Vol.2011-ARC-196 No.25
2011/7/28

レジスタ・リネーミングと ディスパッチ・ネットワークを不要とする トレース・キャッシュ・アーキテクチャ

伊 達 三 雄[†1] 倉 田 成 己[†1] 塩 谷 亮 太[†2]
五 島 正 裕[†1] 坂 井 修 一[†1]

我々の研究室では面積効率の高いスーパスカラ・プロセッサを実現する手法を提案してきた．特に，命令間の依存解析を行うレジスタ・リネーミングに必要なロジックを削減する手法として，Renamed Trace Cache(RTC) を提案した．

RTC は依存解析済みの命令列をトレース・キャッシュに格納し，再利用する．通常のレジスタ・リネーミングでは依存解析結果を再利用することは不可能だが，RTC では，後続の命令が依存元の命令を指定する形式に命令の変換を行うことによって，再利用を可能にしている．RTC ヒット時は，依存解析を行うことなく命令の実行が可能である．ミス時にのみ依存解析を行う．そのときのリネーム幅を最小限にすることで，RMT の面積を大幅に削減することができる．多くの場合は RTC にヒットするため，性能の低下は抑えることができ，面積効率は高くなる．

本稿では RTC を応用し，更にディスパッチされた後の命令列を格納する Dispatched Image Cache(DIC) を提案する．DIC は対応する命令ウィンドウへの分配情報も再利用する．これは，RTC と同様の方法で依存解析を行った命令を，対応する命令ウィンドウに合わせて領域が区切られた DIC にまとめて格納することで可能となる．DIC ヒット時には，得られたイメージをそのまま命令ウィンドウに格納すればよく，レジスタ・リネーミングとディスパッチ・ネットワークの処理を行う必要がない．これらの処理は，ミス時にのみ，小規模のロジックによって時間をかけて行う．そうすることで，性能を落とさずにレジスタ・リネーミングとディスパッチに必要な回路負荷を最小限に抑えることができる．

また DIC では，命令の出現パターンや実行パスの変化によってキャッシュの利用効率が落ちる場合がある．本稿ではキャッシュの利用効率が落ちないような DIC のキャッシュ格納手法の提案・比較を行った．予備評価の結果，高価なキャッシュ格納アルゴリズムを実装することによるキャッシュ利用効率の向上は，そのために必要となる追加ロジックの量に見合っていないという結果が得られた．

1. はじめに

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

コアとして用いられるスーパスカラ・プロセッサの性能を向上させる一次的な方法は，そのウェイ数を増やすことである．しかし一般的なスーパスカラ・プロセッサの制御部 (非演算器部) の回路面積は，ウェイ数に対して 2 乗から 3 乗に比例して増加する[1)]．これは，スーパスカラ・プロセッサの制御部の大部分が RAM/CAM で構成されており，それらの面積がポート数の 2 乗に比例するためである．ウェイ数を増やし多数の命令を同時に実行するためには，RAM/CAM に対して多数のアクセス・ポートが必要となり，回路面積は非常に大きくなってしまう．したがってウェイ数を増加させるとコアの面積効率は低下してしまう．

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

レジスタ・リネーミングは命令間の依存を解析する処理である．リネーミングには，論理レジスタと物理レジスタとの「現在の」マッピングを保持する **RMT (Register Map Table)** が用いられる．詳しくは 2.1 節で述べるが，RMT には通常，1 命令につき 4 本のポートが必要となる．リネーム幅（同時にリネーミングを行う命令数）が 4 であれば，ポート数は 16 にもなる．RAM/CAM の面積はポート数の 2 乗に比例するから，RMT の面積は非常に大きなものとなる．

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University

© 2011 Information Processing Society of Japan

命令ディスパッチは，各命令を対応する命令ウィンドウに分配・格納する処理である．命令ウィンドウは通常，整数，ロード/ストア，浮動小数点といった演算器の種別ごとのサブ・ウィンドウに非集中化される．そのためリネーミング後の命令を適切なサブ・ウィンドウに分配のためのディスパッチ・ネットワークが必要となる．2.2 節で述べるように，ディスパッチ・ネットワークの面積はディスパッチ幅の 2 乗に比例して増加する．

これらロジックはまた，負荷が高く，消費電力と熱の問題も深刻である．これらのロジックは，スーパスカラ・プロセッサのパイプラインのフロント・エンドに属し，すべての命令がこれらのロジックを使用するためである．

本研究室では，これらのうち，リネーミング・ロジックを簡略化する，**Renamed Trace Cache(RTC)** を提案した[2)]．RTC は，端的に言えば,「リネーミング済み」の命令をキャッシュするトレース・キャッシュである．RTC にヒットした場合,「リネーミング済み」の命令が得られるから，そのままディスパッチすればよい．ミスした時にはレジスタ・リネーミングの処理を行ってトレースを構成するが，これは小規模のロジックによって時間をかけて行えばよい．

本稿では，リネーミング・ロジックとディスパッチ・ネットワークを簡略化する手法として **Dispatched Image Cache(DIC)** を提案する．DIC は，端的に言えば，各サブ・ウィンドウにディスパッチされた後の命令列のイメージを格納する命令キャッシュである．DIC では，RTC の考え方をさらに推し進め，依存解析に加え，サブ・ウィンドウへの分配情報もキャッシュする．命令キャッシュは，サブ・ウィンドウに合わせて区切られ，対応するサブ・ウィンドウと直結される．DIC にヒットした場合，得られたイメージをそのままサブ・ウィンドウに格納すればよく，レジスタ・リネーミングとディスパッチを行う必要がない．ミスした時にはレジスタ・リネーミングとディスパッチ・ネットワークの処理を行ってイメージを構成するが，これは小規模のロジックによって時間をかけて行えばよい．その結果，性能を保ちながら，リネーミング・ロジック，および，ディスパッチ・ネットワークに要する回路面積と消費電力を大幅に削減することができる．

本稿の構成は以下の通りである．2 章では一般的なスーパスカラ・プロセッサにおけるレジスタ・リネーミングとディスパッチについて述べる．3 章では先行研究である RTC の実現方を簡単に説明する．4 章では本稿の提案手法である DIC について述べる 5 章で提案手法の予備評価を行う．最後に 6 章で，まとめと今後の方針を述べる．

2. レジスタ・リネーミングとディスパッチ

本章では，一般的なスーパスカラ・プロセッサにおけるレジスタ・リネーミングとディスパッチの役割およびその負荷を明らかにする．

2.1 レジスタ・リネーミング

レジスタ・リネーミングでは命令間の偽の依存を解消し，真の依存を明らかにする．

偽の依存は，異なる変数が同一のレジスタを再利用することで生じる．命令セット・アーキテクチャで指定される**論理レジスタ**の数は限られているため，新しい実行結果は不要になった演算結果を格納しているレジスタを上書きする．このため異なる命令が同じデスティネーション・オペランドを指定しているとき，これらの命令をアウト・オブ・オーダに実行することができない．

したがって偽の依存は，命令ごとに違うレジスタへの書き込みを行うことで解消されるレジスタ・リネーミングは論理レジスタを，実行時に値を保持する**物理レジスタ**にマッピングすることで，この依存を解決する．

このマッピングは，**RMT** と呼ばれる表を用いて行われる．RMT は，論理レジスタ番号と物理レジスタ番号との現在のマッピングを保持する．通常 RMT は RAM によって構成される．この RAM を論理レジスタ番号でアドレッシングして物理レジスタ番号を読み出すことで，論理レジスタ番号と物理レジスタ番号の対応を得る．

レジスタ・リネーミングの負荷

この RMT は非常に高負荷なロジックである．ソース・オペランド 2 つ，デスティネーション・オペランド 1 つのアーキテクチャにおいて，RAM で構成された RMT を用いたレジスタ・リネーミングを考える．1 つの命令をリネームするには,

- デスティネーション論理レジスタに割り当てられていた物理レジスタを解放するための読み出しに 1 ポート
- デスティネーション論理レジスタに新たに割り当てられた物理レジスタ番号の書き込みに 1 ポート
- 2 つのソース論理レジスタに割り当てられている物理レジスタ番号の読み出しに 2 ポート

必要であり，リネームする命令 1 つにつき RMT のポートは 4 本必要である．リネーム幅 4 のスーパスカラ・プロセッサを仮定すると RMT のポート数は 16 となる．RAM/CAM の面積はポート数の 2 乗に比例するから，RMT の回路面積は 16 の 2 乗に比例する非常に大きなものとなる．リネーム幅をさらに増加させると，RMT はデータ・キャッシュ並の

© 2011 Information Processing Society of Japan

大きさとなり現実的ではなくなる．これはフェッチ幅に対して制約を与えており，フェッチ幅を 8 以上にすることを実質的に不可能としている．

また，RMT に対するアクセス頻度は非常に高く，レジスタ・ファイル以上に高頻度にアクセスする．これはリネーミング・ロジックはスーパスカラ・プロセッサのパイプラインのフロント・エンドに属し，レジスタ・ファイルはバック・エンドに属すためである．

レジスタ・ファイルも，概念的には RMT と同様にオペランド 1 つにつき 1 回アクセスを行うが，

- RMT は物理レジスタ解放のための読み出しが必要である
- ソース・オペランドの多くはフォワーディングにより供給される
- 投機ミスのためリネームは行われたが実行はされない命令が存在する

ことが RMT とレジスタ・ファイルの違いである．RMT のような多ポートの RAM に対して高頻度でアクセスを繰り返すと，消費電力や熱の問題も深刻となる．

このように RMT は高面積，高アクセス頻度であるため，レジスタ・リネーミングを省略することができれば，RMT の面積を大幅に削減し，消費電力や発する熱量を削減することができる．

2.2 ディスパッチ

スーパスカラ・プロセッサにおけるディスパッチとは，各命令を対応する命令ウィンドウに分配・格納する処理である．命令ウィンドウは通常，整数，ロード/ストア，浮動小数点といった演算器の種別ごとのサブ・ウィンドウに非集中化される．命令ウィンドウを非集中化することで，ロジックの実行サイズの縮小と，クリティカルパスの分離の効果が得られる[1]．

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

図 1 にディスパッチ幅 4 のときの，スーパスカラ・プロセッサのディスパッチ・ロジックを示す．図 1 ではフェッチ幅 4 として，命令キャッシュから命令をフェッチし，レジスタ・リネーミングを行った後，命令ウィンドウにディスパッチされる回路の様子を表している．ローテータはフェッチされた命令列を並べ直すために存在する．フェッチ直後の命令列がプログラム・オーダを乱すと，依存関係が乱れ正しい実行結果が得られなくなるからである．

図 1 のリネーミング・ロジックから出ている 1 本の線は，リネーム済みの 1 命令を表している．リネーム済みの命令サイズは 60 bit 程度に拡張される．つまりこれら 1 本の線は，

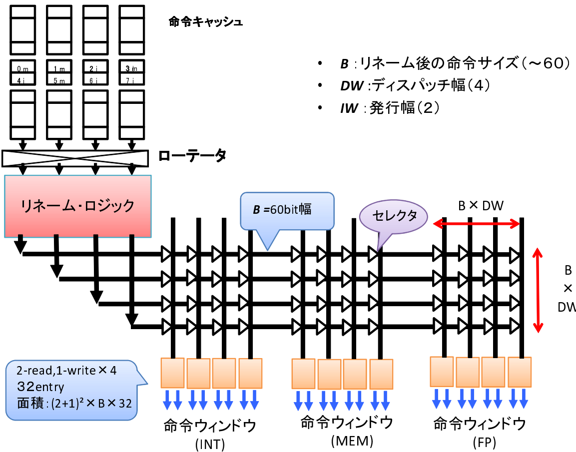

図 1 ディスパッチ・ネットワークの複雑性

それぞれが 60 本程度のビット・ラインをまとめて表していることとなる．

ディスパッチ・ネットワーク

命令毎に格納されるサブ・ウィンドウは異なる．各命令はすべてのサブ・ウィンドウに，セレクタを通して繋がる必要がある．そのためには図 1 のように，格子状のロジックが必要となる．このようなディスパッチのためのロジックを**ディスパッチ・ネットワーク**と呼ぶ．

各サブ・ウィンドウは，基本的にディスパッチ幅分の書き込みポートと，**発行幅**分の読み出しポートを備えた RAM で構成される．実際には，サブ・ウィンドウへの書き込みは連続した領域に書き込まれるため，ランダム・アクセス可能な書き込みポートを 4 つ増やす必要はない．そのため図 1 では，各サブ・ウィンドウを 1 write，2 read の RAM4 つにバンク分けしている．バンク分けされた各サブ・ウィンドウに隙間なく書き込みを行うために，各命令は全てのバンクに繋がらなければならない．そのために，ディスパッチ・ネットワークのすべての交点にセレクタが必要であり，図 1 のような回路になる．

以上のようなディスパッチ・ネットワークの回路面積は，命令ウィンドウの幅と，ディスパッチ幅と，リネーム済み命令のサイズの積で表される．命令ウィンドウの幅もディスパッチ幅に比例するため，ディスパッチ・ネットワークの面積はディスパッチ幅の 2 乗に増加する．各命令のサイズも 60 bit 程度に拡張されているので，ディスパッチ・ネットワークは非常に大きなものとなり，命令ウィンドウのペイロード RAM の 3 倍近い面積となっている．

© 2011 Information Processing Society of Japan

Vol.2011-ARC-196 No.25
2011/7/28

このロジックもリネーミング・ロジックと同様にパイプラインのフロント・エンドに属し，すべての命令がディスパッチ時に利用するため，利用頻度が高く，消費電力や発熱量も大きくなる．このディスパッチ・ネットワークを省略することができれば，レジスタ・リネーミングの省略と同様に，回路面積を大幅に削減でき，消費電力や熱の面からも大きな改善が得られる．

3. Renamed Trace Cache

本章では先行研究である **Renamed Trace Cache(RTC)**[2] について説明する．本稿では RTC の詳細には立ち入らず概略のみを述べる．詳細は 2)，3) を参照していただきたい．

RTC は依存解析を行った後の命令列 (トレース) をキャッシュする**トレース・キャッシュ**[4] である．キャッシュ・ヒット時には「リネーミング済み」の命令列が得られるので，レジスタ・リネーミングの処理を行わずに実行できる．キャッシュ・ミス時にはレジスタ・リネーミングの処理を行ってトレースを構成するが，これはこれは小規模のロジックによって時間をかけて行えばよい．したがってミス時のリネーム幅を最小限に抑えることで，RMT は大幅に簡略化しつつ，性能の低下は防ぐことができる．

3.1 RTC の命令形式

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

通常のレジスタ・リネーミングは，フリーリストからそのとき使用可能な物理レジスタ番号を割り当てる．また命令の実行経路によって，依存する命令が異なることがある．したがってこの方法では，フリーリストの状態や，実行経路によってリネーミング結果は異なってしまい，その結果を再利用することは不可能である．

一方で RTC では依存解析結果を，後続の命令が依存元の命令を指定する形で表す．基本的には，依存元の命令が何命令前に実行されたか，という相対的な距離情報でその指定を行う．この情報は同一の経路を通った場合は常に一定である．さらに経路ごとに区別してキャッシュすることで，依存解析の結果を再利用可能にする．以下ではこの変換形式を **dualflow 形式**と呼ぶ．

図 2 に dualflow 形式に変換された命令の表現形式を例示する．*DstReg* でデスティネーション・オペランドを論理レジスタ番号によって指定する．*SrcL*/*SrcR* でソース・オペランドを指定する．ソース・オペランドは，前述の通り論理レジスタ番号で指定する通常の形式から，「n 命令前」の命令の実行結果として依存元を指定する形式に内部的に変換する．ただし，リタイア済みであることが保証できる命令の実行結果を使用する場合には，ソース・オペランドを論理レジスタ番号のままで指定する．そのため，*RL*/*RR* が 1 のとき *SrcL*/*SrcR* は依存命令への変位を表し，*RL*/*RR* が 0 のとき *SrcL*/*SrcR* は論理レジスタ番号を表すこととする．*imm* は命令の即値フィールドが格納される．

Opcode	DstReg	RL	SrcL	RR	SrcR	Imm

図 2 Dualflow 形式の命令表現

変換前			変換後 (untaken)			変換後 (taken)		
	mov	r1 = ...		mov	r1 = ...		mov	**r1** = ...
	bgt	r1 > 0 then L1		bgt	[-1] > 0 then L1		bgt	[-1] > 0 then L1
	neg	r1 = -r1		neg	**r1** = -[-2]	L1:	add	r3 = r2 + [-2]
L1:	add	r3 = r2 + r1	L1:	add	r3 = r2 + [-1]			

図 3 変換結果の変化

3.2 レジスタのモデル

RTC では物理レジスタ・ファイルと論理レジスタ・ファイルを用いる．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．そうすることで読み出すべき物理レジスタは，命令に割り当てられた物理レジスタのインデックスに変位を加算するだけで決定することができる．また物理レジスタ・ファイルのエントリには，対応する命令のデスティネーション論理レジスタ番号を付随させておく．

命令の実行結果は，実行完了したものから out-of-order に物理レジスタ・ファイルに格納する．その後リタイアする際に，最も古い命令に対応する物理レジスタのエントリから順番に，結果を論理レジスタ・ファイルにコピーする．

3.3 パスの表現

RTC では，命令のソース・オペランドを「n 命令前」と変位で表現する形式に変換する．しかし変換されるの命令までの実行経路によって，ソース・オペランドのプロデューサとなる命令の位置は変化する．**図 3** に例を示す．分岐命令 *bgt* が *untaken* であれば，*add* 命令

© 2011 Information Processing Society of Japan

のソースオペランド *r1* は 1 命令前に依存しているため，変換結果は [−1] となる．*taken* であれば，*neg* 命令は実行されないため *add* の参照距離は [−2] となる．*add* 命令から始まるトレースをキャッシュするとき，*bgt* の結果によって区別してキャッシュしなければならない．

パ　　ス

RTC では実行経路を識別し，経路毎に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．RTC ではパスを用いてトレースを一意に識別する．

パスの表現として，経路上にあるすべての命令アドレスを用いれば十分である．しかしそれでは冗長なので，実行経路の先頭の命令アドレスと，そこからの分岐履歴を用いることとしている[2]．

RTC はこのパス情報をタグアレイとして利用する．このタグを比較することによって，同一の命令アドレスを持つが，変換結果が異なるトレースを識別することができる．

3.4 RTC の効果

RTC により，キャッシュ・ヒット時はリネーミングそのものが省略される．ミス時にのみ，少数の命令ずつ dualflow 形式へ変換を行う．これによって，

- リネーム幅を最小限にすることで RMT に必要な回路面積や消費電力・熱が削減できる
- レジスタ・リネーミングに必要なステージ分だけパイプラインが短縮され，予測ミス・ペナルティが減少する
- 1 ポートで多くの命令列を取り出せる．RMT による制約がなくなるので，フェッチ幅を大きくすることができる

4. Dispatched Image Cache

本章では提案手法である **Dispatched Image Cache(DIC)** について説明する．DIC は各サブ・ウィンドウにディスパッチされた後の命令列のイメージを格納する命令キャッシュである．DIC では，RTC の考え方をさらに推し進め，依存解析に加え，サブ・ウィンドウへの分配情報もキャッシュし再利用する．そうすることで図 1 の複雑なディスパッチ・ネットワークを，**図 4** のように簡略化することができる．

4.1 DIC の構成

DIC の基本的な考え方は，以下の通りである．

- 命令は Dualflow 形式に変換する
- 命令キャッシュの領域を各サブ・ウィンドウに合わせて区切り，対応するサブ・ウィンドウと直結させる
- キャッシュ・ヒット時は，複雑なロジックを介さずに命令列のディスパッチを行うことができる
- キャッシュ・ミス時にのみ，小規模のロジックによって時間をかけてリネーミングとディスパッチを行う
- パスによるトレースの識別は RTC と同様に行う

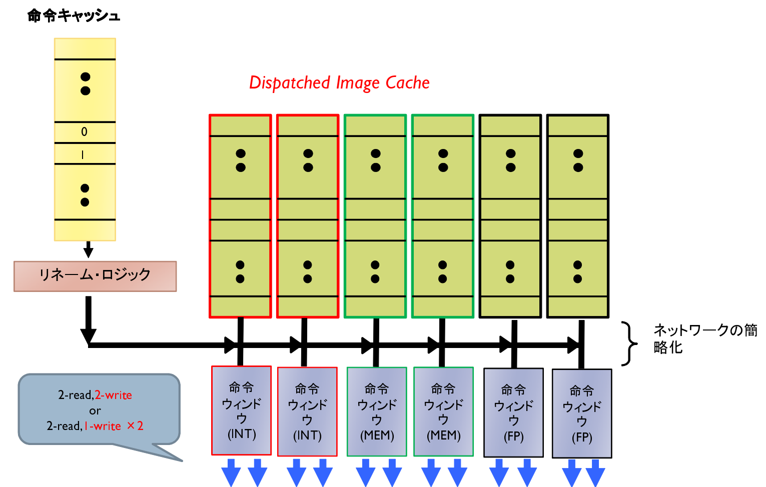

図 4　ディスパッチ・ネットワークの省略

DIC ではサブ・ウィンドウへの分配情報を再利用するために，サブ・ウィンドウに合わせてキャッシュ領域を区切り，対応する領域にのみ命令を格納する．

さらに図 4 では，各命令種別用のキャッシュ領域もバンク分けを行い，2 命令ずつフェッチ(ディスパッチ) できるような DIC の構成をとっている．図 1 と同様にディスパッチ・ネットワークの交点はセレクタを表すが，その数は大幅に削減され，回路面積は大幅に削減されている．

このような DIC を用いることで，DIC ヒット時には，得られたイメージをそのままサブ・ウィンドウに格納すればよく，レジスタ・リネーミングとディスパッチを行う必要がない．

ⓒ 2011 Information Processing Society of Japan

Vol.2011-ARC-196 No.25
2011/7/28

DIC ミス時にのみ，レジスタ・リネーミングとディスパッチ・ネットワークの処理を行ってイメージを構成するが，これは小規模のロジックによって時間をかけて行えばよい．図4では DIC ミス時にフロント・エンドを流れる命令数を 1 としている．そのためリネーミング・ロジックとディスパッチ・ネットワークは，リネーム幅とディスパッチ幅が 1 のときのものへと大幅に縮小されている．これらの結果，性能を保ちながら，リネーミング・ロジック，および，ディスパッチ・ネットワークに要する回路面積と消費電力を大幅に削減することができる．

このように DIC は 1 命令から最大 6 命令のディスパッチ幅を持つ．常に 6 命令ずつディスパッチできれば理想的であるが，命令の出現パターンや，キャッシュ格納アルゴリズムによって DIC の埋まり具合は変わってくる．

たとえば INT 系の命令がしばらく連続した場合は，キャッシュ・ラインあたり 2 命令程度しか埋まらない．また INT 系の命令が 3 命令連続した後に，MEM 系の命令が 3 命令続いた場合は，格納方法によってキャッシュ・ラインあたりの命令数が異なる．このようにキャッシュ格納アルゴリズムを工夫することで，キャッシュの利用効率を高めることができる．キャッシュの利用効率が低ければ，DIC はより大きな容量が必要になる．そうなると DIC によるロジックの減少量以上に，DIC そのものの負荷が大きくなってしまう可能性がある．

以下では，DIC のキャッシュ格納モデルと，そのキャッシュ利用効率について論じる．

4.2 簡単なキャッシュ・ライン形成モデル

図 5 を用いて，比較的ロジックが簡単なキャッシュ・ラインの形成方法の例を説明する．図5 は DIC の内部状態を表しており，各数字は格納されている命令の順番，添え字は命令の種類を表している．0_i から 30_f の命令が順に出現した際に，それぞれのサブ・ウィンドウに合わせて，INT，MEM，FP と領域を区切った DIC に格納する．横一列がキャッシュ・ラインであり，この命令列を 1 つのトレースとする．図5 では {0_i, 1_i, 2_m} や，{3_i, 4_i, 5_m, 6_m, 7_f} が同一キャッシュ・ライン上の命令であることを表す．また，赤線で囲ってある命令 {0 , 27 , 29 } は分岐命令であることを表している．

この手法では以下のようなルールに従ってキャッシュ・ラインを形成する．

- ラインをまたいで命令を追い越さない．すなわち 1 つのキャッシュ領域が埋まれば，同一ラインの他のキャッシュ領域が空いていてもライン形成を完了する
- 同一ラインに 2 命令以上の分岐命令を格納しない

命令を追い越さないとは，図5 の 5_m は 2_m の隣に詰めたりせず，7_f もその上のラインに格納しないということである．後述の手法においてこれらの命令を詰める場合に関して述べるが，そのときフェッチ順とプログラム・オーダが一致しなくなる．したがってプログラム・オーダを再現するためのロジックが追加で必要となり，キャッシュ格納ロジックも複雑になる．この手法では，プログラム順は同一キャッシュ・ライン内のみで覚えておけばよいという点で，簡単である．

同一ラインに 2 命令以上の分岐命令を格納しないとは，27_i の隣に 29_i を格納しないという意味である．これは，通常のトレース・キャッシュのときと同様で，トレース内に複数の分岐命令を含むことによるキャッシュ容量の圧迫や，キャッシュ・ヒット論理の複雑性増加を考慮したものである[4]．

Dualflow 形式の適用

RTC では「n 命令前の命令の実行結果」を表すために $[-n]$ という変換を行った．この手法でも基本的には同様であるが，サブ・ウィンドウがそれぞれ独立であるため少し変更を加える．この手法では，依存元の命令がどのサブ・ウィンドウにいるかを表す情報を図2 の表現形式に付け加える．変換内容は「命令ウィンドウの最後尾の命令から n 命令前」を $[-n]$ と表す．そのために，RL/RR を 2 bit に拡張して，00 のとき $SrcL/SrcR$ は論理レジスタ番号を，01 ，10 ，11 のときは各サブ・ウィンドウ内の変位を表すこととする．

このような形で依存元を指定することで，図5 のモデルの DIC でも依存解析結果を再利用することができる．

キャッシュ利用効率

図5 からもわかるように，この手法を用いたときに命令が格納されていない領域が多く存在する．

ただし，図5 における [×] と [–] では意味合いが異なる．[–] の領域は 2 命令とも命令が埋まっていないため，再利用できる可能性があるが，[×] の領域は片方が埋まっているため再利用できない．再利用できるとは，別のキャッシュ・ラインがその領域を利用できる可能性があるということである．図5 の横一列のキャッシュ・ラインを論理的なものとして見たときに，物理的には別の領域にマッピングすることができ，[–] 領域は無駄にならない可能性がある．

たとえば INT，MEM，FP のメモリ領域を完全に独立に扱い，それぞれが独自にタグアレイを持つ形にすれば，[–] の領域は他のキャッシュ・ラインが自由に利用することができる．これは極端な方法であるが，同様の考え方でこの領域を利用することは難しくない．キャッシュ・ラインが INT 命令のみのときに使う領域と，INT と MEM が混ざっているときに使

© 2011 Information Processing Society of Japan

Vol.2011-ARC-196 No.25
2011/7/28

INT		MEM		FP	
0_i	1_i	2_m	x	-	-
3_i	4_i	5_m	6_m	7_f	x
8_i	9_i	10_m	11_m	-	-
-	-	12_m	13_m	14_f	15_f
18_i	x	-	-	16_f	17_f
-	-	20_m	21_m	19_f	x
-	-	22_m	23_m	-	-
25_i	26_i	24_m	x	-	-
27_i	x	-	-	28_f	x
29_i				30_f	

図 5　ナイーブなキャッシュ・ライン生成方法

用する領域などを分けることでも，論理的な [–] 領域を省略することができる．

一方で [×] 領域を別のキャッシュ・ラインが利用することは難しい．[–] の領域と同様のことを行うためには，はるかに複雑なロジックが必要であり，それを行うことは図 4 のような構成を崩すということでもある．したがって DIC で [×] 領域を再利用することは原理的に不可能である．

今回は，[–] 領域を再利用するための最適な構成方法に関しては評価できなかった．少なくとも [×] の領域を他の命令が再利用することは難しいため，[×] の領域の数を，キャッシュの利用効率を測るうえでのひとつの指針とした．

4.3　アウト・オブ・オーダなキャッシュ・ライン形成モデル

図 6 を用いて，ナイーブなライン形成モデルに対して，命令を詰めて格納する手法を説明する．図 6 は図 5 と同様に DIC の内部状態を表しており，数字は命令の順番を表している．図 6 では $\{0_i,\ 1_i,\ 2_m,\ 5_m\}$ や，$\{3_i,\ 4_i,\ 6_m,\ 10_m,\ 7_f,\ 14_f\}$ が同一のキャッシュ・ラインとなる．

この手法では，先ほどのルールを以下のように変更している．

- M ライン前までに，[×] 領域があれば詰めて格納する
- 同一ラインに 2 命令以上の分岐命令を格納しない．また分岐命令を追い越して詰めない

M は追い越してよいキャッシュ・ラインの数で，図 6 では 3 としている．図 5 と見比べて，5_m を格納する際に，2_m のとなりの [×] となっている領域に 5_m を詰めている．一方で 7_f は，前のキャッシュ・ラインに使いかけの領域 ([×]) がないため詰めない．

この手法では，[×] の領域を極力減らすことを目的としているため，無闇に詰めることに意味はない．[–] の領域に無暗に詰めて [×] 領域を作り出してしまうと，後続の命令が [×] 領域を埋められなくなる可能性があるため，逆効果である．

また分岐命令を追い越して詰めないとは，29_i が分岐命令であるため 30_f は 28_f の隣に詰めない，などとしていることである．

これは分岐予測ミスを考慮しての戦略である．各命令は分岐予測ミスをした際にフラッシュされるべき命令か否か判別できる必要がある．基本的には，分岐命令の後の命令であるか前の命令であるかを表す 1 bit を付け加えればその判断はできる．しかし分岐命令を追い越して詰めた場合，どの分岐の前後であるかを表すために，必要な情報は増えその管理ロジックは複雑になる．ロジックの複雑化に対してキャッシュ利用率の向上が見合わないと判断したため，このモデルでは分岐命令を追い越して詰めないこととした．

Dualflow 形式の適用

この手法であっても基本的な考え方は同じで，依存元の命令が格納されている場所と，その命令ウィンドウの最後尾からの距離を表す形に変換する．ただしこの手法では詰めて格納することによって，プログラム・オーダで後の命令が先にディスパッチされていることがある．そのため初回の変換時と再利用時では，詰めた分だけ誤差が生じる可能性がある．それに対しては，詰めることによってずれた分だけオフセットとして加えることによって正しい変換結果を得ることができる．

図 6 を図 5 と比較すると，無駄になっている [×] 領域は 1 箇所となり，フェッチに必要なサイクル数も 3 サイクル減少している．キャッシュの利用効率という点では，当然こちらの方が優れている．

しかしこの手法では，フェッチ順とプログラム・オーダは一致しないことに注意する必要がある．たとえば，命令 3_i や 4_i がフェッチされるより前のサイクルに命令 5_m がフェッ

© 2011 Information Processing Society of Japan

INT		MEM		FP	
0_i	1_i	2_m	5_m	-	-
3_i	4_i	6_m	10_m	7_f	14_f
8_i	9_i	11_m	12_m	-	-
-	-	13_m	20_m	15_f	16_f
18_i	25_i	21_m	22_m	17_f	19_f
26_i	27_i	23_m	24_m	28_f	x
29_i				30_f	

図 6 アウト・オブ・オーダなキャッシュ・ライン生成方法

チされることとなる．したがって同一ライン内のみの命令順を管理すればよかった図 5 の方法と比べて，プログラム・オーダを再現するために幾つかのロジックを追加する必要がある．

プログラム・オーダの復元

プログラム・オーダを再現するためには，元の順番を表す情報を DIC のデータ・アレイに追加すればよい．しかし単純に各命令に番号を割り当てるのでは追加の情報量が多く，そこからの復元ロジックも高負荷で効率が悪い．

そこで，詰めている命令に，本来の位置からどれだけずれているかという情報を与える．DIC では Dualflow 形式に命令が変換されているため，オペランドの受け渡しは正確に行え，正しい実行結果が得られる．そのため正しい順番で命令のコミットができていれば，正しいプログラム・オーダを再現できる．ずれ情報を用いて本来コミットするタイミングになるように補正をかけることで，正しい順番でコミットを行うことができる．

基本的には以上の方法でプログラム・オーダの復元ができるが，コミット機構を簡単にするために，グループ単位でコミットを行うこともできる[5)]．図 5 の横一列，すなわちプログラム・オーダ順で最大 6 命令を 1 つのコミット・グループとする．グループ内の命令すべてがコミット可能状態になったときにグループでまとめてコミットすることとすれば，各命令キューから先頭の命令を取り出すだけでコミットを行うことができ，ロジックが簡単になる．

Opcode	DstReg	RL	SrcL	RR	SrcR	Imm	Gap	bflag

図 7 Dualflow 形式の拡張

以上のような追加情報を付加した命令の表現形式は，**図 7** のように拡張される．図 2 に加え，*Gap* と *bflag* のフィールドが追加されている．*Gap* には，プログラム・オーダで格納したときに対してどれだけ詰めて格納しているかという変位を格納する．今回の手法では，2 つペアになった命令のうち右側の命令のみ詰める可能性がある．したがってこのビット・フィールドは，各キャッシュ領域に対して 1 つ付け加えればよい．*bflag* には，その命令が分岐命令の前後を表す 1 bit を格納する．

4.4 DIC の効果

DIC ヒット時は，DIC から複雑な論理を一切介さずにサブ・ウィンドウに格納できる．DIC ミス時のフェッチ幅を 1 にするならば，図 1 のようにディスパッチ・ネットワークが大幅に削減できる．同時に RTC と同様の効果も得られ，レジスタ・リネーミングにおける RMT の負荷を大幅に削減することができる．図 1 における命令キャッシュのローテータも不要となっている．また，DIC ヒット時にはリネームやディスパッチに充てられていたパイプラインステージが省略される．パイプラインの段数が短くなるため分岐予測ミス等からの復帰は速くなり，DIC ミス時のフェッチ幅を小さくしたことによる性能低下をある程度打ち消すことができる．

ただし RTC と同様に，DIC ミス時のフェッチ幅が少数であっても性能が落ちないようにするためには，十分な DIC ヒット率が要求される．RTC と同様に，パスによって命令列が区別されるため，通常のトレース・キャッシュと比べてキャッシュの容量を多く消費する．さらに DIC では，命令出現パターンによってもキャッシュ利用効率が低下する．4.2 のモデルに対して，4.3 のモデルのようなキャッシュ格納手法を用いれば利用効率は上昇するが，そのためにはいくつか追加のロジックが必要となる．

5. 評　　価

DIC の性能を計測する予備評価として，キャッシュ格納手法に対して DIC の利用効率がどのように変化するか評価した．利用効率の指針として DIC 内で他の命令が再利用できない領域の数を比較した．すなわち図 5 や図 6 の，[×] 領域の割合を比較した．

© 2011 Information Processing Society of Japan

表 1 プロセッサの構成

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

5.1 評価環境

評価は本研究室で開発したシミュレータ「鬼斬弐」[6) に，キャッシュ格納アルゴリズムを実装して行った．評価したプロセッサのパラメータは**表 1** の通りである．

ベンチマークには，SPECCPU CINT2006[7) のプログラムに含まれる全 12 本のベンチマーク・プログラムを用いた．入力セットには ref を用い，最初の 1G 命令をスキップし直後の 100M 命令を実行した．

また，評価は動的な系列における命令の出現パターンに対して行った．そのため，この結果が DIC のキャッシュ・ヒット率に直結するわけではないが，DIC の性能を決めるその他のパラメータから独立して，格納方法間の効率差のみを比較するうえでは効果的である．

5.2 評価モデル

評価モデルは 4 章の二つのモデル，すなわち命令を追い越して詰めない手法と，キャッシュ・ラインを 3 ラインまで追い越して詰めることを許す手法を用いた．

これら 2 つの手法に加えて，命令を追い越さず，分岐命令でキャッシュ・ラインを区切る格納手法の比較も行った．分岐命令で区切るとは，分岐命令が出現したらキャッシュ・ラインがどんなに空いていても次のキャッシュ・ラインに移ることを意味する．図 5 では，1_i と 2_m を 0_i の隣に格納しないということである．これは，分岐命令のための追加ロジックを最小限にする，最も保守的な格納方法である．

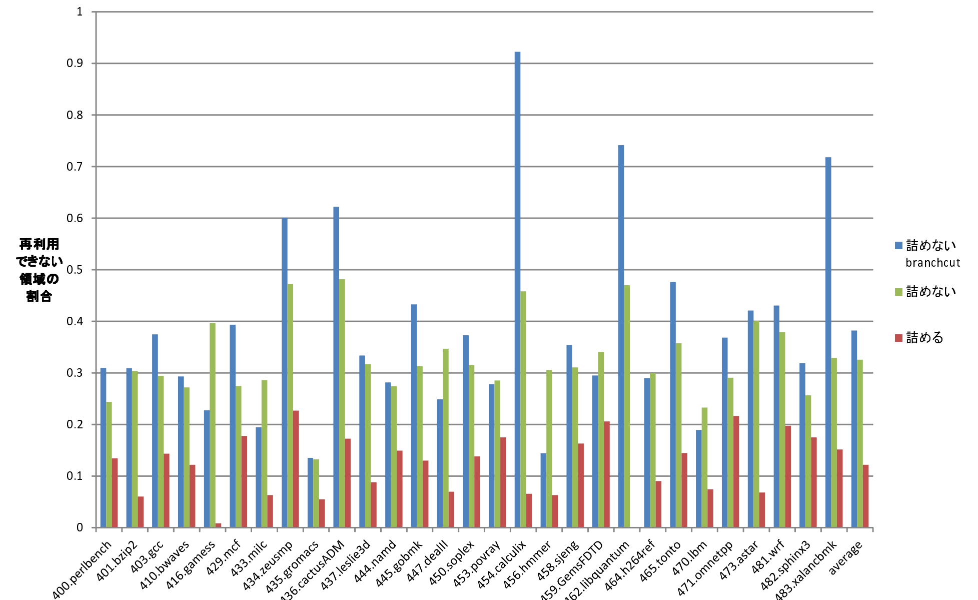

図 8 キャッシュ・格納アルゴリズムの比較

5.3 評価結果

評価結果を**図 8** に示す．縦軸は，命令を格納した領域に対する再利用不可能な領域が発生した割合を表す．全 12 本のベンチマーク・プログラムとそれらの平均値が，キャッシュ格納アルゴリズムによってどのように変化するかを表している．

図 8 のように，ベンチマークの種類によって結果は幅広く異なるが，平均して 3 割を占めていた再利用不可能な領域が，詰めて格納する手法を用いることで 1 割程度まで減少している．

5.4 考察

この評価は動的な出現パターンに対して行っているため，同じ命令列における利用効率の向上を何度もカウントしている．したがって利用効率が改善するループを繰り返すほど，キャッシュ利用効率が改善しているような結果が得られる．

ⓒ 2011 Information Processing Society of Japan

実際の DIC では，それらは単一のキャッシュ領域のみを埋めるので，今回の結果ほどキャッシュ・ヒット率の向上は得られない．このことを考慮すると，詰めない手法で 3 割程度だったものが詰める手法で 1 割程度になるという結果は，そのために必要なハードウェアコストの増加に見合っているとはいえない．

この程度の向上であれば，その他のキャッシュ性能を高める技術を適用するか，単純にキャッシュ容量を増やした方が面積効率は高くなると考えられる．

6. おわりに

本稿はレジスタ・リネーミングとディスパッチ・ネットワークを不要とするスーパスカラ・プロセッサの実現手段として，DIC を提案した．レジスタ・リネーミングとディスパッチ・ネットワークに必要なロジックを大幅に削減しつつ性能を保つことによって，スーパスカラ・プロセッサの面積効率は高くなり消費電力や熱の問題も緩和することができる．

また DIC のキャッシュ・ライン生成アルゴリズムを提案・比較し，キャッシュ利用効率の予備評価を行った．今回の結果から判断すると，空いている領域に命令を詰めてキャッシュ内部までアウト・オブ・オーダにするモデルは，詰めないモデルと比較すると，利用効率の向上量がロジックの増加に見合っていないといえる．

今後は動的な系列に対してではなく，実際に DIC をスーパスカラ・プロセッサに実装した際の性能の評価を行う．また，利用していないキャッシュ領域を他のキャッシュ・ラインが効率良く利用できるような物理的な DIC モデルの提案を行う．

参 考 文 献

1) 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).
2) 入江 英嗣 五島 正裕 坂井修一一林宏憲：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 SACSIS2008, pp.245-254 (2008).
3) 塩谷亮太：面積効率を指向するプロセッサの研究，博士論文，東京大学大学院情報理工学系研究科 (2011).
4) 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).
5) J.M.Tendler, J.S.Dodson, J. J. H.B.: POWER4 system microarchitecture, *IBM J. RES. & DEV. VOL.46 NO.1*, pp.5–25 (2002).
6) 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，pp.120–121 (2009).
7) The Standard Performance Evaluation Corporation: *SPEC CPU2006 suite http://www.spec.org/cpu2006/*.

© 2011 Information Processing Society of Japan