レジスタ・リネーミングとディスパッチ・ネットワークを最小化するプロセッサ・アーキテクチャ

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

我々の研究室では面積効率の高いプロセッサ・アーキテクチャを提案してきた．特にレジスタ・リネーミングに必要な RMT 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 TC に格納し再利用する．依存解析結果の再利用は通常不可能だが，RTC では，後続の命令が依存元の命令を指定する形式に変換することで再利用を可能にしている．この依存解析はミス時にのみ行い，その際のリネーム幅を最小限にすることで RMT の面積を削減できる．本稿では RTC を応用し，更にディスパッチ情報も再利用する Dispatched Image Cache(DIC) を提案する．これは，RTC と同様に依存解析を行った命令を，サブ・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直接ディスパッチできる．ミス時にのみ小規模のロジックで，通常のリネーミングとディスパッチを行う．そうすることで，これらのロジックの負荷を最小限に抑えながらスループット上げることができ，面積効率が向上する．評価の結果，通常のトレース・キャッシュに対する性能低下を 0.4%に抑えつつ，これらの負荷を最小限にできることが確認できた．

Processor Architecture that Minimizes Register Renaming and Dispatch Network

Mitsuo Date,[†] Naruki Kurata,[†] Ryota Shioya,[††]
Masahiro Goshima[†] and Shuichi Sakai[†]

We have proposed some Processor Architectures those improve area-efficiency. Especially, Renamed Trace Cache(RTC) is designed to reduce the cost of RMT, which is used in register renaming. RTC stores renamed instructions and reuses them. The result of conventional renaming can't be reused, so RTC converts instructions into a reusable form that explicitly identifies the producer of the operands. This renaming is needed only on RTC miss. Thus less throughput is necessary for renaming logic, and the cost of RMT can be minimized. In this paper, we propose Dispatched Image Cache(DIC) to reduce the cost of dispatch network by developing the idea of RTC. The cache area of DIC is separated according to sub-windows to reuse the dispatched instructions. On DIC hit, DIC directly feeds instructions to sub-windows. Renaming and dispatch is operated only on DIC miss, and thus less throughput and smaller logic is needed for them. Simulation results shows that performance degradation is smaller than 0.4% when compared with conventional Trace Cache.

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 (回路面積あたりの性能) がより重要な意味を持つ．面積効率の向上は，シングルコア・プロセッサではチップ面積を削減するだけであるが，マルチコア・プロセッサではコア数の増加による最大性能の向上につながるからである．

コアとして用いられるスーパースカラ・プロセッサの性能を向上させる一次的な方法は，そのウェイ数を増やすことである．しかし一般的なスーパースカラ・プロセッサの制御部 (非演算器部) の回路面積は，ウェイ数に対して２乗から３乗に比例して増加する[1)]．これは，スーパースカラ･プロセッサの制御部の大部分が RAM/CAM で構成されており，それらの面積がポート数の２乗に比例するためである．ウェイ数を増やし多数の命令を同時に実行するためには，RAM/CAM

† 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
†† 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University

ⓒ 2012 Information Processing Society of Japan

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ことは，コアの面積効率の低下を招いてしまう．

本稿ではこのようなスーパースカラ・プロセッサの制御部の中で，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．レジスタ・リネーミングには，論理レジスタと物理レジスタとの「現在の」マッピングを保持する **RMT (Register Map Table)** が用いられる．詳しくは 2.1 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 乗に比例するため，RMT の面積は非常に大きなものとなる．

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

これらロジックはまた，消費電力と熱の問題も深刻である．なぜならこれらのロジックは，スーパースカラ・プロセッサのパイプラインのフロント・エンドに属し，利用頻度の高いロジックだからである．面積が大きく利用頻度の高いこれらのロジックは，多くの電力を消費し，熱を発生させる．

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，「リネーミング済み」の命令をキャッシュするトレース・キャッシュであるといえる．RTC にヒットした場合「リネーミング済み」の命令が得られるので，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・ロジックに加え，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といえる．DIC では，RTC の考え方をさらに推し進め，依存解析に加え，サブ・ウィンドウへの分配情報もキャッシュする．そのために，命令キャッシュはサブ・ウィンドウに合わせて領域が区切られ，対応するサブ・ウィンドウと直結される．DIC にヒットした場合，得られたイメージをそのままサブ・ウィンドウに格納すればよく，高負荷なレジスタ・リネーミングとディスパッチを行う必要がない．ミスした時にはレジスタ・リネーミングとディスパッチ・ネットワークの処理を行ってイメージを構成するが，これは小規模のロジックによって時間をかけて行えばよい．その結果，性能を保ちながらリネーミング・ロジック，およびディスパッチ・ネットワークに要する回路面積と消費電力を大きく削減することができる．

本稿の構成は以下の通りである．2 章では一般的なスーパースカラ・プロセッサにおけるレジスタ・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 の実現方法を簡単に説明する．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，一般的なスーパースカラ・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，およびこれら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・リネーミングは命令間の偽の依存を解消し，真の依存を明らかにする．

一般的にレジスタ・リネーミングは，命令セット・アーキテクチャで指定される**論理レジスタ**を，実行時に値を保持する**物理レジスタ**にマッピングすることで依存を解消する．このマッピングは，**RMT** と呼ばれる表を用いて行われる．通常 RMT は RAM によって構成され，論理レジスタ番号と物理レジスタ番号との「現在」のマッピングを保持する．そしてこの RMT は非常に高負荷なロジックとなっている．

レジスタ・リネーミングの負荷

RAM で構成された RMT を用いた一般的なレジスタ・リネーミングを考える．ソース・オペランド 2 つ，デスティネーション・オペランド 1 つを持つ命令をリネームするには，

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

が必要であり，リネームする命令 1 つにつき RMT のポートは 4 本必要である．リネーム幅 4 のスーパースカラ・プロセッサを仮定すると RMT のポート数は 16 となる．RAM/CAM の面積はポート数の 2 乗に比例するので，RMT の回路面積は 16 の 2 乗に比例

ⓒ 2012 Information Processing Society of Japan

する非常に大きなものとなる．

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

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

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

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

2.2　ディスパッチ

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

一般的に命令ウィンドウは整数演算，ロード/ストア，浮動小数点演算といった演算器の種別ごとのサブ・ウィンドウに非集中化される．これは命令ウィンドウを非集中化することで，ロジックの実行サイズの縮小と，クリティカルパスの分離の効果が得られるためである[1]．このサブ・ウィンドウに，対応する命令を分配する処理がディスパッチである．

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

図 1 にディスパッチ幅 4 のときのディスパッチ・ロジックを示す．図 1 はフェッチ幅 4 として，命令キャッシュから命令をフェッチしてから，レジスタ・リネーミングを行い，命令ウィンドウにディスパッチされるまでの一連の回路を表している．図 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 60bit 程度に拡張されるため，これらの線はそれぞれが 60 本程度のビット・ラインをまとめて表していることとなる．

命令キャッシュ内の命令ミックスは，各種類の命令が

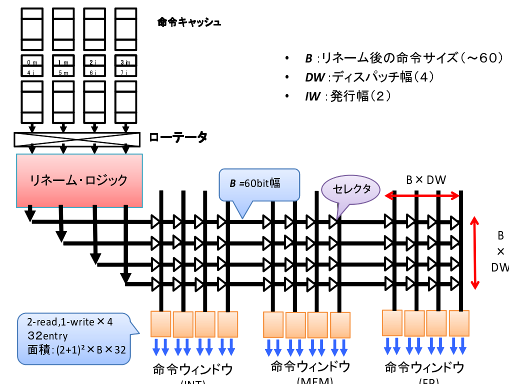

図 1　ディスパッチ・ネットワークの複雑性

不規則に入り混じって格納されている．したがって 4 命令すべてが同一種類の命令であったときにも対応するために，各命令はすべてのサブ・ウィンドウにセレクタを通して繋がる必要がある．そのためには図 1 のように，交点にセレクタを配した格子状のロジックが必要となる．このようなディスパッチのためのロジックを**ディスパッチ・ネットワーク**と呼ぶこととする．

ディスパッチの負荷

各サブ・ウィンドウは，基本的にディスパッチ幅分の書き込みポートと，**発行幅**分の読み出しポートを備えた RAM で構成される．ただし実際には，サブ・ウィンドウへの書き込みは連続した領域に書き込まれるため，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．代わりに図 1 では，各サブ・ウィンドウを 1write，2read × 4 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，ディスパッチ・ネットワークのすべての交点には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，ディスパッチ幅，リネーム済み命令のサイズの積で表される．命令ウィンドウの幅もディスパッチ幅に比例するため，図 1 におけるディスパッチ・ネットワークの面積は $B^2 \times DW^2 \times 3$ で表される．すなわちディスパッチ・ネットワークの面積もディスパッチ幅の 2 乗に比例して増大する．各命令のサイズも 60bit 程度に拡張されているので，ディスパッチ・ネットワークは非常に大きなものとなり，単純に計算すると命令ウィンドウのペイロード RAM の 3 倍近い面積となっている．

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

© 2012 Information Processing Society of Japan

2つのロジックを同時に除去できれば，フロント・エンドのスループットに対する制約が完全になくなる．

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 **Trace Cache(TC)**[4] である．RTC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．RTC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．したがってミス時に必要なリネーム幅を最小限に抑えることで，性能の低下は防ぎつつ，RMTを大幅に簡略化することができる．

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．これは通常のレジスタ・リネーミングでは，同じ命令に対しても異なるマッピング情報を与えることがあるからである．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．また同一の命令に対しても，それまでの実行経路によって依存する命令が異なることがある．これらの理由により通常の方法では，同じ命令に対しても違うリネーミング結果を与えてしまい，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で指定する．*SrcL*/*SrcR* でソース・オペランドを指定する．ソース・オペランドは，前述の通り論理レジスタ番号で指定する通常の形式から,「n命令前」の命令の実行結果として依存元を指定する形式に内部的に変換される．ただし，リタイア済みであることが保証できる命令の実行結果を使用する場合には，ソース・オペランドを論理レジスタ番号のままで指定する．そのため，*RL*/*RR* が1のとき *SrcL*/*SrcR* は依存命令への変位を表し，*RL*/*RR* が0のとき *SrcL*/*SrcR* は論理レジスタ番号を表すこと

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

とする．dualflow形式への変換は通常のレジスタ・リネーミングと同様に行うことができる2)3)．

3.2 RTCのレジスタ・モデル

RTCでは物理レジスタ・ファイルと論理レジスタ・ファイルという2種類のレジスタ・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．この各エントリは，命令に対してシーケンシャルに割り当てられ，各命令のデスティネーション・オペランドと1対1に対応付ける．こうすることで物理レジスタを読み出す際は，自命令に割り当てられた物理レジスタのインデックスに「依存元を指定する」変位を加算するだけで，読み出すエントリを決定できる．また物理レジスタ・ファイルの各エントリには，対応する命令のデスティネーション論理レジスタ番号を付随させておく．

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

3.3 RTCのキャッシュ方法

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し，実行経路毎に区別してキャッシュすると述べた．ここでは，どのようにして経路毎に区別するのかを述べる．

図3 に実行経路によって変換結果が変化する命令列の例を示す．分岐命令 *bgt* が *untaken* であれば，*add* 命令のソースオペランド *r1* は1命令前に依存しているため，変換結果は $[-1]$ となる．*taken* であれば，*neg* 命令は実行されないため *add* の参照距離は $[-2]$ となる．*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

命令の実行経路を識別し，区別してキャッシュするための経路情報を**パス**と呼ぶこととする．RTCではパスを用いてトレースを一意に識別する．パスの表現としては，経路上にあるすべての命令アドレスを用いれば十分ではある．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

ⓒ 2012 Information Processing Society of Japan

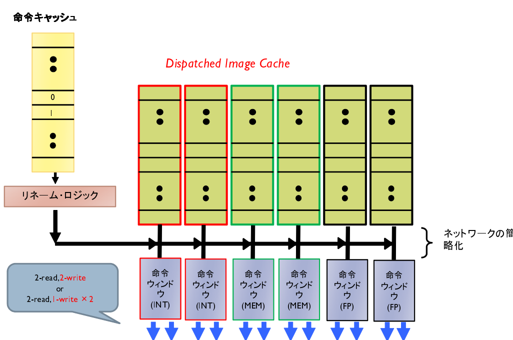

図 4 ディスパッチ・ネットワークの省略

INT		MEM		FP	
0_i	1_i	2_m	x	-	-
3_i	4_i	6_m	7_m	5_f	x
8_i	9_i	10_m	11_m	-	-
-	-	12_m	13_m	14_f	15_f
18_i	19_i	16_m	x	17_f	x
20_i					

図 5 ナイーブなキャッシュ・ライン生成方法

用いることとしている．RTC はこのパス情報をタグアレイとして利用する．命令実行時に PC の履歴と分岐情報を保持しておき，トレースを生成する際に，必要となるパス情報を取り出す．このタグを比較することによって，同一の命令アドレスを持つが，変換結果が異なるトレースを識別することができる．

以上のようにパス情報をタグとして用いるため，通常の TC と比べてタグアレイは大きくなる．また，同一の命令列であっても実行経路によって区別するため，RTC のキャッシュ利用効率は低下することとなる．

3.4 RTC の効果

RTC により，RTC ヒット時はリネーミングが省略される．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すことができ，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 の考え方を応用し，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．そうすることで高負荷な RMT とディスパッチ・ネットワークを最小化することができる．

4.1 DIC の基本的な考え方

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
- 命令格納領域を各サブ・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
- DIC 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
- DIC 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 は以上のような DIC の構成を表す．区切られたキャッシュ領域には対応する種類の命令のみを格納するので，サブ・ウィンドウと直結させることができ，ディスパッチ・ネットワークは大幅に簡略化される．さらに図 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 命令ずつ最大 6 命令同時にフェッチ (ディスパッチ) できるような DIC の構成をとっている．また図 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．これによって図 1 と比べると，セレクタ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．

4.2 キャッシュ・ライン形成モデル

通常の TC や RTC は，実行順にならんだ命令列 (トレース) を，キャッシュ上で連続した領域に保存するが，DIC の場合は必ずしもそのようにできない．これは命令ミックスには偏りがあるからである．INT 系の命令がしばらく連続する場合もあれば，INT 系と MEM 系の命令が交互に出現する場合も考えられる．このような場合，命令の格納方法によっては，DIC 上の命令列は実行順に並んでいないこととなる．

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

簡単なキャッシュ・ライン形成モデル

ディスパッチされた命令を命令順に一定数まとめることが基本的なキャッシュ・ライン形成方法となる．その具体的な例を**図 5** を用いて説明する．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 20_*i* の命令が順に出現した際に，それぞれのサブ・ウィンドウに合わせて，INT，MEM，FP と領域を区切った DIC に格納する．横一列がキャッシュ・ラインであり，この命令列を 1 つのディスパッチ・グループとする．図 5 では {0_i，1_i，2_m} や，{3_i，4_i，6_m，7_m，5_f} が同一キャッシュ・ライン上の命令である

© 2012 Information Processing Society of Japan

ことを表す．実際には連続したキャッシュ・ラインを，DIC 上で物理的に連続したエントリに確保する必要はないが，説明のためこのように表わしている．この手法では

- ラインをまたいで命令を追い越さない

というルールに従ってキャッシュ・ラインを形成している．すなわち 1 種類のキャッシュ領域が埋まれば，同一ラインの他のキャッシュ領域が空いていてもライン形成を完了することとしている．図 5 上では 6_m は 2_m の隣に詰めたりせず，5_f もその上のラインに格納しないということである．もしこれらの命令を詰める場合，フェッチ順とプログラム・オーダが一致しなくなる．そのようにしないこの手法では，同一キャッシュ・ライン内でしかプログラム・オーダが乱れ得ないという点で簡単である．

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．命令ウィンドウを非集中化した際の実装に関しては 3) に詳しく記されているが，DIC でも同様の手法を用いる．端的には，

- 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

という変更を施す．このキャッシュ・ライン形成モデルでも基本的には同様である．そのために，図 2 の RL/RR を 2bit 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．

プログラム・オーダの復元

一般的に，正しいソース・オペランドが供給され，正しい順序でリタイアが行われれば正しい実行結果が得られる．このモデルでは命令を上述の方法で dualflow 形式に変換するので，DIC 上の配置によって依存元の命令情報は失われず，正しいソース・オペランドが得られる．さらに正しい順序でリタイアするためには，なんらかの形でプログラム・オーダを DIC に保持しておけばよい．図 5 のモデルであれば，同一キャッシュ・ライン内で何番目にリタイアするかを表す 3bit を付け加えればよい．さらに図 5 のモデルに

- 同一のデスティネーション・オペランドを持つ命令は同一のキャッシュ・ラインに格納しない

というルールを追加すれば，そのような情報は不要となり，ディスパッチ・グループのリタイア順のみを守ればよくなる[5]．

4.3 キャッシュ利用効率

3.3 節で，RTC ではパスによってトレースを区別するため，キャッシュの利用効率が低下すると述べた．DIC でもそれは同様で，更にキャッシュ・ラインの形成モデルによってもキャッシュの利用効率が低下する．たとえば 4.2 節で例示したモデルでは，図 5 からもわかるように，命令が格納されていない領域が多く存在する．この領域は無駄になっており，命令の出現パターンによってはキャッシュの利用効率が低下する可能性がある．

INT		MEM		FP	
0_i	1_i	2_m	6_m	5_f	14_f
3_i	4_i	7_m	10_m	15_f	17_f
8_i	9_i	11_m	12_m	-	-
18_i	19_i	13_m	16_m	-	-
20_i					

図 6 キャッシュ利用効率の高いキャッシュ・ライン生成方法

DIC の利用効率を高めるためには

- 無駄な領域を生じさせないようなキャッシュ・ライン形成モデルを用いる
- DIC の物理的な構成を調整する
- 格納する命令列を効果的に選択する

などの対処が考えられる．

ライン形成モデルの工夫

キャッシュ・ライン形成する際に”詰めて”格納することでキャッシュの利用効率を高めることができる．たとえば**図 6** のようにキャッシュ・ラインを形成することができれば，高いキャッシュの利用効率が得られる．しかし前述のように，このような格納手法を実現するためには，プログラム・オーダを再現するためのロジックが複雑になる．現状ではこのようなロジックの複雑化に見合うようなモデルは実現できていない．

一例として SPECCPU 2006[6] のベンチマーク・プログラムの動的な命令系列に対して，図 5 の手法と図 6 の手法で無駄になっているキャッシュ領域の割合を比較すると，20%が 10%に向上する程度であった．それに対して，図 6 に Dualflow 形式を適用するためには詰めた分を補正する仕組みが必要となり，リオーダ・バッファの複雑化も招く．10%程度の効率向上に対して，このようなロジックの複雑化は割に合わないと考えている．

キャッシュの物理構成

DIC の物理的な構成を変更することによっても利用効率の低下を防ぐことができる．なぜなら DIC の各キャッシュ領域は対称的である必要はなく，個別に扱うこともできるからである．

簡単な方法として命令出現頻度の偏りに合わせて，各キャッシュ領域の容量に静的に偏りを持たせることができる．これはライン方向，ウェイ (セット) 方向の両方に対して行える．今までの例では，キャッシュ・ラインは (INT, MEM, FP) の命令数を $(2, 2, 2)$ の最大 6 命令で構成していたが，全体的に INT 系の命令が多いのであれば，これを $(3, 2, 2)$ とすることは機

ⓒ 2012 Information Processing Society of Japan

能的に難しくない．

また (INT, MEM, FP) のキャッシュ領域のエントリ数に偏りを持たせることもでき，例えばウェイ数を $(8, 4, 4)$ とすることもできる．INT のキャッシュ領域の半分は，キャッシュ・ラインに INT 命令しか埋まらなかったときに使い，残りの半分は多種類の命令と共有したときに用いる，とすることも難しくない．図5をこのようにした場合，INT の [–] 領域は利用することができ，無駄になる領域は削減される．

以上のような命令出現の偏りがどの程度で，それに対してどのように DIC を構成するのか，といった評価は 5 章で行う．

命令列の選択的格納

DIC に格納する命令列を効果的に選択することでも，利用効率を高めることができる．たとえば，利用頻度の高い命令列 (トレース) を，一度格納されるとほとんど使われることのないトレースが追い出してしまうことは避けたい．それに対しては頻繁に通るパスを特定し，そのパスを通ったトレースのみを格納するなどの対策が考えられる．

また DIC ヒットして欲しいときと，DIC ミスしても影響が大きくないときが存在すると考えられる．基本的に分岐予測ミス後などで命令ウィンドウが空の状態のときには，速く命令ウィンドウを詰めたいので，DIC ヒットして欲しい．このとき当たらなければ，パイプラインが短くなっている恩恵を受けられない．ただし分岐予測ミスをするパスは，頻繁に通らないパスであることも少なくないので，DIC から追い出されてしまいがちだとも考えられる．逆にデータ・キャッシュミスを頻繁に起こし命令ウィンドウを詰まらせる命令の後続のトレースなどの必要度は低いかもしれない．これらの手法に関しては現在考察中である．

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4のようにディスパッチ・ネットワークの回路面積を大幅に削減できる．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．さらに DIC のライン・サイズや (INT, MEM, FP) の比率は，フロント・エンドのロジックの制約を受けずに自由に設定することができ，高いスループットを実現することもできる．

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．しかし 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 TC と比べてキャッシュ・エントリを多く消費する．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可能性もある．

表 1 プロセッサの構成

ISA	Alpha21164A
pipeline stages	Fetch:3,Rename:2,Dispatch:2,Issue:2
fetch width	4 inst.
issue width	Int:2, FP:2, Mem:2
instruction window	Int:32,FP:16, Mem:16
branch predictor	8KB g-share
BTB	2K entries,4way
RAS	8 entries
L1C	32KB,4way,3cycles,64B/line
L2C	4MB,8way,10cycles,64B/line
main memory	200cycles
Trace Cache	2K entries,8way, 4inst./line,branch:1inst.

5. 評　　価

5.1 評 価 環 境

プロセッサ・シミュレータ「鬼斬弐」[7] に，TC と DIC を実装して評価を行った．

評価には SPECCPU 2006[6] に含まれる全 29 本のベンチマーク・プログラムを用いた．入力データ・セットには ref を用い，最初の 1G 命令をスキップし直後の 100M 命令の評価を行った．評価したプロセッサの基本的なパラメータは**表 1** の通りである．

5.2 評価モデル

以下のモデルについて評価を行った．

Base(TC):
評価のベース・ラインとなるモデルで，通常の TC を持つ．各キャッシュ・ラインに最大 4 命令を含み，その内分岐を 1 命令のみ許すようなモデルとした．ただし後方分岐はその時点でトレースを区切ることとした．ここで後方分岐とはプログラム・カウンタが小さくなる方向へ飛ぶ分岐命令を意味しており，後方分岐でトレースを区切らなかった場合，小さなループに対して大量のトレースが生成されることを考慮しての戦略である．また予測が困難な間接分岐命令に関しては，その後続の命令を同一のキャッシュ・ラインに格納しないこととした．これはトレース内に含む分岐命令数が多くなるほど分岐予測が困難になり，生成されるトレースも多くなることへの対策である．

DIC:
DIC を持つモデル．各キャッシュ・ラインの (INT, MEM, FP) の命令数を $(2, 2, 2)$，$(3, 2, 1)$，$(3, 2, 2)$ とした 3 つのモデルを用いた．分岐に関しては，すべての分岐命令の後続の命令は同一の

© 2012 Information Processing Society of Japan

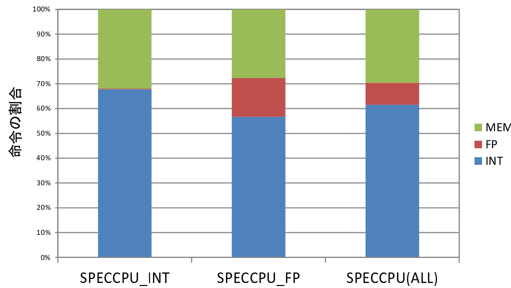

図 7 命令出現頻度の偏り

キャッシュ・ラインに格納しないこととした．これは TC の場合以上に容量が圧迫され，分岐命令を含まないときの方が平均的に高い性能が得られたためである．また，パスに含まれる間接分岐命令の数は 3 命令とした[2),3)]．それ以上の間接分岐命令がパスに含まれるときは，DIC に格納せず，通常の命令キャッシュから取り出すこととなる．このモデルでは DIC にヒットし続ける限りは，リネーミングとディスパッチに複雑なロジックを介さない．また DIC ミス時のリネーム幅，ディスパッチ幅は 1 命令とし，リネーミング/ディスパッチステージを 1cycle とした．

5.3 命令出現頻度の偏り

4.3 節で述べたように，DIC では命令出現パターンの偏りによってキャッシュ利用効率が変化する場合がある．これに対してキャッシュの物理構成を静的に変化させることによって，利用効率の低下を防げる．

そこで，評価プログラムにおける各命令出現の偏りに関する評価を行った．評価は動的な命令系列に対して，各サブ・ウィンドウにディスパッチされる命令数をカウントすることで，全体の命令に対する，INT，MEM，FP の出現割合を計測した．

図 7 にその測定結果を示す．ディスパッチされた全命令に対する INT，MEM，FP の割合を表し，SPECCPU 2006_INT(12 本)，SPECCPU 2006_FP(17 本)，及び SPECCPU 2006 全体 (29 本) のそれぞれの平均値に関して計測結果を示している．

全体平均に対して FP が占める割合は 10% 未満であり，圧倒的に少ないことがわかる．最も FP の占める割合の多かったプログラムは 470.lbm で 60% であったが，その他は FP 系のプログラムでも 20% 程度の低い割合を示した．

今回は一般的な CPU のベンチマーク・プログラムである SPECCPU 2006 の命令ミックスの偏りに合わせて，DIC のキャッシュ・ラインの比率にも偏りを持たせて性能を評価することとした．

5.4 キャッシュ利用率

図 8 にキャッシュ利用率に関して全ベンチマーク・プログラムの平均を比較したグラフを示す．TC と DIC の各 3 モデルずつ計 6 モデルのキャッシュ利用率の比較しており，すべて容量は 4k エントリで揃えている．DIC は上述の 3 つのモデル (DIC:2,2,2)，(DIC:3,2,1)，(DIC:3,2,2) を用い，TC は上述のモデル (TC:bbc) に加え前方分岐も許すモデル (TC:1br)，分岐命令を一切含まないモデル (TC:nbr) を用いた．

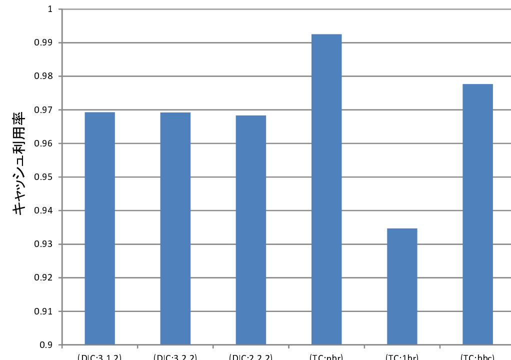

図 8 各モデルのキャッシュ利用率

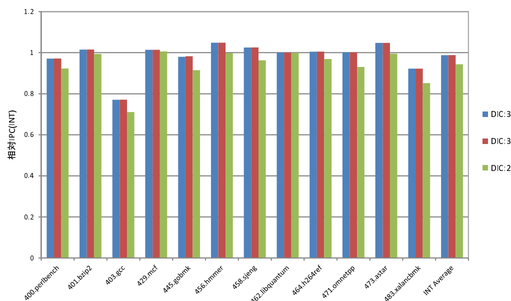

図 9 Base に対する相対 IPC(INT)

利用率は全実行された命令の内，DIC，TC から供給された命令の割合を示す．単純な読み出しを試みた回数に対して，それが成功した回数としてキャッシュ・ヒット率を用いない理由は，DIC と TC 間で多少不公平になると考えたからである．DIC では分岐予測ミスによって結局フラッシュされることとなる命令列は，DIC ミスを引き起こしていることが多い．これは DIC がパスに依存しているからであり，稀なパスを通る際には分岐予測が外れやすく，また DIC も外れやすいためである．そしてこの DIC ミスは性能に悪影響を与えていない．したがって性能上に有意なキャッシュ利用率を図る意味でこのような計測を行った．

4.4 節で，TC と比較すると，DIC はパスによってトレースを区別するため，キャッシュの利用効率は低下すると述べた．図 8 から，分岐を含まない TC に対して DIC(3 つのモデルすべて分岐を含まない) の利用率低下は 2% 程度となっている．それに対して，分岐を含むか否か，特に後方分岐で区切るか否かの利用率への影響は 5% 程度と比較的大きなものとなっている．

5.5 IPC

図 9，**図 10** に，Base モデル (TC:bbc) に対する DIC の相対 IPC を，SPECCPU 2006 の INT 系と FP

© 2012 Information Processing Society of Japan

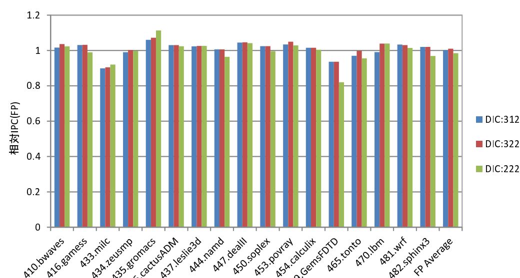

図 10 Base に対する相対 IPC(FP)

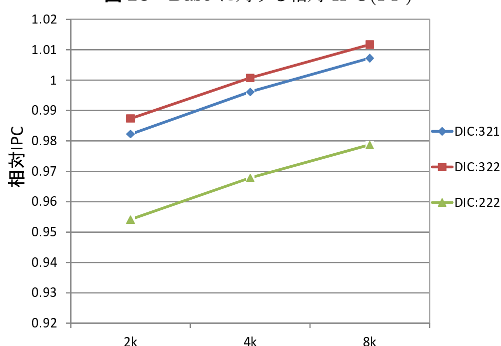

図 11 キャッシュ容量を変化させたときの Base に対する相対 IPC

系で分けて示す．TC，DIC のすべての系列は 8way, 4k エントリの容量を持たせている．

系列毎に大きな傾向の違いはなく，基本的にはキャッシュ・ラインに INT 系命令を多く格納できる方が高い性能が得られ，またキャッシュ・ラインが長いほど性能が上昇している．FP 系のベンチマークの方が系列間の性能差は小さくなり，Base と比べても性能差が小さい．

$(2, 2, 2)$ の系列で最大性能が得られている 435.gromacs では，他系列との DIC ヒット率の差が例外的に大きくなっている．また FP 命令の割合の高い 470.lbm では FP スロット数の差が性能に表れている．一方で TC と比べて全体的に性能差が大きい 403.gcc などは，ワーキングセットが大きく分岐命令も多いため，DIC ヒット率が大幅に低下し性能も低下している．

また，**図 11** は TC，DIC のエントリ数を 2k から 8k に振ったときの平均相対 IPC を表す．各系列は図 9 図 10 と同じキャッシュ・ライン構成である．4k エントリのとき，$(3, 2, 1)$ のモデルで平均 0.4% 程度の性能低下に抑えられている．8k エントリ以上ではこのモデルの TC より DIC の方が高い性能が得られるようになる．

6. おわりに

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

評価の結果，4k エントリ程度で比較すると DIC の性能低下は 0.4% 程に抑えられることがわかった．ただし，DIC は TC と比較すると，パスをタグとして用いる分キャッシュ容量は大きくなる．このことによる面積増加は省略されたロジックの分をある程度打ち消してしまう．今現在の実装では必要とする DIC の容量が大きすぎるため面積削減の効果はあまり期待できない．DIC を導入することによる回路面積や消費電力の評価は今後の課題である．

さらにその他の課題として，より効率的にキャッシュを扱い，面積増加を防ぐ手法を検討していきたい．

参 考 文 献

1) 五島正裕: Out-of-Order ILP プロセッサにおける命令スケジューリングの高速化の研究 (2004).
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, 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) Eric Rotenberg, Steve Bennett, J. E. S.: 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. H. B.: POWER4 system microarchitecture, *IBM J.RES.and DEV. VOL.46 NO.1* (2002).
6) The Standard Performance Evaluation Corporation: *SPEC CPU2006 suite* *http://www.spec.org/cpu2006/*.
7) 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).

© 2012 Information Processing Society of Japan