実装したモデルである．このモデルは非常に理想化されており，ヒット・ミス予測は常にヒットする．レジスタ・キャッシュ・ミスによって生じるパイプラインの乱れは，このモデルでは全く発生しない．したがって，“infinite” なレジスタ・キャッシュと比較した場合の性能低下は，レジスタ・キャッシュ・ミス時に発行幅が 2 倍消費されることと，レジスタ・キャッシュ・ミスによって実行そのものが遅れることのみによる（4.2 節）．

4.4.2 節 で述べたように，NORCS と LORCS は以下の同じ効果を持つ．

1. レジスタ・ファイルの回路面積の縮小と，それによる消費電力の削減
2. バイパス・ネットワークの簡略化

しかし，これに対し，両者の性能（IPC）は大きく異なる．したがって，各モデルにおける性能あたりの回路面積や消費電力を比較することが重要となる．以下ではまず，各モデルの性能や回路面積，消費電力について個別に示した後，それらの

関連について述べる．

4.5.2 レジスタ・キャッシュ・ヒット率と置き換えアルゴリズム

本節では，LRU や USE-B による置き換えに加え，**疑似 OPT 置き換え**を用いた場合の評価についても述べる．

OPT 置き換えは理想的な置き換えアルゴリズムであり，最も良い性能を示すアルゴリズムとして知られている [60]．このアルゴリズムでは，最も遠い将来まで参照されないエントリを置き換え対象として選択する．この OPT 置き換えを実際に実装することは現実的に困難であるため，本節では以下で述べる疑似 OPT 置き換えを用いる．疑似 OPT 置き換えでは，その時点でのインフライト命令の中で，最も将来まで参照されないエントリを置き換え対象として選択する．

図 4.17 に，LORCS の全ベンチマークにおける平均レジスタ・キャッシュ・ヒット率を示す．図中の POPT は疑似 OPT 置き換えを示す．同図では，メイン・レジスタ・ファイルのポート数は読み書き共に 2 ポートに固定し，レジスタ・キャッシュ・ミス時の挙動については STALL モデルに固定している．これは，レジスタ・キャッシュ・ヒット率は，レジスタ・キャッシュのエントリ数と置き換えアルゴリズムにのみ強く影響をうけるためである．

NORCS についてもレジスタ・キャッシュ・ヒット率の評価を行ったが，本稿では LORCS の結果についてのみ示す．これはヒット率について，NORCS と LORCS の間でほとんど結果に差がなかったためである．

グラフからは USE-B モデルが特に高いヒット率を示していることがわかる．USE-B モデルは，LRU モデルと比較して 3–4% 程度高いヒット率を示しており，理想的な POPT モデルに近いヒット率を達成している．

以降では，NORCS の評価では LRU モデルを用い，LORCS の評価では USE-B モデルを用いて行う．NORCS はヒット率が大幅に低い LRU 置き換えを用いるにも関わらず，USE-B 置き換えを用いる LORCS よりもずっと良い性能を示す．

4.5.3 LORCS の構成

LORCS は NORCS と比較して，より多くのパラメータを取る余地がある．このため，本節では評価対象となる LORCS の，探索空間の絞り込みを行う．

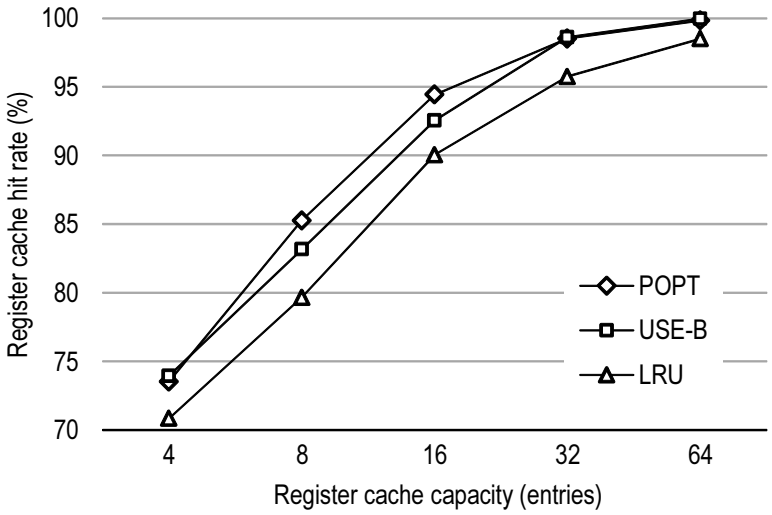


図 4.17: LORCS のレジスタ・キャッシュ・ヒット率

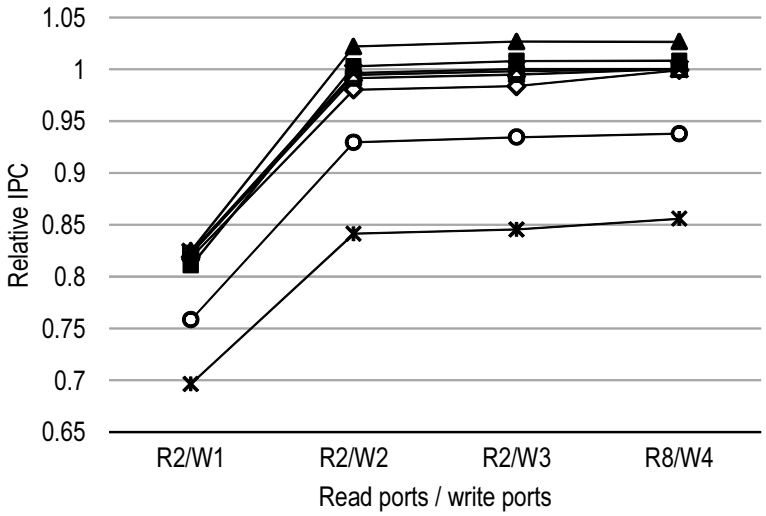
4.5.3.1 メイン・レジスタ・ファイルのポート数

まず，メイン・レジスタ・ファイルのポート数を変化させながら LORCS と NORCS の IPC を評価した．図 4.18(a) と 図 4.18(b) に，8 read /4 write のフルポートを備えたレジスタ・ファイルに対する平均相対 IPC を示す.

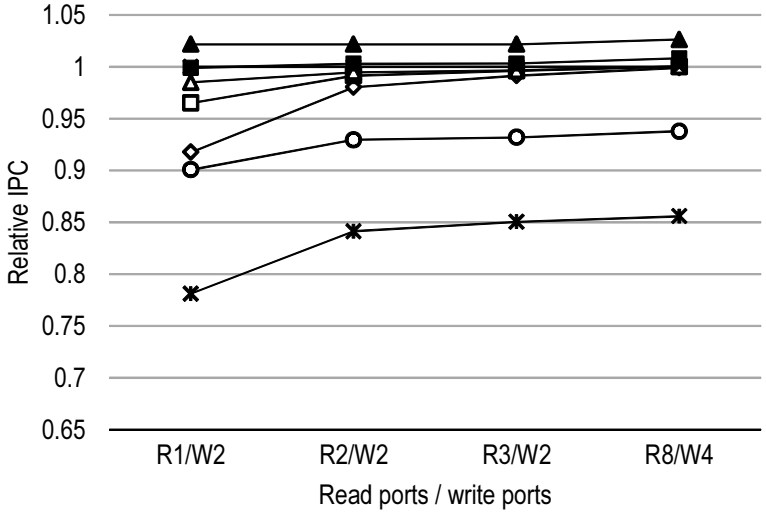
図 4.18(a) は，メイン・レジスタ・ファイルの読み出しポート数を 2 に固定して，書き込みポート数を変化させた場合の IPC である．これに対し，図 4.18(b) は逆に書き込みポートを 2 に固定して，読み出しポート数を変化させた場合の IPC である.

これらのグラフでは，8，16，32，“infinite” エントリのレジスタ・キャッシュの結果についてのみ示している．これは，4 エントリと 64 エントリレジスタ・キャッシュの結果が，これらの結果とほとんど同じ傾向を示していたためである.

これらのグラフは，フルポートに近い性能を維持するためには，メイン・レジスタ・ファイルには読み書きにそれぞれ 2 ポート程度があれば十分であることを示している．そこで，以降では，メイン・レジスタ・ファイルのポート数を読み書きそれぞれ 2 ポートに固定した上で評価を行う.



(a) 読み出しポートを 2 に固定した場合



(b) 書き込みポートを 2 に固定した場合

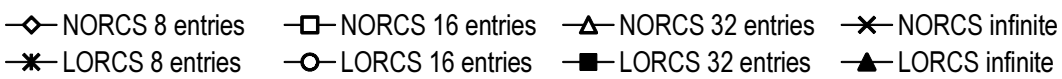


図 4.18: メイン・レジスタ・ファイルのポート数を固定した場合の相対平均 IPC

4.5.3.2 LORCS のレジスタ・キャッシュ・ミス時の挙動

図 4.19 に，レジスタ・キャッシュ・ミス時のモデル毎による平均相対 IPC を示す．このグラフではレジスタ・キャッシュのエントリ数を変化させている．各モデルの IPC は “infinite” モデルの IPC に対する相対値である．

4.2 節 で述べたように，グラフでは実際に FLUSH モデルが最も悪い性能を示している．これに対し，STALL モデルは，理想的なモデルである SELECTIVE-FLUSH モデルや PRED-PERFECT モデルに近い性能を示している．

SELECTIVE-FLUSH モデルでは，ミスを起こした命令と，それに依存した命令のみをフラッシュしている．しかしこのモデルではフラッシュとリプレイを行うため，4.2 節で述べた様に，レジスタ・キャッシュ・ミスによるペナルティは STALL モデルよりも非常に大きい．このために，両者では大きな性能の違いは現れない．

PRED-PERFECT モデルはヒット・ミス予測が完全にヒットする理想的なモデルである．しかし，4.2.3 節で述べたように，このモデルではレジスタ・キャッシュ・ミスを起こした命令を 2 回発行する必要があり，発行幅を 2 倍消費してしまう．このため，PRED-PERFECT モデルでもまた，STALL と大きな性能の違いは現れない．

STALL モデルは，理想的なモデルに近い性能を示す上，実装も最も単純である．そこで，以降では，レジスタ・キャッシュ・ミス時の挙動のモデルを STALL モデルに固定した上で，評価を行う．

4.5.4 IPC

図 4.20 に，各モデルの PRF モデルに対する平均相対 IPC を示す．ここでは，

1. PRF-IB，
2. LRU と USE-B 置き換えによる LORCS
3. LRU 置き換えによる NORCS

の評価を行った．グラフでは，8，16，32，“infinite” エントリのレジスタ・キャッシュについてのみ，評価結果を載せている．これは 4 エントリと 64 エントリの結果も，これらとほぼ同じ傾向を示していたためである．

グラフ中の “min”，“max”，“average” の各棒は，それぞれ全ベンチマークにおける最小，最大，平均の結果を示す．その他の “456.hmmer” などの棒については，特

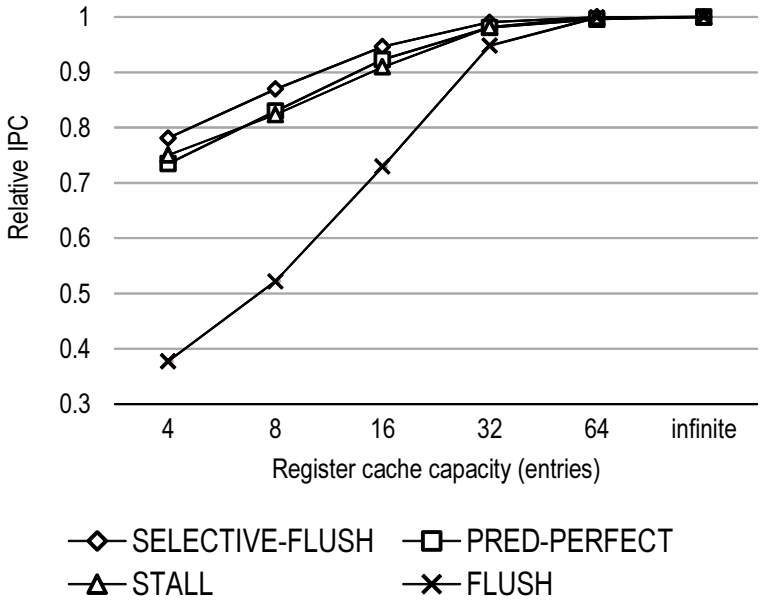


図 4.19: LORCS（USE-B）の平均相対 IPC

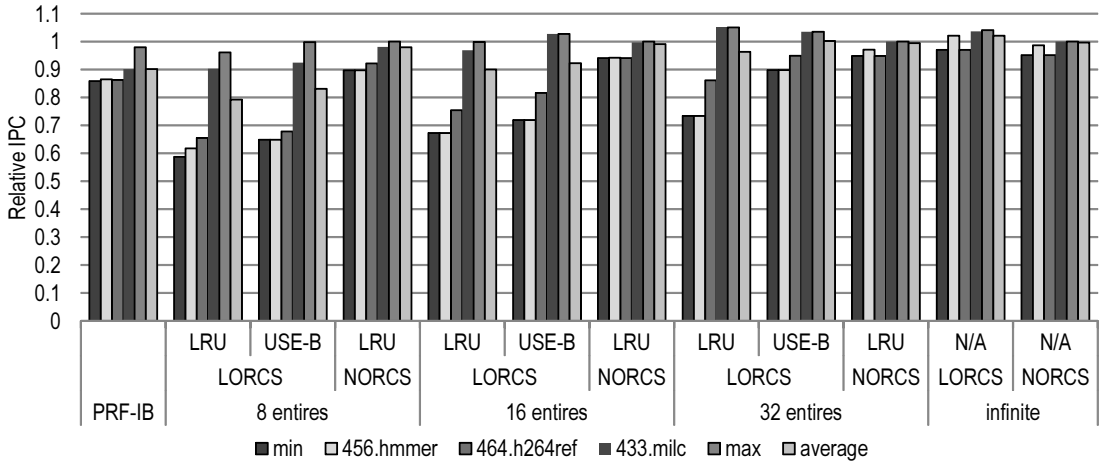


図 4.20: 各モデルの相対 IPC

に高い性能と低い性能を示した特定のベンチマークについての結果である．

グラフからは，NORCS による性能低下は非常に小さいことがわかる．レジスタ・キャッシュの容量が 8 エントリであっても，IPC の低下はたかだか 2.0 %程度である．また，プログラムごとの IPC についても大きな変化はないことがわかる．NORCS では，レジスタ・キャッシュ・ヒット率が性能に及ぼす影響は小さいため，このように大きな性能低下は起きない．（4.4.2 節）．

LORCS のうち，“infinite” エントリのレジスタ・キャッシュを持つものと，32 エントリの USE-B ポリシを行うレジスタ・キャッシュについては，ベースライン・モデルに対して性能向上を見せている．これらは，パイプライの短縮によるものである．しかし，この性能向上はわずかなものであり，前者で 2.1%，後者で 0.2%程度にすぎない．

これに対し，その他の構成を持つ LORCS は大幅な性能低下を見せている．USE-B ポリシの 8 エントリと 16 エントリのレジスタ・キャッシュでは，性能低下はそれぞれ 16.9% と 7.3% である．LRU ポリシの場合，性能低下はより顕著である．8，16，32 エントリのレジスタ・キャッシュにおいて，それぞれ 20.8%，10.0%，3.6% 性能が低下している．これに加え，LORCS では，プログラムごとの IPC についても大きな変化を見せている．特に，LRU ポリシによる 32 エントリのレジスタ・キャッシュを持つ構成では，456.hmmer において 25%以上性能が低下している．

これらの性能低下は，LORCS ではレジスタ・キャッシュ・ヒット率が性能に大きな影響を与えるためである（4.4.2 節）．レジスタ・キャッシュ・ヒット率は，その置き換えアルゴリズムに大きく依存し，また，プログラム毎にも大きく異なる．平均的な性能が良い場合でも，前述したような特定のプログラムにおいて大きな性能低下が起きることは望ましくない．

全体として，PRF モデルより良い性能を得るためには，

1. NORCS では 8 エントリの LRU ポリシによるレジスタ・キャッシュがあれば十分であるのに対し，
2. LORCS では 32 エントリの USE-B ポリシによるレジスタ・キャッシュが必要である

と言える．また，上記の構成は同時に，NORCS と LORCS のそれぞれにおいて “infinite” エントリのレジスタ・キャッシュを持つ場合と同程度の性能を達成するために必要なパラメータでもある．

4.5.5 実効ミス率

表 4.3 と表 4.4 に，LORCS と NORCS の実効ミス率を示す．これらの実効ミス率は，パイプラインが乱れる確率，すなわち 1 サイクルあたりのレジスタ・キャッシュ・ミスに起因するストールの発生率を表す．表 4.3 は 32 エントリの USE-B ポリシによるレジスタ・キャッシュを持つ場合の結果であり，表 4.4 は 8 エントリの LRU ポリシによるレジスタ・キャッシュを持つ場合の結果である．これらの 2 つのモデルは，4.5.4 節 でほぼ同じ性能を示したものである．

これらの表では，いくつかの特徴的なベンチマークの結果と，全ベンチマークの平均の結果を載せている．表内の “Issued ” と “Read” の列は，サイクルあたりの命令発行数と，サイクルあたりのレジスタ・キャッシュにアクセスを行ったオペランドの数をそれぞれ表す．“RC Hit” と “Effec Miss” は，アクセスあたりのレジスタ・キャッシュ・ミス率と，パイプラインの実効ミス率をそれぞれ表す．

4.4.2 節で述べたように，LORCS の実効ミス率は，レジスタ・キャッシュそのもののミス率よりも非常に大きい．表 4.3 では，464.h264ref のレジスタ・キャッシュ・ヒット率は 99.0% あるが，しかし，サイクルあたりのオペランド数が 2.57 あるため，実効ミス率は 8.8% にもなっている．この結果，464.h264ref の IPC は 5.1% 低下している．

4.4.2 節で述べたように，NORCS の実効ミス率は，レジスタ・キャッシュ・ヒット率の影響をあまり受けない．表 4.4 内の NORCS のレジスタ・キャッシュ・ヒット率は，表 4.3 内の LORCS のものよりも大幅に低い．これは，レジスタ・キャッシュの容量が 4 分の 1 であることと，置き換えアルゴリズムがより単純な LRU であるためである．これに対し，それぞれの実効ミス率や IPC は大きくは違わないことがわかる．NORCS は，LORCS よりも大幅に低いレジスタ・キャッシュ・ヒット率しか持たない場合であっても，同等の性能を達成できるのである．

4.5.6 回路面積と消費電力

CACTI 5.3 モデル [3] を用いて，各モデルにおける回路面積と消費電力を評価した．評価は，ITRS の 45 nm と 32 nm テクノロジ・ノード [47] を用いて行ったが，両者はほぼ同じ傾向を示したため，以下では 32nm のものについてのみ示す．

表 4.3: LORCS の Effective ミス率

	LORCS with 32 entry-RC (USE-B)				
	Issued	Read	RC Hit(%)	Effc Miss(%)	IPC
429.mcf	0.44	0.53	92.2	3.4	0.99
456.hmmer	1.88	2.49	94.2	15.7	0.90
464.h264ref	1.91	2.57	99.0	8.8	0.95
average	1.48	1.90	98.6	2.7	1.00

表 4.4: NORCS の Effective ミス率

	NORCS with 8 entry-RC (LRU)				
	Issued	Read	RC Hit(%)	Effc Miss(%)	IPC
429.mcf	0.49	0.60	82.83	0.98	1.00
456.hmmer	1.90	2.53	62.95	11.7	0.90
464.h264ref	1.87	2.51	73.14	8.7	0.92
average	1.46	1.89	79.91	2.3	0.98

4.5.6.1 回路面積

図 4.21 に，各モデルの PRF モデルに対する相対回路面積を示す．これらの回路面積は，レジスタ・キャッシュとメイン・レジスタ・ファイル，Use-predictor のものを含む．

レジスタ・キャッシュとメイン・レジスタ・ファイルの面積は，NORCS と LORCS で同じである．これはそれぞれのアレイが同じエントリ数とポート数を持ち，物理的には同じものであるためである．

メイン・レジスタ・ファイルの回路面積は，PRF のものと比較して 12.2% にまで縮小されている．これはメイン・レジスタ・ファイルのポート数が 12 から 4 にまで減っているためである．RAM の回路面積はポート数の 2 乗に比例するため，結果として回路面積は大幅に縮小される．$(4/12)^2 \approx 11.1\%$ であり，実際の縮小率である 12.2% におおよそ一致する．これに対し，レジスタ・キャッシュの回路面積はその容量の割に大きい．これは，レジスタ・キャッシュが 12 ポートを持つためである．

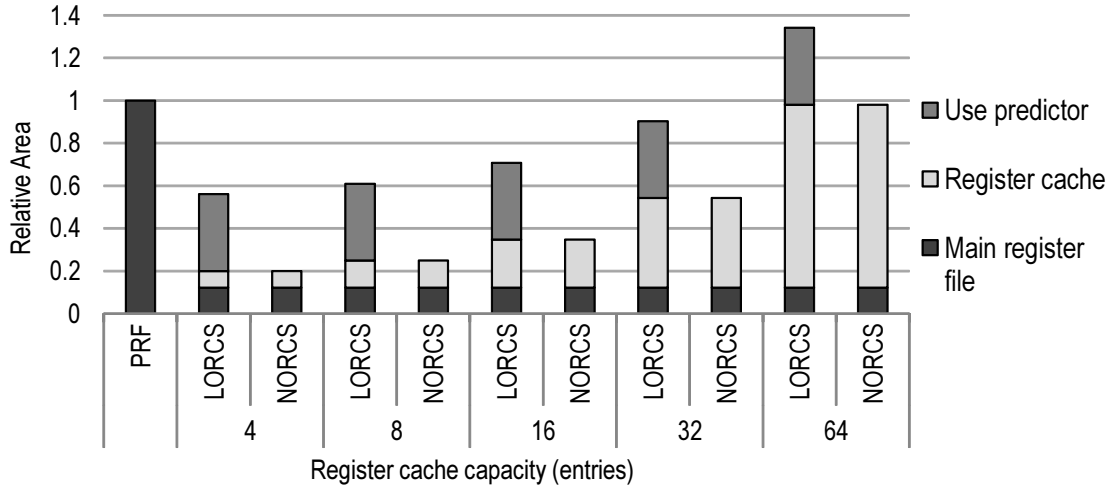


図 4.21: 相対回路面積

Use-predictor の回路面積は，PRF モデルによるレジスタ・ファイルの 36.1% になり，レジスタ・キャッシュやメイン・レジスタ・ファイルに比べると相対的に大きい．これは，Use-predictor が多数のポートを持つためである．一般に，Use-predictor のような予測器は，フェッチ幅に応じた数の読み出しポートと，リタイア幅に応じた数の書き込みポートが必要となる．読み出しポートはフロントエンドにおける予測データの読み出しに使われ，書き込みポートはリタイア時の学習データの書き込みに使われる [59]．表 4.1 に示す評価環境の場合，Use-predictor には読み書きそれぞれ 4 ポートが必要となる．

NORCS のレジスタ・キャッシュとメイン・レジスタ・ファイルの回路面積を合わせたものは，PRF によるレジスタ・ファイルと比べて大幅に縮小されている．回路面積は，4，8，16，32，64 エントリのレジスタ・キャッシュの場合で，19.9%，24.9%，34.7%，42.0%，98.0%にまでそれぞれ縮小されている．LORCS では，レジスタ・キャッシュとメイン・レジスタ・ファイルの回路面積そのものは NORCS と変わらない．しかし，Use-predictor を含んだ場合，回路面積は非常に大きなものとなっている．その回路面積は，4，8，16，32，64 エントリのレジスタ・キャッシュの場合で，PRF によるレジスタ・ファイルと比べてそれぞれ 56.1%，61.0%，70.8%，90.3%，134.1% となっている．

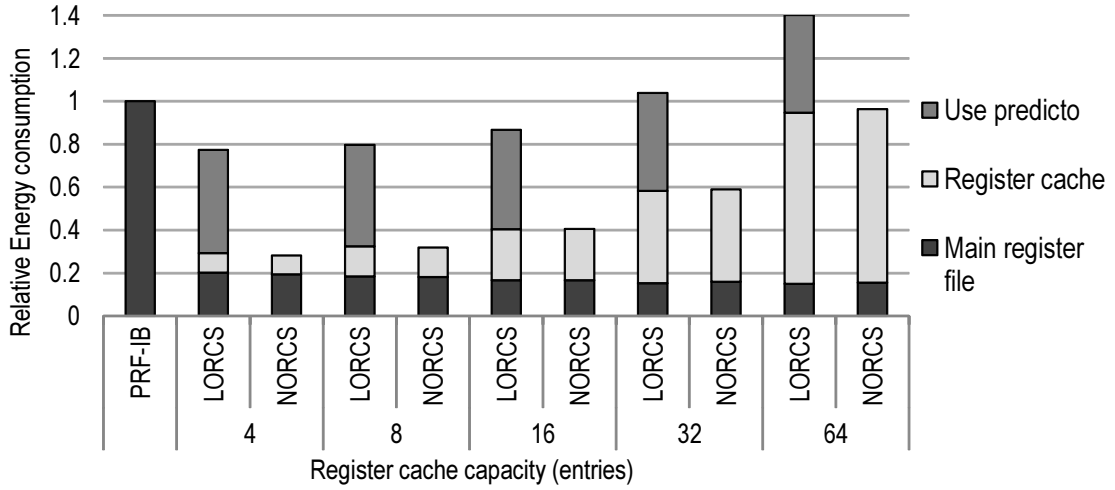


図 4.22: 平均相対消費電力

4.5.6.2 消費電力

図 4.22 に，各モデルの PRF モデルに対する相対消費電力を示す．これらの消費電力は，レジスタ・キャッシュとメイン・レジスタ・ファイル，Use-predictor のものを含む．これらの消費電力はプログラム毎に評価を行ったものを，全体で平均したものである．

NORCS のレジスタ・キャッシュとメイン・レジスタ・ファイルの消費電力を合わせたものは，PRF によるレジスタ・ファイルのものと比べて大幅に縮小されている．消費電力は，4，8，16，32，64 エントリのレジスタ・キャッシュの場合で，28.2%，31.9%，40.6%，59.0%，96.3%にまでそれぞれ縮小されている．

NORCS と LORCS では，レジスタ・キャッシュとメイン・レジスタ・ファイルのみによる消費電力の違いはわずかである．これは，双方でレジスタ・キャッシュとメイン・レジスタ・ファイルのアレイそのものは同じであることと，それらに対するアクセス数も大きくは変わらないためである．これに対し，LORCS では，Use-predictor の分を含んだ場合の消費電力は非常に大きい．Use-predictor による消費電力は，PRF モデルによるレジスタ・ファイルの 48.1% にもなる．Use-predictor を含んだ場合の消費電力は，4，8，16，32，64 エントリのレジスタ・キャッシュの場合で，PRF によるレジスタ・ファイルと比べてそれぞれ 77.4%，79.8%，86.7%，103.8%，140.1% となっている．

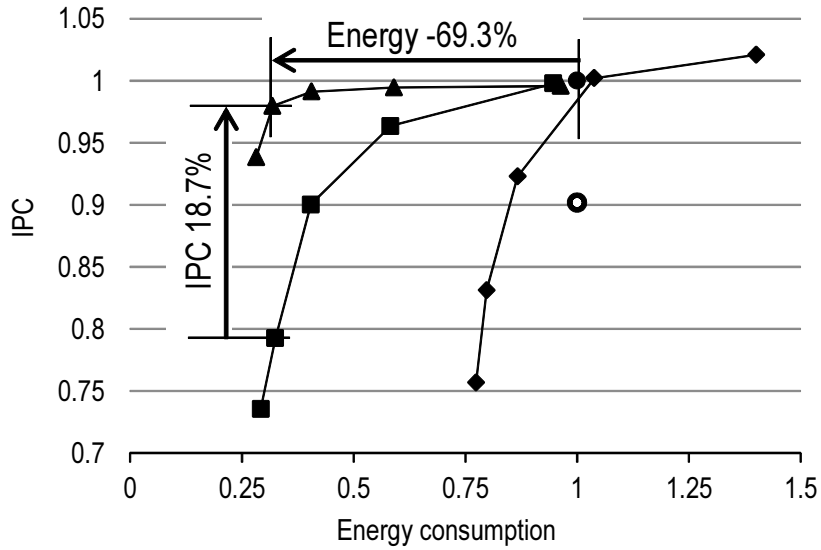
4.5.7 消費電力あたりのIPC

図4.23に，各モデルの消費電力あたりの相対IPCを示す．各IPCと消費電力は，PRFモデルによるレジスタ・ファイルのものに対する相対値であり，それぞれは図4.20と図4.22に示したものと同じである．グラフの各線上の点は，左から右に向かって4，8，16，32，64エントリのレジスタ・キャッシュのものを表す．図4.23(a)は，ベンチマーク全体で平均を取った場合のものであり，図4.23(b)はベンチマークの中で最悪のIPCを持つものについてのグラフである．IPCは高いほど，また消費電力は少ないほどよいため，これらのグラフでは左上にある点ほど消費電力あたりの性能が良いことを意味する．

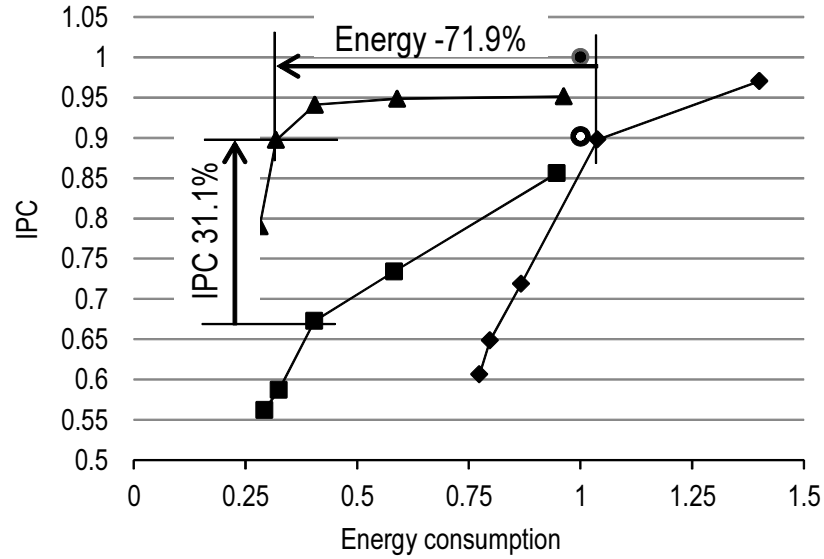
図4.23(a)からは，NORCSは大きな性能低下を伴うことなく消費電力を下げていることがわかる．これに対し，LORCSではLRUとUSE-Bポリシの双方において，IPCと消費電力はトレードオフの関係にある．LORCSでは，小容量のレジスタ・キャッシュを用いることで消費電力を削減することができるが，同時にIPCも下がってしまう．

8エントリのLRUポリシによるレジスタ・キャッシュを持つNORCSと，64エントリのLRUポリシによるレジスタ・キャッシュを持つLORCSでは，IPCは大きくは違わず，その差は1.8%である．しかし，それらの消費電力は大きく異なり，同構成のNORCSはLORCSと比べて69.3%消費電力を削減している．NORCSとLORCSのうち，8エントリのLRUポリシによるレジスタ・キャッシュを持つ場合は，それぞれほぼ同程度の電力を消費する．しかし，IPCは大きく異なり，同構成のNORCSはLORCSと比べてIPCが18.7%向上している．

図4.23(b)では，NORCSとLORCSの差はよりはっきりと現れている．8エントリのLRUポリシによるレジスタ・キャッシュを持つNORCSと，32エントリのUSE-Bポリシによるレジスタ・キャッシュを持つLORCSでは，ほぼ同じIPCを示している．これに対し，それらの消費電力の差は大きく，同構成のNORCSは，LORCSと比べて消費電力を71.9%削減している．同図でも，NORCSとLORCSのうち，8エントリのLRUポリシによるレジスタ・キャッシュを持つ場合は，それぞれほぼ同程度の電力を消費する．しかし，IPCは大きく異なり，同構成のNORCSはLORCSと比べてIPCが31.1%向上している．



(a) 平均



(b) 最悪

図 4.23: 消費電力あたりの IPC

4.5.8 発行幅の広いスーパスカラ・プロセッサでの評価

これまでに述べた評価結果は，4.5.1 節で説明したような，現実的な大きさのスーパスカラ・プロセッサを対象として行ったものである．これに対し，これまでに提案されている LORCS は，512 エントリのレジスタ・ファイルを持ち，8 命令同時発行が可能なような非常に発行幅の広いスーパスカラ・プロセッサを対象としたものであった [8]．本節では，このような発行幅の広いスーパスカラ・プロセッサにおけるレジスタ・キャッシュ・システムの評価について述べる．

表 4.1 と 表 4.2 内の右側の列にある “Ultra-wide” に，本節においてベースラインとして用いるスーパスカラ・プロセッサの構成を示す．この構成は 8 命令同時発行可能なスーパスカラ・プロセッサであり，Butts らが LORCS の評価に用いたものとほぼ同じものである [8]．レジスタ・キャッシュ・システムの構成については，これまでに述べた評価のものと大きくは変わらないが，いくつかの部分を Butts らが評価に用いたものと同様のものに変更した．メイン・レジスタ・ファイルのポート数は読み書き共に 4 ポートとし，レジスタ・キャッシュは 2 ウェイのセット・アソシアティブ構成である．レジスタ・キャッシュのインデクシングには，4.5.1.2 節で述べた，Butts らの提案しているインデクシングを用いている [8].

図 4.24 に各モデルの相対 IPC を示す．これらは，ベースラインとなる PRF モデルの IPC に対する相対 IPC である．各ラベルの意味については，4.5.4 節のものと同じである．

グラフより，各モデルにおける IPC は，4.5.4 節で述べた評価結果と同様の傾向を見せている．NORCS による性能低下は小さく，16，32，64 エントリのレジスタ・キャッシュを持つ場合において，それぞれ 0.12%，0.6%，0.03%の性能低下となっている．これに対し，LORCS は大きく性能を低下させており，16，32，64 エントリのレジスタ・キャッシュを持つ場合において，それぞれ 16%，9.7%，4.3%の性能低下となっている．

Butts らは PRF-IB モデルに対して LORCS の評価を行っており，64 エントリの USE-B ポリシによるレジスタ・キャッシュを持つ LORCS であれば，PRF-IB よりも良い性能となることを示した [8]．本節で行った評価の場合，同じ構成の 64 エントリのレジスタ・キャッシュを持つ LORCS は，PRF-IB に対して 6.6%の性能向上を示している．これは Butts らの評価結果である 6% と良く一致する．これに対し，16 エントリの LRU ポリシによるレジスタ・キャッシュを持つ NORCS はこれより

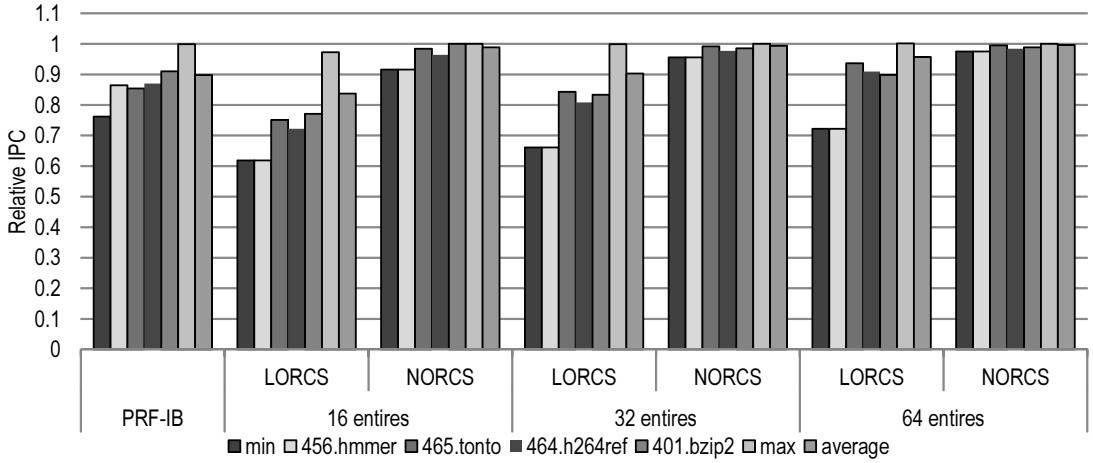


図 4.24: 8 命令同時発行可能なプロセッサにおける相対平均 IPC

も良い性能を示しており，PRF-IB よりも 10.1%性能が向上している．

これらの評価結果は，NORCS は，非常に幅の広いスーパスカラ・プロセッサの場合であってもうまく働くことを示している．16 エントリの LRU ポリシによるレジスタ・キャッシュを持つ NORCS は，64 エントリの USE-B ポリシによるレジスタ・キャッシュを持つ LORCS よりも良い性能を示す．

4.5.9 SMT プロセッサによる評価

4 節で述べたように，SMT プロセッサでは，より大きなレジスタ・ファイルが必要となる．本節では，表 4.1 の左側の列にある構成において，2 ウェイの SMT 実行を行った場合の評価について述べる．

図 4.5.9 に，各モデルの消費電力あたりの IPC を示す．グラフの見方については，4.5.7 節内のものと同じである．これらの結果は，SPEC CPU 2006 に含まれる 29 本のベンチマークについて，全ての組み合わせを 2 スレッドで実行して評価を行い，それらの平均を取ったものである．

グラフからは，4.5.4 節で述べたシングルスレッド実行の場合と比べ，IPC が全体に低下していることがわかる．しかし，NORCS の性能低下は小さなものに留まっている．8 エントリと 16 エントリのレジスタ・キャッシュを持つ場合，それらの性能低下はそれぞれ 4.1%と 1.8%である．これに対し，LORCS は大きな性能低下を見

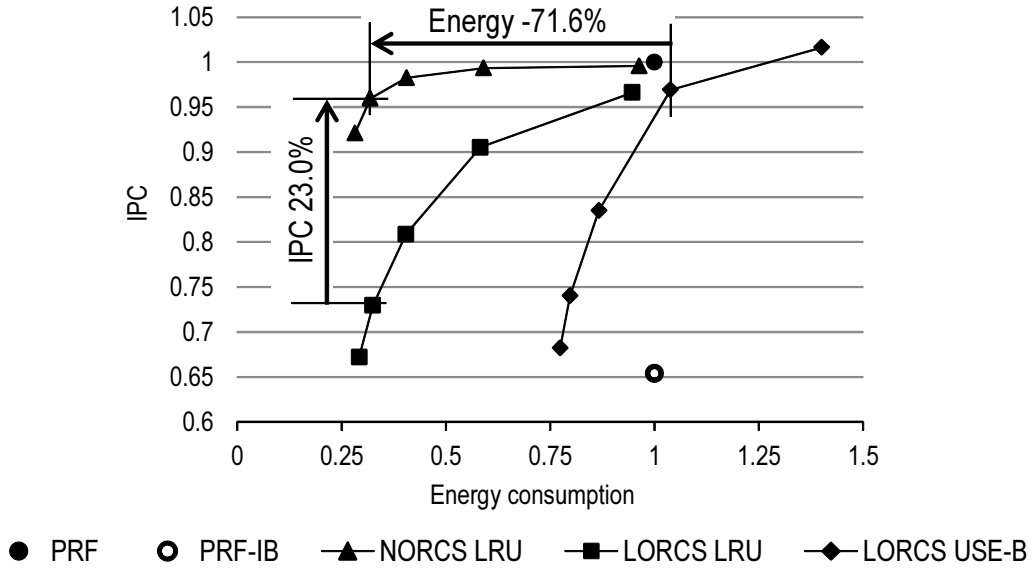


図 4.25: SMT プロセッサにおける消費電力あたりの性能

せている．USE-B ポリシの場合であっても，8，16，32 エントリのレジスタ・キャッシュを持つ LORCS では，性能はそれぞれ 26.0%，16.5%，3.1%低下している．

8 エントリの LRU ポリシによるレジスタ・キャッシュを持つ NORCS と，32 エントリの USE-B ポリシによるレジスタ・キャッシュを持つ LORCS では，ほぼ同じ IPC を示しており，その差は 0.94%である．これに対し，それらの消費電力の差は大きく，同構成の NORCS は，LORCS と比べて消費電力を 71.6%削減している．同図でも，NORCS と LORCS のうち，8 エントリの LRU ポリシによるレジスタ・キャッシュを持つ場合は，それぞれほぼ同程度の電力を消費する．しかし，IPC は大きく異なり，同構成の NORCS は LORCS と比べて IPC が 23.3%向上している．

これらの評価結果は，NORCS は，SMT プロセッサ上であってもうまく働くことを示している．

4.6 本章のまとめ

レジスタ・ファイルは，最近のスーパスカラ・プロセッサの中で最も高コストなユニットの 1 つである．4 節で述べた様に，巨大なレジスタ・ファイルはさまざまな問題を引きおこす．本章では，レイテンシの短縮を行わないレジスタ・キャッ

シュ・システムである，NORCS の提案を行った．NORCS は，レジスタ・キャッシュのミスを仮定したパイプラインを持つことに特徴がある．提案では，メイン・レジスタ・ファイルへのアクセス・ステージを設けるため，既存のレジスタ・キャッシュ・システムに対してパイプライン・ラッチを追加した．この小さな変更は，しかし性能において非常に大きな差を生じる．NORCS は，レジスタ・キャッシュのミスのみではパイプラインが乱れないためである．

本章では，NORCS と LORCS，パイプライン化されたレジスタ・ファイルについて，IPC や回路面積，消費電力の評価を行った．これらの評価は，現実的な規模のスーパスカラ・プロセッサから 8 ウェイの非常に大きなスーパスカラ・プロセッサ，また SMT プロセッサについてのものを含む．評価の結果，NORCS は，2.1%の小さな性能低下で，回路面積と消費電力をパイプライン化されたレジスタ・ファイルに対してそれぞれ 24.9% と 31.9% にまで削減できる事を示した．8 エントリの LRU ポリシによる NORCS は，既存の 32 エントリの USE-B ポリシによる理想的なモデルに近いヒット率を持った LORCS と同等の性能を持つ．しかし，その回路面積と消費電力は大きく異なり，同構成の NORCS は，LORCS と比べて回路面積と消費電力をそれぞれ 27.6% と 31.9% にまで減らす事ができる．

第5章

要素技術の統合と評価

本章では，これまでに述べた面積効率を向上させる要素技術について，それらの統合と評価について述べる．以下，5.1 節では要素技術の統合について説明する．続く 5.2 節では，要素技術を統合したプロセッサの面積効率ついて評価を行う．

5.1 要素の技術の統合

表 5.1 に，前章までに述べた Out-of-Order スーパスカラ・プロセッサの主要なコンポーネントと，その回路面積を縮小する要素技術についてまとめる．

これらの要素技術は，基本的にはそれぞれが直行しているため，そのまま統合することが可能である．また，統合によって新たに得られる利点として，マトリクス・スケジューラのために必要なプロデューサ・テーブル（2.3.2 節）の省略がある．5.1.1 節では，このプロデューサ・テーブルの省略について説明する．

命令ウィンドウの非集中化とリネームド・トレース・キャッシュの統合は，そのままでは行えない．非集中化を行う事により，サブウィンドウ間での依存関係（変位）について，考慮する必要があるためである．5.1.2 節では，この命令ウィンドウの非集中化とリネームド・トレース・キャッシュの統合について説明する．

5.1.1 プロデューサ・テーブルの省略

マトリクス・スケジューラとリネームド・トレース・キャッシュは，組み合わせて使用することにより，マトリクス・スケジューラのために必要なプロデューサ・

表 5.1: 主要なコンポーネントと面積効率を向上させる要素技術

ステージ	コンポーネント	要素技術
リネーミング	RMT	リネームド・トレース・キャッシュ
ディスパッチ/発行	命令キュー	命令ウィンドウの非集中化
セレクト	セレクト・ロジック	命令ウィンドウの非集中化
ウェイクアップ	ウェイクアップ・ロジック	マトリクス・スケジューラ
レジスタ・アクセス	レジスタ・ファイル	NORCS
メモリ・アクセス	ロード・ストア・キュー	NoSQ

テーブル（2.3.2 節）を省略することができる．リネームド・トレース・キャッシュでは，レジスタ番号による依存関係を命令間の変位へ変換し，キャッシュする．この命令間の変位は，プロデューサ・テーブルの内容そのものである．このため，リネームド・トレース・キャッシュのフェッチによって得られた命令間の変位を元に，マトリクス・スケジューラにおける依存行列の内容を得ることができる．

5.1.2　リネームド・トレース・キャッシュの非集中化への対応

3 節で述べたように，リネームド・トレース・キャッシュは命令ウィンドウ中の変位によって依存関係を表現する．このため，命令ウィンドウを複数に非集中化した場合，変位の表現もそれにあわせて対応を行う必要がある．以下ではこれを実現するための 2 つの異なる方法について説明する．

5.1.2.1　論理的な集中を保った非集中化の方法

リネームド・トレース・キャッシュの非集中化を行う方法の 1 つは，命令ウィンドウを論理的には集中化したまま，物理的に非集中化することである．図 5.1 に，これによる非集中化を行った命令ウィンドウの実装例を示す．同図では，INT，LS，FP の系統ごとのサブウィンドウと，INT と FP の 2 つのレジスタ・ファイルに非集中化されている．

この実装では，命令ウィンドウやレジスタ・ファイルは論理的には集中化されたままである．各サブウィンドウとレジスタ・ファイルでは，同一の ID を持つエントリは，1 つの命令によって占有される．図 5.1 の場合，最上段には INT の add 命令が格納されているため，LS と FP のサブウィンドウ，および FP のレジスタ・

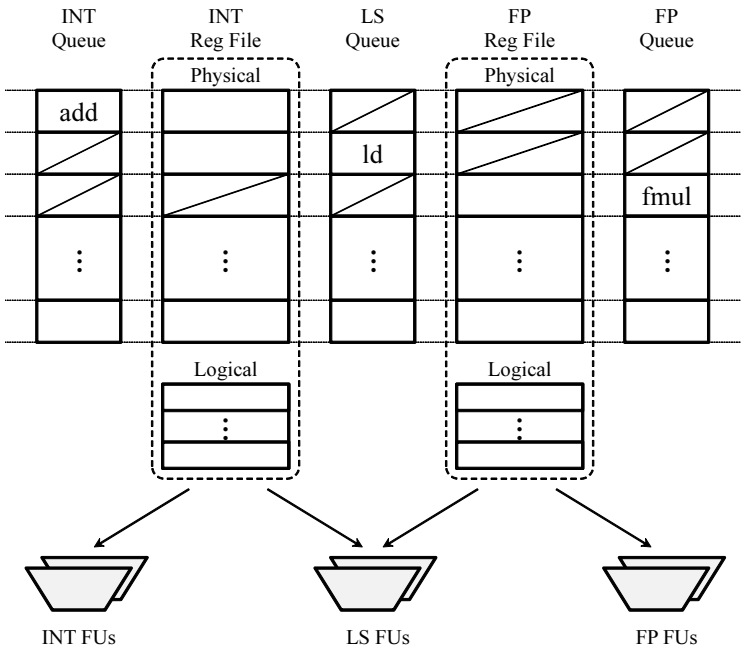


図 5.1: 命令ウィンドウの論理集中/物理非集中化

ファイルは使用できない．論理的には集中型命令ウィンドウと等価であるため，リネームド・トレース・キャッシュの命令の変換は，集中型の命令と同じように行えばよい．

この実装では，発行幅 IS が分割されるため，セレクト・ロジックや命令キューを縮小する事ができる．ただし，ウィンドウ・サイズ WS は縮小されず，さらにサブウィンドウやレジスタ・ファイル間で，ID が同一のエントリは 1 つしか使用できないため，容量効率が低下する．

5.1.2.2 物理レジスタ・ファイルの分割による非集中化の方法

リネームド・トレース・キャッシュの非集中化を行うもう 1 つの方法は，物理レジスタをサブウィンドウごとに保持するよう，分割を行うことである．図 5.2 に，これによる非集中化を行った命令ウィンドウの実装例を示す．同図では，命令ウィンドウは INT，LS，FP の系統ごとのサブウィンドウに非集中化されている．また，INT の物理レジスタ・ファイルは INT と LS に，FP の物理レジスタ・ファイルは LS と FP に，それぞれサブウィンドウ毎に非集中化されている．なお，論理レジスタ・ファイルについては，INT と FP の 2 つである．同図の実装では，簡単のた

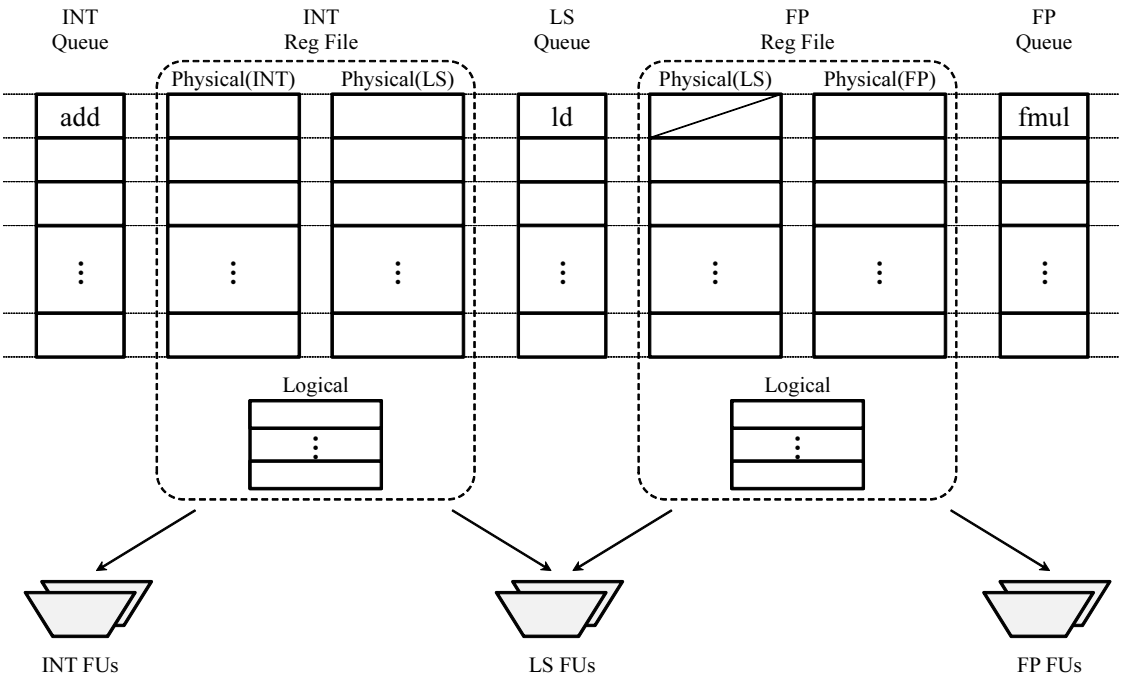


図 5.2: 物理レジスタ・ファイルの分割による命令ウィンドウの非集中化

め，INT と FP 間の直接的なデータの授受は無いものとしている．もし，INT と FP 間で直接にデータの授受を行う場合は，INT と FP のサブウィンドウごとに，それぞれ FP と INT の物理レジスタ・ファイルを持つ必要がある．以下では，この非集中化を行うために必要なレジスタ参照モデルの変更について説明する．

変位の表現方法の変更

この実装では，論理的には単一である INT や FP のレジスタを，それぞれのサブウィンドウごとの物理レジスタ・ファイルに非集中化している．各サブウィンドウ中の命令は，自身に対応する物理レジスタ・ファイルのエントリに実行結果を書き込む．

この非集中化を行うため，3.2.1 節で述べた変位の表現変更に変更を加える．命令は，各サブウィンドウ中の変位とサブウィンドウの種類を指定することによってソース・オペランドを指定する．

図 5.3 に，この変更を行った DMT と，それが指す物理レジスタのエントリの対応を示す．この DMT は，

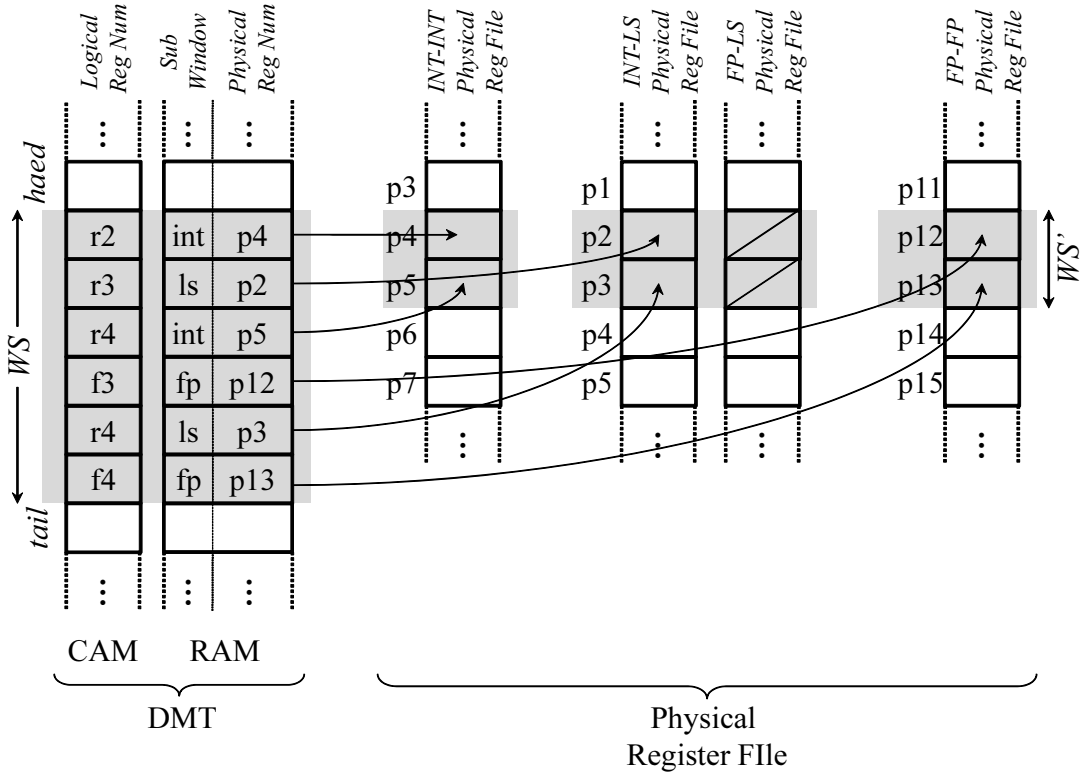


図 5.3: 物理レジスタ・ファイルを非集中化した場合の DMT

1. 3.3.1 節 で述べた CAM 式 DMT の CAM そのものと，
2. CAM のエントリからサブウィンドウへのマッピングを保持する RAM

からなる．RAM の各エントリは，サブウィンドウの種類とサブウィンドウ中の位置を保持する．

この DMT への参照は，まず，3.3.1 節で述べた CAM 式 DMT の場合と同様に，論理レジスタ番号をキーにして CAM の優先順位付き検索を行う．通常の CAM 式 DMT の場合，この優先順位付き検索の結果が，そのままソース・オペランドの位置となる．これに対し，非集中化のための変更を行った DMT では，優先順位付き検索のヒットした位置をインデックスとして RAM のエントリを読み出す．読み出されたエントリの内容に従って，最終的な参照先のサブウィンドウとサブウィンドウ中の変位を得る．

単一の DMT を検索することによって参照先を決定するのは，論理レジスタに対する更新のうち，プログラム・オーダにおいて最新のものを取得するためである．

DMT を物理レジスタに合わせて非集中化した場合，同一の論理レジスタに対して，どのサブウィンドウによる更新が最新のものかがわからなくなってしまう．

DMT 側の命令ウィンドウは，非集中化されたレジスタ・ファイルにおける各サブウィンドウの集合となっている．レジスタ・ファイルにおける各サブウィンドウの $head$ ポインタと $tail$ ポインタは独立して制御を行う．これに対し，DMT 側の $head$ ポインタについては，各サブウィンドウ中のプログラム・オーダ上で最も後続にある $head$ に同期させて制御を行う．また，$tail$ も同様にして，各サブウィンドウ中で最も先行する $tail$ に合わせて制御を行う．

5.1.2.3　非集中化の効果

本項で述べた方法は，2.3.1 節の場合と同様に，命令ウィンドウ・サイズ WS と発行幅 IS の双方を縮小することが出来る．また，5.1.2.1 節で述べた，論理的な集中を保つ方法で発生する容量効率の低下は発生しない．

これに加え，物理レジスタ・ファイル非集中化されて演算器と直結しているため，それぞれの書き込みポートを削減することができる．ただし，サブウィンドウ間でお互いの物理レジスタ・ファイルを読み出すため，読み出しポート数は削減されない．

本項で述べた方法の不利な点は DMT に追加の RAM 領域が必要である点である．しかし，DMT は少数のポートのみを持つため（3.3.1 節），通常このことは大幅な回路面積の増加にはつながらない．

また，物理レジスタ・ファイルが非集中化されるため，断片化による性能低下が起きる可能性がある．ただし，この点においても，通常の命令ウィンドウの非集中化による断片化が性能をほとんど下げないのと同様に（2.3.1 節），大きな性能低下には繋がらない．

5.2　面積効率の評価

本節では，これまでに提案した**リネームド・トレース・キャッシュ**（3 節）や**NORCS**（4 節），およびその他の面積効率を向上させる技術（2.3 節）を統合した場合の，プロセッサ全体の面積効率について評価を行う．

1 節で述べたように，面積効率とは，回路面積あたりの実行性能である．このうち，実行性能の評価は 3.4 節や 4.5 節で行った評価と同じように，**鬼斬弐**[44] を用いて評価を行った．ベンチマーク・プログラムや評価を行う命令数などについても，基本的には先に行った評価の場合と同じである．評価では SPEC CPU2006 [58] に含まれる全 29 本のベンチマーク・プログラムを用いた．ベンチマーク・プログラムのコンパイルには gcc 4.2.2 を用い，コンパイル・オプションには “–O3”を指定した．入力データ・セットには ref を用い，プログラムの先頭 1 G 命令をスキップして，続く 100 M 命令の評価を行った．

回路面積については，2.2.3 節で述べた Alpha 21464[31] のフロア・プランを元に，各コンポーネントの処理幅とエントリ数からその回路面積の見積もりを行った．この回路面積の見積もりモデルについては後で詳しく述べる．

5.2.1 評価モデル

プロセッサの面積効率はキャッシュの容量や Out-of-Order 実行の有無によって大きく変化する．そこで，本節では以下のモデルのそれぞれについて，L2 キャッシュの容量を変化させながら評価を行った．

Inorder:

In-Order 実行を行うモデル．発行幅については，1，2，4，8 命令のものについて評価を行った．表 5.2 の “Inorder” の列に，これらのプロセッサのパラメータを示す．列内の 1w，2w，4w，8w は，それぞれのコンポーネントにおける処理の幅を表す．ここで，処理の幅とは 1 サイクルあたりに同時に処理可能な命令の数である（2.2.1 節）．特に，“Inorder” ラベルの直下にある 1w，2w，4w，8w については，それぞれ発行幅を表す．演算器の数については，2，4，8 命令同時発行可能なモデルにおいて，それぞれ Alpha 21064[61]，Alpha 21264[21]，Alpha 21464[31] と同じとなるよう設定した[1]．

OoO-Base:

Out-of-Order 実行を行うモデル．同時発行可能な命令数については，Inorder モデルの場合と同様に 1，2，4，8 命令のものについて評価を行った．表 5.2 の

[1] Alpha 21064 は 2 命令同時発行可能な In-Order スーパスカラ・プロセッサ， Alpha 21264 と Alpha 21464 は，それぞれが 4 命令と 8 命令同時発行可能な Out-of-Order スーパスカラ・プロセッサである．

表 5.2: 各モデルにおけるコンポーネントの構成

Type	Component		Inorder 1w	2w	4w	8w	OoO 1w	2w	4w	8w	Alpha 21464 8w
Mem	Control	IW	N/A				64 entries				128 entries
							1w	2w	4w	8w	8w
		RMT	N/A				64 entries				64 entries
							1w	2w	4w	8w	8w
		LSQ	N/A				64 entries				64 entries
							1w	2w	2w	4w	4w
		RF	64 entries				256 entries				512 entries
			1w	2w	4w	8w	1w	2w	4w	8w	8w
	Cache	L1I	32 KB				←				64 KB
			1w								2w
		L1D	32 KB				←				64 KB
			1w	2w	2w	4w					4w
		L2	0∼2 MB				←				3 MB
			1w								1w
	Branch Predictor		8 KB				←				44 KB
			1w								1w
Logic	FU	INT	1w	2w	4w	8w	←				8w
		FP	1w	1w	2w	4w	←				4w

表 5.3: 各モデルの基本的な構成

Name	Inorder	OoO
pipeline stages	fetch:3, decode:1, reg r/w:2	fetch:3, rename:2, dispatch:2, issue:2, reg r/w:2
branch miss penalty	7 cycles	13 cycles
branch predictor	8 KB g-share	←
BTB	2 K entries, 4 way	←
RAS	8 entries	←
L1IC	32 KB, 4 way set assoc, 64 B/line, 3 cycles	←
L1DC	32 KB, 4 way set assoc, 64 B/line, 3 cycles	←
L2C	0∼2 MB, 8 way set assoc, 64 B/line, 10 cycles	←
main memory	200 cycles	←

“OoO” の列にこれらのプロセッサのパラメータを示す．Out-of-Order 実行のモデルについては，発行幅に関わらず 64 エントリの集中型の命令ウィンドウを持つものとした．Out-of-Order 実行の制御に関わる部分以外のコンポーネントについては Inorder モデルのものと同じである．すなわち，キャッシュの容量や分岐予測器の容量，演算器の数は，同時発行可能な命令数が同じ Inorder モデルのものと同じである．

OoO-Prop:

Out-of-Order 実行を行うモデル．OoO-Base モデルとの違いは，これまでに提案した**リネームド・トレース・キャッシュ**，**NORCS**，および既に提案されている**マトリクス・スケジューラ**と **NoSQ** を実装していることである．それぞれのモデルについては後で詳しく述べる．

各モデルの，その他のパラメータを表 5.3 にまとめる．モデル間におけるパラメータの主な違いは，方式の違いによるパイプライン・ステージ構成の違いと，それによる分岐予測ミス・ペナルティの違いである．Inorder モデルでは，Out-of-Order 実行に必要ないくつかのステージが存在しないため，命令パイプラインが Out-of-Order 実行を行うモデルよりも短い．その他のパラメータについては，基本的には各モデルにおいて共通となっている．

5.2.2 回路面積のモデル

表 5.2 に示す各コンポーネントについて回路面積の見積もりを行い，それらの合計を用いることでプロセッサ全体の回路面積の検討を行った．各コンポーネントの回路面積については，2.2.3 節で述べた Alpha 21464[31] のフロア・プランを元に，各コンポーネントの処理幅とエントリ数から見積もりを行った．以下では，この回路面積のモデルを演算器とメモリの場合に分けて説明する．

演算器

演算器の占める回路面積は単純にその搭載数に比例する [2]．そこで，Alpha 21464 のフロア・プランを元に，整数演算器と浮動小数点演算器の 1 つあたりの回路面積を求め，各モデルの持つ演算器の数をかけあわせることによって回路面積を求めた．

メモリ

表 5.2 に示す演算器以外のコンポーネントは，全て RAM ないしは CAM によって構成される（2.2.2 節）．この RAM と CAM の回路面積は，そのエントリ数とポート数の 2 乗に比例する [2]．そこで，フロア・プランから実際の面積を計測し，そのエントリ数とポート数の 2 乗によって正規化した定数を求めることで，各コンポーネントの持つエントリ数やポート数から，その回路面積を見積もった．

ただし，Alpha 21464 ではレジスタ・ファイルや命令キューのマルチバンク化が行われているため，それらの論理的なエントリ数とポート数が物理的な数と一致しない [31]．そこで，以下で述べるモデルを用いることで，マルチバンク化による効果を含んだ見積もりを行った．以下に，このモデルによる各コンポーネントの回路面積を表す式を示す．

$$Area = P_p^2 \times E_p \times C_a \tag{5.1}$$

$$= (W \times C_p)^2 \times (E_l \times C_e) \times C_a \tag{5.2}$$

$$= W^2 \times E_l \times C_{we} \tag{5.3}$$

ここで $Area$ は対象となるコンポーネントの回路面積である．P_p と E_p は，それぞれ物理的なポート数とエントリ数を表す．C_a はコンポーネント毎に決まる定数である，式 5.1 は，回路面積が物理的なポート数の 2 乗とエントリ数に比例することを表す．

式 5.2 は，式 5.1 の物理的なポート数とエントリ数を，処理幅 W と論理的なエントリ数 E_l を用いて表した式である．2.2.2 節で述べたように，各コンポーネントに必要なポート数は，その処理幅 W に比例して決まる．また，マルチバンク化を行う場合，バンクの分割方法が一定であれば，各バンクのポート数はマルチバンク化前のポート数の定数倍となる．したがって，物理的なポート数 P_p は，コンポーネント毎の定数 C_p を用いて $P_p = W \times C_p$ と表すことができる．エントリ数 E_p についても同様であり，バンクの分割方法が一定であれば，論理的なエントリ数 E_l と定数 C_e を用いて $E_p = E_l \times C_e$ と表すことができる．

式 5.3 は，式 5.2 内の定数を 1 つの定数 C_{we} にまとめたものである．この式より，Alpha 21464 の各コンポーネントの面積と処理幅，論理的なエントリ数から，各コンポーネントの C_{we} を求めることができる．Alpha 21464 の各コンポーネントの処理幅と論理的なエントリ数は表 5.2 に示す通りである [31].

以降の評価では，このようにして求めた C_{we} を用い，各コンポーネントの処理幅と論理的なエントリ数を式 5.3 に代入することによって得た回路面積を用いる．この回路面積は，各モデルにおいて，Alpha 21464 と同じようにしてマルチバンク化を行った場合の回路面積であると考えることができる．

5.2.2.1 コンポーネント毎の詳細

本項では，表 5.2 に示す各コンポーネントについて，それぞれのパラメータの詳細について述べる．

- **RMT**
  RMT は整数と浮動小数点のそれぞれに論理レジスタ数と同じ容量が必要となる．このため，その容量を整数と浮動小数点の論理レジスタ数の合計である 64 エントリとした．
- **RF**
  レジスタ・ファイルは In-Order 実行を行う場合でも必要である．このため，Inorder モデルでは，その容量を整数と浮動小数点の論理レジスタ数の合計である 64 エントリとした．
- **フェッチ・ユニットと分岐予測器**
  Alpha 21464 は，44 KB ものテーブルを用いる非常に大きな **2Bc-gskew 分岐予測器**[62] を使用している [63]．このためフェッチ・ユニットの大部分はこの予測器のテーブルによって占められていると予想される．本節の評価では，フェッチ・ユニットの全ての部分が，この予測器のためのテーブルによって占められるものとして評価を行った．
- **命令バッファ**
  Alpha 21464 では，毎サイクル L1 命令キャッシュから 2 つの独立したブロックを読み出し，それらを結合して使用することでフェッチ幅を増加させている [31]．2.2.2 節のフロア・プラン中にある命令バッファはこれを実現するためのバッファである．本節の評価で用いるモデルでは，表 5.2 に示すように，L1 命令キャッシュは 1 ポートであり，このような結合を行わない．このため，上記の命令バッファについては，その回路面積を省略して計算している．

5.2.3 各要素技術を適用した場合の影響

以下に，本節の評価で用いる要素技術について，それぞれの性能への影響と回路面積の削減量についてまとめる．

- **リネームド・トレース・キャッシュ**
  リネームド・トレース・キャッシュは，3.4 節で述べたように，0.4%の性能低下で，RMT の回路面積を 5.1% にまで縮小することが可能である．ただし，パスを格納する必要があるために，トレース・キャッシュの回路面積が 48.1%増加する．
- **NORCS**
  NORCS は，4.5 節で述べたように，2.1%の性能低下で，レジスタ・ファイルの回路面積を 24.9%にまで縮小する．
- **マトリクス・スケジューラ**
  マトリクス・スケジューラは，従来の CAM を用いるウェイクアップ・ロジックの回路面積を 18.6%にまで縮小する [30]．なお，マトリクス・スケジューラは直接的には性能に影響を与えない．
- **NoSQ**
  NoSQ は，ロード・ストア・キューを完全に省略することが可能である [16]．また，NoSQ では従来のロード・ストア・キューと比較してストア命令とロード命令間の値の受け渡しが高速化されるため，2%程度性能が向上する [16]．

以降の評価における OoO-Prop モデルでは，上記の各技術を同時に用いたものとして，各コンポーネントの回路面積を上記で述べた分縮小したものとして計算を行う．また，性能については，それぞれの技術を適用した際の上記の性能向上/低下率をかけあわせた 0.995 倍を一律に用いるものとする．

5.2.4 評価結果

プロセッサの面積効率は，キャッシュの容量や Out-of-Order 実行の有無によって大きく左右される．そこで，まず最初に L2 キャッシュの容量を変えた場合の変化や，Out-of-Order 実行の有無による一般的な差について述べた後，要素技術を含んだプロセッサの面積効率について述べる．

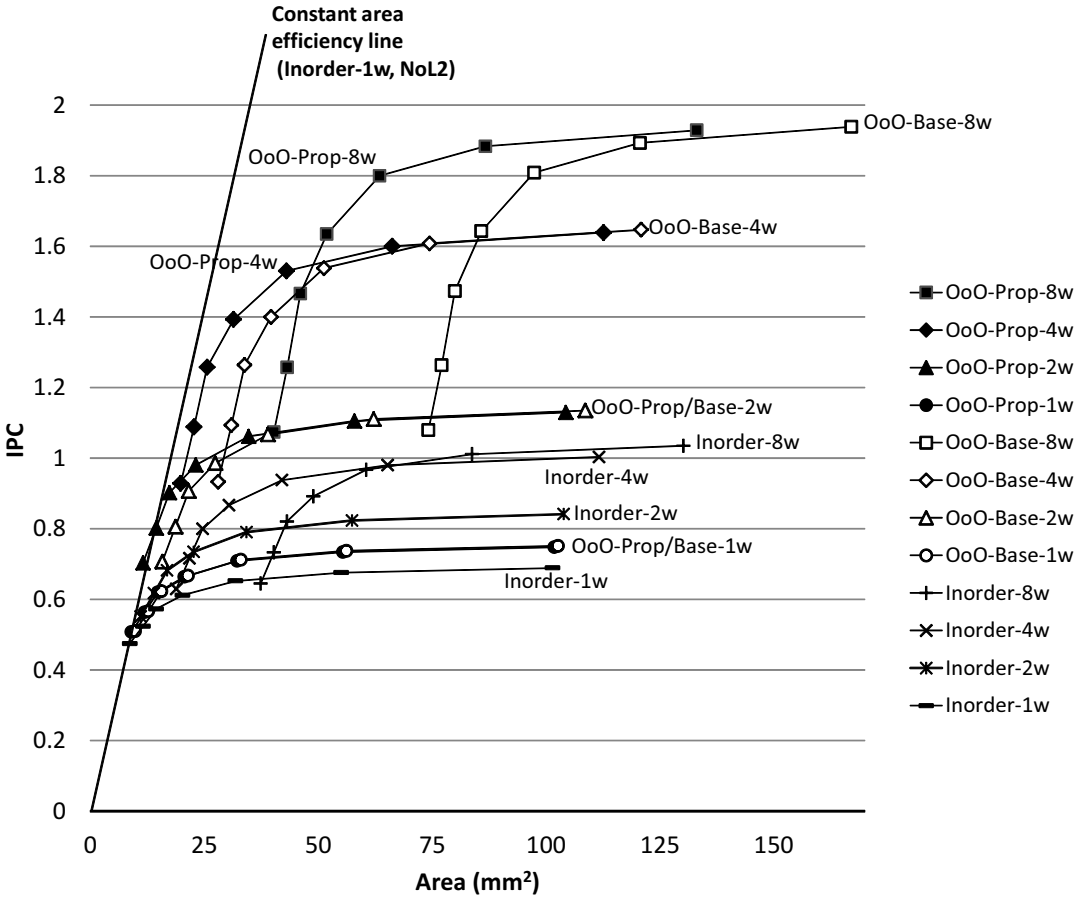


図 5.4: 各モデルにおける面積当たりの IPC

図 5.4 に各モデルの面積あたりの平均 IPC を示す．グラフの横軸は，5.2.2 節で述べたモデルに基づいて求めた回路面積である．縦軸は，SPEC CPU 2006 における全ベンチマーク・プログラムの結果を平均したものである．このグラフでは，面積は小さいほど，IPC は高いほどよいため，グラフ上で左上にある点ほど良い面積効率を持つことを意味する．

各系列のラベルは，モデル名に発行幅を組み合わせた名前となっている．たとえば“OoO-Prop-8w”は，OoO-Prop モデルの 8 命令同時発行を行う構成を表す．折れ線上の各点は，各モデルにおいて L2 キャッシュの容量を変えた場合の結果を表しており，左から順に L2 キャッシュなし，64KB，128KB，256KB，512KB，1MB，2MB の結果を表している．OoO-Prop モデルと OoO-Base モデルの発行幅が同じ系列に関しては，折れ線の上の各点において，それぞれ同じ形のマーカーを使用しており，両者の間では塗りつぶしの有無のみが異なる．OoO-Prop モデルの各点は塗りつぶされたマーカーにより示しており，OoO-Base モデルの各点は塗りつぶされていないマーカーによって示している．

L2 キャッシュによる IPC の変化

L2 キャッシュの容量を 512 KB 以上に増加させた場合，いずれのモデルにおいても IPC はほとんど向上しない．たとえば，L2 キャッシュを 512 KB から 1 MB に増加させた場合，全モデルにおいて最も IPC が向上した OoO-Base-8w であっても，その IPC 向上率は 4.7%に留まる．これに対し，回路面積は著しく増加する．OoO-Base-8w の場合，L2 キャッシュを 512 KB から 1 MB に増加させることで，その回路面積は 23.8%も増加している．

L2 キャッシュの増加による IPC の向上率は，発行幅が広いほど，また InO 実行よりも Out-of-Order 実行を行うものほど高い．L2 キャッシュなしの構成から 64 KB に変更を行った場合，Inorder-1w では IPC が 10.3%向上しているのに対し，OoO-Base-8w では IPC が 17.0%向上している．また，L2 キャッシュなしの構成に対して L2 キャッシュを 512 KB まで増加させた場合，Inorder-1w では IPC が 37.4%向上しているのに対し，OoO-Base-8w では IPC が 67.6%向上している．

Out-of-Order 実行の有無による違い

発行幅を増加させた場合，Out-of-Order 実行を行うモデルでは，In-Order 実行を行うモデルと比べてより IPC が向上する．

発行幅が 1 命令である Inorder-1w と OoO-Base-1w では，両者の性能は大きくは変わらない．512 KB の L2 キャッシュを持つ Inorder-1w と OoO-Base-1w の IPC は，それぞれ 0.652 と 0.712 であり，その差は 9.23%である．これに対し，発行幅が 8 命令である Inorder-8w と OoO-Base-8w では，両者の性能は大きく異なる．512 KB の L2 キャッシュを持つモデルでは，IPC は，それぞれ 0.967 と 1.81 であり，その差は 87.2%にもなる．

In-Order 実行を行うモデルでは，発行幅を 4 命令から 8 命令に増加させても，IPC はほとんど向上していない．これに対し，その回路面積は大幅に増加している．これは主に，発行幅が増加したことにより，レジスタ・ファイルのポート数が増えることで，その回路面積が増大しているためである．

Out-of-Order 実行を行うモデルでは，発行幅を 4 命令から 8 命令に増加させた場合には大きな IPC の向上が認められる．512 KB の L2 キャッシュを持つ OoO-Base-8w と OoO-Base-4w を比較した場合，IPC は 17.6%向上している．

面積効率の向上

図 5.4 のグラフ上の直線は，L2 キャッシュを持たない構成の Inorder-1w の面積効率に対して，各面積における面積効率が一定となるように引かれたものである．この L2 キャッシュを持たない Inorder-1w は，全モデルの中で最も単純な構成であり，要素技術の統合を行っていないモデルの中では最も面積効率が高い．

OoO-Prop-2w や OoO-Prop-4w では，L2 キャッシュを増加させた場合に 256 KB 程度まではこの線に追従して面積効率を保っている．面積効率は同程度であるものの，OoO-Prop モデルでは IPC そのものは Inorder-1w よりも大幅に高い．ほぼ同じ面積効率を持つ Inorder-1w と 128KB の L2 キャッシュを持つ OoO-Prop-2w や OoO-Prop-4w を比較した場合，前者では 90.1%，後者では 165%も IPC は高くなる．

5.2.4.1 面積一定時の性能

本項では各モデルの面積効率について，より詳細に述べる．図 5.5 は，各モデルの面積あたりの IPC を求めた上で，一定の面積（100mm^2）で実現可能な IPC をプロットしたものである．各折れ線は前項の 5.4 節と同じマーカーを使用している．

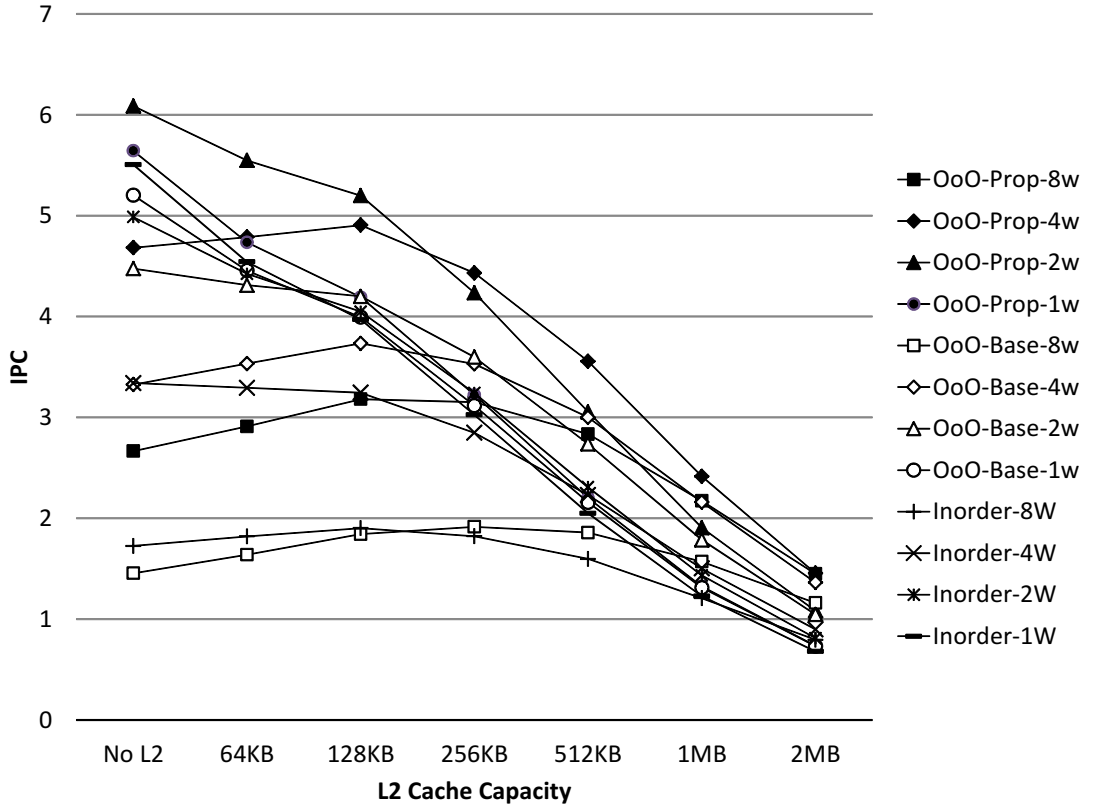


図 5.5: 100mm^2 あたりの IPC

評価を行った全モデルの中で，最も高い面積効率を持つのは，要素技術を統合した OoO-Prop-2w と OoO-Prop-1w である．Inorder-1w と比較して，両者はそれぞれ 10.6%と 2.49%，面積効率が高くなっている．この Inorder-1w は，要素技術の統合を行っていないモデルの中では最も高い面積効率を持つモデルである．また，各 L2 キャッシュの容量毎に面積効率を見た場合，128 KB 以下では OoO-Prop-2w が，それより大きい場合は OoO-Prop-4w の面積効率が最も高い．

L2 キャッシュの容量に対する面積効率の変化はモデル毎に異なる．In-Order 実行を行うモデルや発行幅が 2 命令以下の Out-of-Order 実行を行うモデルでは，基本的に L2 キャッシュを増加させると面積効率は低下する．これに対し，発行幅が 4 命令以上の Out-of-Order 実行を行うモデルでは，面積効率は L2 キャッシュが 128 KB から 256 KB の時に最も大きくなる．L2 キャッシュを増加させていった場合，256 KB を超えたところで Inorder-1w と OoO-Prop-8w の面積効率は逆転し，後者の方の方が高くなる．

表 5.4: Out-of-Order スーパスカラ・プロセッサにおける面積効率の改善率

Issue width	No L2	Best L2	
1w	8.46%	←	(No L2)
2w	36.0%	←	(No L2)
4w	40.8%	31.4%	(128 KB)
8w	83.3%	64.5%	(256 KB)

以上の結果をもとに，OoO-Prop モデルの OoO-Base モデルに対する面積効率の改善率を表 5.4 にまとめる．“No L2” の列は L2 キャッシュを持たない構成の改善率を表す．L2 キャッシュを持たない構成の場合，その回路面積はプロセッサ・コアのみの面積であるため，この列はプロセッサ・コアのみを見た場合の面積効率の改善率であると考えることができる．プロセッサ・コアのみの面積効率の改善率は，8 命令同時発行時において，最大 83.3%になる．

“Best L2” の列は，面積効率が最も高くなる L2 キャッシュの容量と，その際の面積効率の改善率である．単純に面積効率を高めることのみを考えた場合，発行幅が 2 命令以下の構成では L2 キャッシュを積まない方が良い．発行幅が 4 命令，ないしは 8 命令の場合，それぞれ 128 KB，256 KB の L2 キャッシュを搭載した場合に最も面積効率は高くなる．この場合，L2 キャッシュを含んだプロセッサ全体の面積効率は，それぞれ 31.4%と 64.5%と大きく改善している．

5.2.5 評価のまとめ

本節では，L2 キャッシュの容量やプロセッサの構成を変化させた上で，要素技術の統合による面積効率の改善について評価を行った．

L2 キャッシュの増加による IPC の向上率は，発行幅が広いほど，また In-Order 実行よりも Out-of-Order 実行を行う場合ほど高い．ただし，L2 キャッシュは非常に大きな面積を占めるため，面積効率の観点からは一定以下にした方が良い．In-Order 実行を行う場合，L2 キャッシュを増加させた場合の面積あたりの IPC の向上は低く，むしろ全く L2 キャッシュを搭載しない方が面積効率は高い．Out-of-Order 実行を行う場合，発行幅が 4 命令以上の場合は L2 キャッシュの増加にしたがって面積効率が改善し，128 KB から 256 KB 程度の容量を持つ場合に面積効率が最も良くなる．なお，各要素技術は IPC にほとんど影響を与えないため，要素技術の統合の有無とは無関係にこの傾向を持つ．

要素技術の統合により，プロセッサ・コアの面積効率は，8 命令同時発行の場合において最大で 83.3%改善することを確認した．また，要素技術の統合により，既存の構成中では最も面積効率が高い In-Order スカラ・プロセッサと同等以上の面積効率を達成した．要素技術の統合を行った 2 命令同時発行可能な Out-of-Order スーパスカラ・プロセッサの面積効率は，L2 キャッシュを持たない In-Order スカラ・プロセッサと比較して 10.6%高い．また，128 KB の L2 キャッシュを持つ 4 命令同時発行可能な Out-of-Order スーパスカラ・プロセッサの場合，L2 キャッシュを持たない In-Order スカラ・プロセッサと比較して 89.1%の面積効率を持つ．これらの要素技術の統合を行った Out-of-Order スーパスカラ・プロセッサは，面積効率こそ同程度であるものの，その IPC は In-Order スカラ・プロセッサよりも大幅に高く，前者の場合で 90.1%，後者の場合で 165%も IPC は高くなる．

第6章

結論

スーパスカラ・プロセッサを構成する制御部には，一般に，そのウェイ数の 2 乗から 3 乗に比例した回路面積が必要となる．このため，演算器が回路面積全体に占める割合が低下し，その面積効率が大きく低下していた．本稿は，このスーパスカラ・プロセッサの面積効率を高める研究の成果についてまとめたものである．得られた主要な成果は，以下の通りである．

リネームド・トレース・キャッシュ

リネームド・トレース・キャッシュは，レジスタ・リネーミングの結果をキャッシュする手法である．リネームド・トレース・キャッシュでは，キャッシュにヒットし続ける限りにおいてはレジスタ・リネーミングを行う必要はない．レジスタ・リネーミングは，リネームド・トレース・キャッシュにミスした時のみ行われる．この時にリネーミングを行う DMT は，1 サイクルあたり 1 命令程度を処理できれば十分である．このため，通常の RMT と比べると，そのポート数を大幅に少なくすることができる．また，リネームド・トレース・キャッシュからのリネーム結果の読み出しは，リネーム幅とは無関係に 1 ポートで行う事ができる．このため，リネームド・トレース・キャッシュでは，リネーム幅に比例したポートが必要となる通常の RMT とは異なり，回路を巨大化させることなくリネーム幅を広くすることができる．評価の結果，リネームド・トレース・キャッシュは，0.4% の性能低下で，リネーム・ロジックの面積を元の 5.1%にまで縮小する事を示した．

非レイテンシ指向レジスタ・キャッシュ・システム

NORCSは，レジスタ・キャッシュのミスを仮定したパイプラインを持つレジスタ・キャッシュ・システムである．NORCSでは，従来のレジスタ・キャッシュのペナルティを，それよりも発生確率の大幅に低い分岐予測ミス・ペナルティに転化させる．これにより，性能低下をほとんど起こすことなく，メイン・レジスタ・ファイルのポート数を削減する事ができる．評価の結果，NORCSは，2.1%の性能低下で，回路面積と消費電力をパイプライン化されたレジスタ・ファイルに対してそれぞれ24.9%と31.9%にまで削減できる事を示した．単純なLRUポリシによる8エントリのNORCSは，高度な置き換えを行う既存の32エントリのLORCSと同等の性能を持つ．しかし，その回路面積と消費電力は大きく異なり，同構成のNORCSは，LORCSと比べて回路面積と消費電力をそれぞれ27.6%と31.9%にまで減らす事ができる．

これらの研究成果と，これまでに提案されてきた要素技術を統合することにより，スーパスカラ・プロセッサを構成する制御回路の全域にわたって回路面積を大幅に縮小する事が可能になった．評価の結果，提案手法を含む要素技術の統合を行う事により，In-Orderスカラ・プロセッサと同等以上に高い面積効率を保ちながら，それよりも遥かに高いIPCを実現できることを確認した．スーパスカラ・プロセッサを構成する制御部のうち，その回路面積の大きさが問題となるようなものは，もはや残されていない．スーパスカラ・プロセッサの複雑化による面積効率の低下の問題は，解決されたと結論づけることができる．

今後の発展方向としては，リネームド・トレース・キャッシュからのフェッチがOut-of-Orderに行える点の利用があげられる．従来のプロセッサではレジスタ・リネーミングの処理をプログラム・オーダに従って行う必要があるため，フェッチはIn-Orderに行わなければならなかった．これに対し，リネームド・トレース・キャッシュではリネーム済みの命令をキャッシュに格納しているため，パスさえ一致していればフェッチをOut-of-Orderに行う事が可能である．この点を利用することにより，フロントエンドの処理をより簡略化することや，フェッチのスループットの増加，プログラム・オーダ上で離れたブロックのフェッチを独立に行うことなどの様々な発展が実現できるものと期待できる．

謝辞

本論文の執筆にあたり，御指導，御鞭撻を賜った東京大学 五島正裕 准教授 に深謝いたします.

本論文をまとめるにあたり，大変多くの方々からご指導，ご助言，ご協力を頂きました．東京大学 坂井 修一 教授からは，6 年間の長きにわたり，ご指導を賜りました．東京大学 田浦 健次朗 准教授，東京大学 喜連川 優 教授，東京大学 中村 宏 教授，東京大学 藤田 昌宏 教授には，審査において大変有益なご助言を頂きました.

当時研究室の学生であった堀尾 一生氏，倉田 成己氏には，論文の執筆や日々の議論を通じて，多くのご協力を頂きました.

当時研究室の学生であった，一林 宏憲氏，渡辺 憲一氏，亘理 靖展氏には，卒業された後も含め，シミュレータ鬼斬の開発や，評価データの収集など，様々な点において多くのご協力，ご助言を頂きました.

その他，研究室に在籍した多くの皆様には，研究生活を通じて様々なご協力，ご支援を頂きました.

ここに深甚なる謝意を表します.

参考文献

[1] Chip Architect: *The photograph is an excerpt from* `http://www.chip-architect.com/`.

[2] Rixner, S., Dally, W., Khailany, B., Mattson, P., Kapasi, U. and Owens, J.: Register organization for media processing, *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 375–386 (2000).

[3] Thoziyoor, S., Muralimanohar, N., Ahn, J. and Jouppi, N.: CACTI 5.1., Technical report, HP Laboratories (2008).

[4] 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, 情報処理学会論文誌コンピューティングシステム, Vol. 1, No. 2, pp. 22–33 (2008).

[5] Wijeratne, S., Siddaiah, N., Mathew, S., Anders, M., Krishnamurthy, R., Anderson, J., Hwang, S., Ernest, M. and Nardin, M.: A 9GHz 65nm Intel Pentium 4 Processor Integer Execution Core, *Proceedings of the Ineternational Solid-State Circuits Conference*, pp. 353–365 (2006).

[6] Yung, R. and Wilhelm, N. C.: Caching processor general registers, *Proceedings of the International Conference on Computer Design*, pp. 307–312 (1995).

[7] Cruz, J., Gonzalez, A., Valero, M. and Topham, N.: Multiple-Banked Register File Architecture, *Proceedings of the International Symposium on Computer Architecture*, pp. 316–325 (2000).

[8] Butts, J. A. and Sohi, G. S.: Use-Based Register Caching with Decoupled Indexing, *Proceedings of the International Symposium on Computer Architecture*, pp. 302–313 (2004).

[9] 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 回路面積指向レジスタ・キャッシュの評価, 情報処理学会研究報告 2008-ARC-178, pp. 13–18 (2008).

[10] 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 回路面積指向レジスタ・キャッシュ, 先進的計算基盤システムシンポジウム SACSIS 2008, pp. 229–236 (2008).

[11] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System not for Latency Reduction Purpose, *Proceedings of the International Symposium on Microarchitecture*, pp. 301–312 (2010).

[12] Goshima, M., Nishino, K., Nakashima, Y., Mori, S., Kitamura, T. and Tomita, S.: A High-Speed Dynamic Instruction Scheduling Scheme for Superscalar Processors, *Proceedings of the International Symposium on Microarchitecture*, pp. 225–236 (2001).

[13] 五島正裕, 西野賢悟, 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: スーパースケーラのための高速な動的命令スケジューリング方式, 情報処理学会論文誌 : ハイパフォーマンスコンピューティングシステム, Vol. 42, No. SIG 9(HPS 3), pp. 77–92 (2001).

[14] 五島正裕, 西野賢悟, 小西将人, 中島康彦, 森眞一郎, 北村俊明, 富田眞治: 行列に基づく Out-of-Order スケジューリング方式の評価, 情報処理学会論文誌 : ハイパフォーマンスコンピューティングシステム, Vol. 43, No. SIG 6(HPS5), pp. 13–23 (2002).

[15] Sassone, P. G., Rupley, II, J., Brekelbaum, E., Loh, G. H. and Black, B.: Matrix Scheduler Reloaded, *Proceedings of the International Symposium on Computer Architecture*, pp. 335–346 (2007).

[16] Sha, T., Martin, M. and Roth, A.: NoSQ: Store-Load Communication without a Store Queue, *Proceedings of the International Symposium on Microarchitecture*, pp. 285–296 (2006).

[17] Subramaniam, S. and Loh, G. H.: Fire-and-Forget: Load/Store Scheduling with No Store Queue at All, *Proceedings of the International Symposium on Microarchitecture*, pp. 273 –284 (2006).

[18] Gwennap, L.: Intel's P6 uses decoupled superscalar design, *Microprocessor Report*, Vol. 9, No. 2, pp. 9–15 (1995).

[19] Keltcher, C., McGrath, K., Ahmed, A. and Conway, P.: The AMD Opteron processor for multiprocessor servers, *Micro, IEEE*, Vol. 23, No. 2, pp. 66 – 76 (2003).

[20] Yeager, K.: The Mips R10000 Superscalar Microprocessor, *Micro, IEEE*, Vol. 16, No. 2, pp. 28–41 (1996).

[21] Kessler, R.: The Alpha 21264 microprocessor, *IEEE micro*, Vol. 19, No. 2, pp. 24–36 (1999).

[22] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A. and Roussel, P.: The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal*, Vol. 5 (2001).

[23] McFarling, S.: Combining Branch Predictors, Technical report, Digital Equipment Corporation Western Research Lab (1993).

[24] Jimenez, D. and Lin, C.: Dynamic branch prediction with perceptrons, *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 197–206 (2001).

[25] Tune, E., Liang, D., Tullsen, D. M. and Calder, B.: Dynamic Prediction of Critical Path Instructions, *Proceedings of the International Symposium on High-Performance Computer Architecture*, pp. 185–195 (2001).

[26] Fields, B. and Blodik, S. R. R.: Focusing Processor Policies via Critical-Path Prediction, *Proceedings of the International Symposium on Computer Architecture*, pp. 59–70 (2001).

[27] 安藤秀樹: 命令レベル並列処理, コロナ社 (2005).

[28] Asato, C., Montoye, R., Gmuender, J., Simmons, E. W., Ike, A. and Zasio, J.: A 14-port 3.8ns 116-word 64b Read Renaming Register File, *Proceedings of the Ineternational Solid-State Circuits Conference*, pp. 104–105 (1995).

[29] Palacharla, S., Jouppi, N. P. and Smith, J. E.: Quantifying the Complexity of Superscalar Processors, Technical report, University of Wisconsin-Madison (1996).

[30] 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, 博士論文 (2004).

[31] Preston, R., Badeau, R., Bailey, D., Bell, S., Biro, L., Bowhill, W., Dever, D., Felix, S., Gammack, R., Germini, V., Gowan, M., Gronowski, P., Jackson, D., Mehta, S., Morton, S., Pickholtz, J., Reilly, M. and Smith, M.: Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading, *Proceedings of the International Solid-State Circuits Conference*, Vol. 1, pp. 334–472 vol.1 (2002).

[32] Cain, H. and Lipasti, M.: Memory ordering: a value-based approach, *Proceedings of the International Symposium on Computer Architecture*, pp. 90–101 (2004).

[33] Chrysos, G. and Emer, J.: Memory dependence prediction using store sets, *Proceedings of the International Symposium on Computer Architecture*, pp. 142 –153 (1998).

[34] Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L. and Stamm, R. L.: Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor, *Proceedings of the International Symposium on Computer Architecture*, pp. 191–202 (1996).

[35] Marr, D., Binns, F., Hill, D., Hinton, G., Koufaty, D., Miller, J. and Upton, M.: Hyper-Threading Technology Architecture and Microarchitecture, *Intel Technology Journal*, Vol. 6, No. 1, pp. 4–15 (2002).

[36] Monreal, T., Gonzalez, A., Valero, M., Gonzalez, J. and Vinals, V.: Delaying physical register allocation through virtual-physical registers, *Proceedings of the International Symposium on Microarchitecture*, pp. 186–192 (1999).

[37] Balkan, D., Sharkey, J., Ponomarev, D. and Aggarwal, A.: Address-Value Decoupling for Early Register Deallocation, *Proceedings of the International Conference on Parallel Processing*, pp. 337–346 (2006).

[38] Manne, S., Klauser, A. and Grunwald, D.: Pipeline gating: speculation control for energy reduction, *Proceedings of the International Symposium on Computer Architecture*, pp. 132–141 (1998).

[39] Liu, T. and Lu, S.-L.: Performance improvement with circuit-level speculation, *Proceedings of the International Symposium on Microarchitecture*, pp. 348–355 (2000).

[40] 三輪忍, 張鵬, 横山弘基, 堀部悠平, 中條拓伯: キャッシュを用いたレジスタ・マップ表の回路面積削減, 情報処理学会論文誌コンピューティングシステム, Vol. 3, No. 3, pp. 44–55 (2010).

[41] Akkary, H., Rajwar, R. and Srinivasan, S. T.: Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors, *Proceedings of the International Symposium on Microarchitecture*, pp. 423–434 (2003).

[42] Rotenberg, E., Bennett, S. and Smith, J. E.: Trace cache: a low latency approach to high bandwidth instruction fetching, *Proceedings of the International Symposium on Microarchitecture*, pp. 24–35 (1996).

[43] Henry, D. S., Kuszmaul, B. C., Loh, G. H. and Sami, R.: Circuits for Wide-Window Superscalar Processors, *Proceedings of the International Symposium on Computer Architecture*, pp. 236–247 (2000).

[44] Sakai/Goshima Lab: *Processor Simulator Onikiri 2* `http://www.mtl.t.u-tokyo.ac.jp/~onikiri2/`.

[45] Burger, D. and Austin, T. M.: The simplescalar tool set, version 2.0, Technical report, University of Wisconsin-Madison, Computer Sciences Department (1997).

[46] The Standard Performance Evaluation Corporation: *SPEC CPU2000 suite* `http://www.spec.org/cpu2000/`.

[47] Semiconductor Industries Association: *International Technology Roadmap for Semiconductors* `http://www.itrs.net/` (2005).

[48] 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).

[49] 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).

[50] Sakai, S., Yamaguchi, Y., Hiraki, K., Kodama, Y. and Yuba, T.: An Architecture of a Dataflow Single-Chip Processor, *Proceedings of the International Symposium on Computer Architecture*, pp. 46–53 (1989).

[51] Sakai, S., Kodama, Y., Hiraki, K. and Yamaguchi, Y.: Design of the Dataflow Single-Chip Processor EMC-R, *Journal of Information Processing*, Vol. 13, No. 2, pp. 165–173 (1990).

[52] Sakai, S., Kodama, Y. and Yamaguchi, Y.: Prototype Implementation of a Highly Parallel Dataflow Machien EM-4, *Proceedings of the International Parallel Processing Symposium*, pp. 278–286 (1991).

[53] Preston, R. P., Badeau, R. W., Bailey, D. W., Bell, S. L., Biro, L. L., Bowhill, W. J., Dever, D. E., Felix, S., Gammack, R., Germini, V., Gowan, M. K., Gronowski, P., Jackson, D., Mehta, S., Morton, S. V., Pickholtz, J. D., Reilly, M. H. and Smith, M. J.: Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading, *Proceedings of the Ineternational Solid-State Circuits Conference*, pp. 334–335 (2002).

[54] Sprangle, E. and Carmean, D.: Increasing processor performance by implementing deeper pipelines, *Proceedings of the International Symposium on Computer Architecture*, pp. 25–34 (2002).

[55] Palacharla, S. and Smith, J. E.: Complexity-Effective Superscalar Processors, *Proceedings of the International Symposium on Computer Architecture*, pp. 206–218 (1997).

[56] Ahuja, P., Clark, D. and Rogers, A.: The performance impact of incomplete bypassing in processor pipelines, *Proceedings of the International Symposium on Microarchitecture*, pp. 36–45 (1995).

[57] Hennessy, J. and Patterson, D.: *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers (1996).

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

[59] Butts, J. A. and Sohi, G. S.: Characterizing and predicting value degree of use, *Proceedings of the International Symposium on Microarchitecture*, pp. 15–16 (2002).

[60] Aho, A. V., Denning, P. J. and Ullman, J. D.: Principles of Optimal Page Replacement, *Journal of the ACM (JACM)*, Vol. 18, pp. 80–93 (1971).

[61] McLellan, E.: The Alpha AXP architecture and 21064 processor, *Micro, IEEE*, Vol. 13, No. 3, pp. 36–47 (1993).

[62] Seznec, A. and Michaud, P.: De-aliased hybrid branch predictors, Technical report, INRIA (1999).

[63] Seznec, A., Felix, S., Krishnan, V. and Sazeides, Y.: Design tradeoffs for the alpha EV8 conditional branch predictor, *Proceedings of the International Symposium on Computer Architecture*, pp. 295 –306 (2002).

著者発表論文

雑誌論文

[1] Shioya, R., Kim, D., Horio, K., Goshima, M. and Sakai, S.: Low-Overhead Architecture for Security Tag, *IEICE Transactions on Information and Systems*, Vol. E94-D, No. 1, pp. 69–78 (2011).

[2] 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, 情報処理学会論文誌コンピューティングシステム, Vol. 1, No. 2, pp. 22–33 (2008).

[3] 勝沼聡, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: SWIFT: 文字列ごとの情報フロー追跡手法, 情報処理学会論文誌コンピューティングシステム, Vol. 1, No. 2, pp. 261–274 (2008).

[4] 塩谷亮太, ルォン ディンフォン, 入江英嗣, 五島正裕, 坂井修一: マルチコア・プロセッサの不均質共有キャッシュにおける LRU 大域置き換えアルゴリズム, 情報処理学会論文誌コンピューティングシステム, Vol. 48, No. SIG3, pp. 59–74 (2007).

国際会議

[5] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System not for Latency Reduction Purpose, *IEEE International Symposium on Microarchitecture (MICRO 43)*, pp. 301–312 (2010).

[6] Shioya, R., Kim, D., Horio, K., Goshima, M. and Sakai, S.: Low-overhead architecture for security tag, *IEEE International Symposium on Pacific Rim Dependable Computing (PRDC 2009)*, pp. 135–142 (2009).

[7] Li, K., Shioya, R., Goshima, M. and Sakai, S.: String-wise information flow tracking against script injection attacks, *IEEE International Symposium on Pacific Rim Dependable Computing (PRDC 2009)*, pp. 169–176 (2009).

[8] Katsunuma, S., Kurita, H., Shioya, R., Shimizu, K., Irie, H., Goshima, M. and Sakai, S.: Base Address Recognition with Data Flow Tracking for Injection Attack Detection, *IEEE International Symposium on Pacific Rim Dependable Computing (PRDC 2006)*, pp. 165–172 (2006).

口頭発表

[9] 塩谷亮太, 倉田成己, 中島潤, 五島正裕, 坂井修一: Switch-On-Future-Event マルチスレッディング, 先進的計算基盤システムシンポジウム SACSIS 2010, pp. 157–165 (2010).

[10] 堀尾一生, 塩谷亮太, 五島正裕, 坂井修一: 面積効率を指向するプロセッサの設計と実装, 先進的計算基盤システムシンポジウム SACSIS 2010, pp. 339–346 (2010).

[11] 喜多貴信, 塩谷亮太, 五島正裕, 坂井修一: タイミング制約を緩和するクロッキング方式の提案, 先進的計算基盤システムシンポジウム SACSIS 2010, pp. 347–354 (2010).

[12] 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 回路面積指向レジスタ・キャッシュ, 先進的計算基盤システムシンポジウム SACSIS 2008, pp. 229–236 (2008).

[13] 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, 先進的計算基盤システムシンポジウム SACSIS 2008, pp. 245–254 (2008).

[14] 勝沼聡, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: SWIFT: 文字列ごとの情報フロー追跡手法, 先進的計算基盤システムシンポジウム SACSIS 2008, pp. 167–176 (2008).

[15] 金大雄, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 可変長タグをサポートする低オーバヘッド・タグ・アーキテクチャ, 先進的計算基盤システムシンポジウム SACSIS 2008, pp. 177–185 (2008).

[16] 原健太郎, 塩谷亮太, 田浦健二朗: メモリアクセス最適化を適応した汎用プロセッサと Cell の性能比較, 先進的計算基盤システムシンポジウム SACSIS 2008, pp. 157–166 (2008).

[17] 塩谷亮太, ルォン ディンフォン, 入江英嗣, 五島正裕, 坂井修一: マルチコア・プロセッサの不均質共有キャッシュにおける LRU 大域置き換えアルゴリズム, 先進的計算基盤システムシンポジウム SACSIS 2006, Vol. 2006, No. 5, pp. 23–31 (2006).

[18] 勝沼聡, 栗田弘之, 塩谷亮太, 清水一人, 入江英嗣, 五島正裕, 坂井修一: アドレスオフセットに着目したデータフロー追跡による注入攻撃の検出, 先進的計算基盤システムシンポジウム SACSIS 2006, Vol. 2006, No. 5, pp. 515–524 (2006).

[19] 堀部悠平, 三輪忍, 塩谷亮太, 五島正裕, 中條拓伯: 選択的キャッシュ・アロケーション：マルチスレッド環境におけるキャッシュ利用効率の向上手法, 情報処理学会研究報告 2010-ARC-190, No. 1, pp. 1–8 (2010).

[20] 伊藤悠二, 塩谷亮太, 五島正裕, 坂井修一: 最適なロールバック・ポイントを選択するトランザクショナル・メモリ, 情報処理学会研究報告 2010-ARC-190, No. 9, pp. 1–9 (2010).

[21] 有馬慧, 岡田崇志, 塩谷亮太, 五島正裕, 坂井修一: 過渡故障耐性を持つ Out-of-Order スーパスカラ・プロセッサのコミット方式, 情報処理学会研究報告 2010-ARC-190, No. 10, pp. 1–10 (2010).

[22] Kurata, N., Shioya, R., Nakashima, J., Goshima, M. and Sakai, S.: An Improvement of Switch-on-Future-Event Multithreading, 情報処理学会研究報告 2010-ARC-190, No. 27, pp. 1–9 (2010).

[23] 都井紘, 塩谷亮太, 五島正裕, 坂井修一: 文字列ごとの情報フロー追跡手法のＰＨＰへの実装と評価, 情報処理学会研究報告 2010-OS-115, No. 4, pp. 1–11 (2010).

[24] 岡田崇志, 喜多貴信, 塩谷亮太, 五島正裕, 坂井修一: 耐永久故障 FPGA アーキテクチャ, 電子情報通信学会研究報告 CPSY 2010, Vol. 1, pp. 221–222 (2010).

[25] 有馬慧, 岡田崇志, 喜多貴信, 塩谷亮太, 五島正裕, 坂井修一: Out-of-Order スーパスカラ・プロセッサの耐過渡故障方式の改良, 電子情報通信学会研究報告 CPSY 2010, Vol. 1, pp. 223–224 (2010).

[26] 喜多貴信, 塩谷亮太, 五島正裕, 坂井修一: タイミング制約を緩和するクロッキング方式の提案, 情報処理学会第 72 会全国大会, Vol. 1, pp. 239–240 (2010).

[27] 江口修平, 塩谷亮太, 五島正裕, 坂井修一: プロセッサ性能に対する主記憶バンド幅の影響の評価, 情報処理学会第 72 会全国大会, Vol. 1, pp. 179–180 (2010).

[28] 堀尾一生, 塩谷亮太, 五島正裕, 坂井修一: 面積効率を指向するプロセッサの設計と実装, 情報処理学会第 72 会全国大会, Vol. 1, pp. 181–182 (2010).

[29] 横田侑樹, 塩谷亮太, 五島正裕, 坂井修一: 情報漏洩防止プラットフォーム, 情報処理学会第 72 会全国大会, Vol. 3, pp. 629–630 (2010).

[30] 王彦鈞, 堀尾一生, 塩谷亮太, 五島正裕, 坂井修一: リネームドトレースキャッシュアーキテクチャの評価, 情報処理学会第 72 会全国大会, Vol. 1, pp. 191–192 (2010).

[31] 文栄光, 塩谷亮太, 五島正裕, 坂井修一: 情報漏洩防止のためのプラットフォーム認証, 情報処理学会第 72 会全国大会, Vol. 3, pp. 632–633 (2010).

[32] 伊藤悠二, 塩谷亮太, 五島正裕, 坂井修一: 最適なロールバック・ポイントを選択するネスティッド・トランザクショナル・メモリの評価, 情報処理学会第 72 会全国大会, Vol. 1, pp. 187–188 (2010).

[33] 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 繰り返し構造に着目した分岐プレディクションの改良, 情報処理学会第 72 会全国大会, Vol. 1, pp. 213–214 (2010).

[34] 有馬慧, 岡田崇志, 堀尾一生, 喜多貴信, 塩谷亮太, 五島正裕, 坂井修一: 過渡故障耐性を持つ Out-of-Order スーパスカラ・プロセッサの評価, 情報処理学会第 72 会全国大会, Vol. 1, pp. 223–224 (2010).

[35] 都井紘, 塩谷亮太, 五島正裕, 坂井修一: 文字列ごとの情報フロー追跡手法の PHP への実装, 情報処理学会第 72 会全国大会, Vol. 3, pp. 623–624 (2010).

[36] 文栄光, 塩谷亮太, 五島正裕, 坂井修一: 情報漏洩防止のためのプラットフォーム認証, 電子情報通信学会研究報告 CPSY 2009, pp. 13–18 (2009).

[37] 横田侑樹, 塩谷亮太, 五島正裕, 坂井修一: 情報漏洩防止プラットフォーム, 電子情報通信学会研究報告 CPSY 2009, pp. 7–12 (2009).

[38] 伊藤悠二, 塩谷亮太, 五島正裕, 坂井修一: 最適なロールバック・ポイントを選択するネスティッド・トランザクショナル・メモリ, 情報処理学会研究報告 2009-ARC-184, No. 5, pp. 1–11 (2009).

[39] 堀尾一生, 塩谷亮太, 五島正裕, 坂井修一: 面積効率を指向するプロセッサの設計, 情報処理学会研究報告 2009-ARC-184, No. 27, pp. 1–7 (2009).

[40] 喜多貴信, 樽井翔, 塩谷亮太, 五島正裕, 坂井修一: タイミング制約を緩和するクロッキング方式の予備評価, 電子情報通信学会研究報告 CPSY 2009, pp. 61–66 (2009).

[41] 塩谷亮太, 五島正裕, 坂井修一: 分岐プレディシジョン, 情報処理学会研究報告 2008-ARC-179, pp. 67–72 (2008).

[42] 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 回路面積指向レジスタ・キャッシュの評価, 情報処理学会研究報告 2008-ARC-178, pp. 13–18 (2008).

[43] 安藤徹, 塩谷亮太, 五島正裕, 坂井修一: プログラムの繰り返し構造に着目した動的なヘルパースレッディング, 情報処理学会研究報告 2008-ARC-179, pp. 139–144 (2008).

[44] 江口修平, 塩谷亮太, 五島正裕, 坂井修一: プロセッサ性能に対する主記憶バンド幅の影響の評価, 情報処理学会研究報告 2008-ARC-180, pp. 15–20 (2008).

[45] 堀尾一生, 塩谷亮太, 五島正裕, 坂井修一: ツインテール・アーキテクチャの評価, 情報処理学会研究報告 2008-ARC-179, pp. 7–12 (2008).

[46] 樽井翔, 塩谷亮太, 五島正裕, 坂井修一: タイミングフォールト耐性を持つクロッキング方式, 電子情報通信学会研究報告 CPSY 2008-14, pp. 25–30 (2008).

[47] 勝沼聡, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 文字列に着目した情報フロー追跡によるインジェクション攻撃の検出, 組込技術とネットワークに関するワークショップ ETNET 2008, pp. 25–30 (2008).

[48] 喜多貴信, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 予測ミスした命令の実行を継続する投機手法, 情報処理学会研究報告 2008-ARC-178, pp. 7–12 (2008).

[49] 横田侑樹, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 情報漏洩防止のための暗黙的インフォメーションフロー追跡, 情報処理学会第 70 回全国大会, pp. 111–112 (2008).

[50] 入江英嗣, 杉本健, 塩谷亮太, 渡辺憲一, 五島正裕, 坂井修一: パッシブ WAB の改良による低コストなレジスタ書き込みエラー検出手法, 電子情報通信学会研究報告 CPSY 2008-3, pp. 13–18 (2008).

[51] 栗田弘之, 塩谷亮太, 入江英嗣, 五島正裕, 坂井修一: 動的なインフォメーションフロー制御による情報漏洩防止手法, 情報処理学会報告 2007-ARC-172, Vol. 2007, No. 17, pp. 227–232 (2007).

[52] Toi, H., Shioya, R., Goshima, M. and Sakai, S.: Yet Another Taint Mode for PHP, *IEEE International Symposium on Pacific Rim Dependable Computing (PRDC 2010)* (2010)(Poster).

[53] 堀部悠平, 三輪忍, 塩谷亮太, 五島正裕, 中條拓拍: 選択的キャッシュ・ライン・アロケーションによるキャッシュの容量効率向上, 先進的計算基盤システムシンポジウム SACSIS 2010, pp. 121–122 (2010). (ポスター).

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

[55] 江口修平, 塩谷亮太, 五島正裕, 坂井修一: 主記憶バンド幅がプロセッサ性能に与える影響の評価, 先進的計算基盤システムシンポジウム SACSIS 2009, pp. 147–148 (2009). (ポスター).

[56] Kunbo, L., Shioya, R., Goshima, M. and Sakai, S.: String-Wise Information Flow Tracking, *STARC Forum/Symposium* (2008). (ポスター).