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　22–33 (Aug. 2008)

逆 Dualflow アーキテクチャ

一 林 宏 憲[†1,*1]　塩 谷 亮 太[†1]　入 江 英 嗣[†1]
五 島 正 裕[†1]　坂 井 修 一[†1]

Out-of-order スーパスカラ・プロセッサにおいて，レジスタ・リネーミングは命令間の依存を解析するために行われる．しかし，レジスタ・リネーミングに用いる RAM である RMT（Register Map Table）は，ポート数が非常に多く遅延が大きい．また，アクセス頻度が高く消費電力も大きい．このため，レジスタ・リネーミングは高価な処理であり，またレジスタ・リネーミングの幅を増やすことも難しい．本研究では，レジスタ・リネーミングを省略する手法として逆 dualflow アーキテクチャを提案する．逆 dualflow アーキテクチャでは，命令のオペランドをプロデューサへの変位に動的に変換してトレースキャッシュに保存・再利用することにより，レジスタ・リネーミングを省略する．その代わり，トレースの種類が増加することによりトレースキャッシュ・ミス率が増加するが，パイプラインが短くなる効果により性能は維持できる．

Anti-dualflow Architecture

Hironori Ichibayashi,[†1,*1] Ryota Shioya,[†1]
Hidetsugu Irie,[†1] Masahiro Goshima[†1]
and Shuichi Sakai[†1]

In out-of-order superscalar processor, register renaming is employed in order to analyze instruction dependence. RMT (Register Map Table), the RAM used in register renaming, is, however, heavily multi-ported and thus suffers from high latency. Additionally it is accessed so frequently that it consumes much energy. Therefore register renaming is very costly and hard to widen. In this paper, we propose a method to eliminate register renaming — anti-dualflow architecture. In anti-dualflow architecture, each operand of an instruction is dynamically converted to the displacement to the producer of the operand, and converted instructions are stored in trace cache and reused, thus eliminating register renaming. The cost is increase in the number of traces and trace cache miss rate, but shorter pipeline keeps performance.

1. はじめに

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

レジスタ・リネーミングでは，論理レジスタと物理レジスタとの「現在の」マッピングを保持する **RMT（Register Map Table）**を読み書きする．RMT を RAM で実装した場合，典型的な 4-way 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 にもなる．このような多ポートの RAM の遅延は配線遅延に支配されているため，LSI の微細化，動作周波数の向上とともに，レジスタ・リネーミングに必要なサイクル数が増加する傾向にある．やや極端な例であるが，Pentium 4 プロセッサでは，レジスタ・リネーミングに 3 サイクルが割り当てられている[1)]．また，フロントエンドの幅を増加させせることは，RMT のポート数がさらに多く必要になるため，非常に困難である．

また，2 章で詳しく述べるように，RMT はアクセス頻度が非常に高い．RMT のサイクルあたりの平均アクセス回数は同じくアクセス頻度の高い要素であるレジスタ・ファイルと比べても 2.3 倍にもなる．このため，RMT の消費電力は容量の割に大きくなってしまう．

レジスタ・リネーミングを行わない命令セット・アーキテクチャとして，五島らの **dualflow アーキテクチャ**[2)–4)] がある．Dualflow アーキテクチャでは，通常の制御駆動型の命令セット・アーキテクチャにあるようなレジスタを定義しない．代わりに，データを生産する命令—**プロデューサ**と，データを消費する命令—**コンシューマ**を指定することで，明示的にデータを受け渡す．

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

しかし，Dualflow アーキテクチャには，命令の配置に関して強い制約がある．Dualflow

†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
∗1 現在，任天堂株式会社
Presently with Nintendo Co., Ltd.

© 2008 Information Processing Society of Japan

アーキテクチャでは，「n 命令後」というふうにコンシューマを指定する．そのため，たとえば，分岐命令を越えてデータを受け渡すには，taken 側と untaken 側でコンシューマの位置を揃えなければならない．適切な位置に有用な命令を配置することができない場合には，転送命令や nop 命令などの本来無用な命令を挿入する必要がある．これらの無用命令のため，通常の制御駆動型の命令セット・アーキテクチャに比べて，命令数が増加してしまっていた．

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

- 命令セット・アーキテクチャは通常の制御駆動型のものとして，dualflow アーキテクチャを内部表現—マイクロアーキテクチャとして利用する．制御駆動型命令セット・アーキテクチャを動的に dualflow アーキテクチャに変換する．
- 変換後の命令は，**トレースキャッシュ**[5] に格納する．

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

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

以下，2 章で，レジスタ・リネーミングの負荷について述べる．次いで，3 章において，提案手法について説明する．4 章で，提案手法の評価を行う．最後に，5 章でまとめと今後の課題を述べる．

2. レジスタ・リネーミング

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

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

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

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

RMT とレジスタ・ファイルのアクセス頻度のシミュレーションを行ったところ，ソース・オペランドの 53%はフォワーディングにより供給され，RMT のサイクルあたり平均アクセス回数はレジスタ・ファイルの 2.3 倍であった．用いたモデルは 4 章のベースモデルである．

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

3. 逆 Dualflow アーキテクチャ

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

逆 dualflow アーキテクチャの基本的な考え方は次のとおりである：

- 命令の実行結果を格納するサイクリックなバッファである**物理レジスタ・ファイル**を用意し，命令の出現した順番と同じ順番で実行結果を格納する．
- 物理レジスタ・ファイルとは別に，論理レジスタの値を保持する**論理レジスタ・ファイル**を用意し，命令の実行結果を in-order に格納する．
- 初回の実行時に命令を変換し，ソース・オペランドを「n 命令前」の実行結果として物理レジスタ・ファイル上での変位で指定するようにする．ただし，3.2 節や 3.4 節で述べるようにリタイア済みであり実行結果が論理レジスタ・ファイルに格納されているこ

© 2008 Information Processing Society of Japan

図 1　Dualflow 形式
Fig. 1　Dualflow format.

とが保証できる命令の実行結果を使用する場合には，ソース・オペランドを論理レジスタ番号のままで指定する．

- 変換した命令をトレースキャッシュにキャッシュ・再利用する．

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

以下，逆 dualflow アーキテクチャについて詳細に述べる．

3.1　命令の内部表現：dualflow 形式

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

以下，この形式を **dualflow 形式**と呼ぶ．**図 1** に，dualflow 形式の例を示す．各フィールドの意味は次のとおりである：

Opcode　*Opcode* は，命令のオペ・コードを示す．

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

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

Imm　*Imm* は，即値を示す．

3.2　Dualflow 形式の命令の実行

Dualflow 形式の命令の実行方法を，**図 2** を用いて説明する．$[-n]$ は，「n 命令前の命令の実行結果」を表す．

まず，各命令に対し，物理レジスタ・ファイルのエントリを命令の出現した順番と同じ順番で割り当てる．順番に割り当てることにより，読み出すべき物理レジスタは，命令に割り当てられた物理レジスタのインデックスに変位を加算するだけで決定することができる．物理レジスタ・ファイルのエントリには，対応する命令のデスティネーション論理レジスタ番

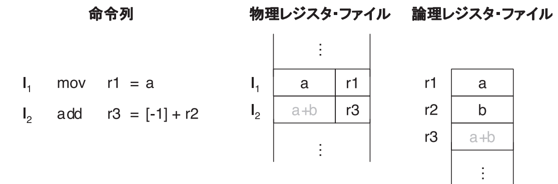


図 2　Dualflow 形式の命令の実行
Fig. 2　Execution of dualflow-form instructions.

号を付随させておく．命令の実行結果は，実行の終わったものから out-of-order に物理レジスタ・ファイルに格納する．その後，最も古い命令に対応する物理レジスタのエントリから順番に，すなわち in-order に実行結果を論理レジスタ・ファイルにコピーする．

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

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

3.3　Dualflow 形式への変換とトレースキャッシュ

Dualflow 形式への変換は，レジスタ・リネーミングと同様に行うことができる．まず，命令にフェッチされた順の通し番号を振る．論理レジスタ番号とそのプロデューサの番号の対応表を用意し，命令がフェッチされるたびに更新する．対応表からソース・オペランド・レジスタに対応する命令の番号を読み出し，変換対象の命令の番号との差をとれば，何命令前の命令の実行結果をソース・オペランドとして用いればよいかが分かる．3.2 節で述べたように，WS 以上前の命令の実行結果を参照する場合には論理レジスタ番号で参照する．

この変換のスループットを**変換幅**と呼ぶことにする．変換幅をフェッチ幅と同じにする場合，結局，この対応表の負荷が RMT と同様に高くなってしまう．よって，変換幅は小さくする必要がある．変換幅によってトレースキャッシュ・ミス時のフロントエンドのスループットは制限されるが，ヒット時にはキャッシュした変換済みの命令を再利用できる．よって，トレースキャッシュに十分にヒットしていれば，変換幅が小さくても性能への影響は少

© 2008 Information Processing Society of Japan

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
Fig. 3　Variation of translated instructions.

ないと考えられる．

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

3.4　変換結果の識別子：パス

逆 dualflow アーキテクチャでは，命令のソース・オペランドを「n 命令前」と変位で表現する形式に変換する．しかし，変換中の命令までに実行した命令によって，ソース・オペランドのプロデューサとなる命令の位置は変化する．**図 3** に例を示す．分岐命令 bgt の成否により，r1 のプロデューサとなる命令も変わり，add 命令の変換結果も異なってしまう．このため，同じアドレスのトレースであっても変換結果のトレースは区別してキャッシュする必要がある．

変換結果のトレースを一意に識別できる情報を**パス** と呼ぶことにする．変換結果はソース・オペランドのプロデューサとなる命令の位置で変化するのであるから，変換結果を一意に識別するためには，トレースにあるソース・オペランドすべてについてそれらのプロデューサの位置が一意に特定できればよい．特定するためには，トレースにある命令のソース・オペランドのプロデューサとなる命令のうち，最も遠くの命令からそのトレースまでに実行した命令列が特定できれば十分である．すなわち，最も遠くのプロデューサを L 命令前とすると，パスは直前に実行した L 個の命令の列である．ただし，詳細は 3.6 節で述べるが，トレースキャッシュのコンフリクト・ミスを軽減するためにパスの長さ L は一定の値 L_{min} を最短とする．

また，3.2 節で述べたように WS 以上前の命令はリタイアしており，それらの命令の実行結果は論理レジスタ番号で参照できるので，パスの長さ L は最長でも $(WS-1)$ でよい．

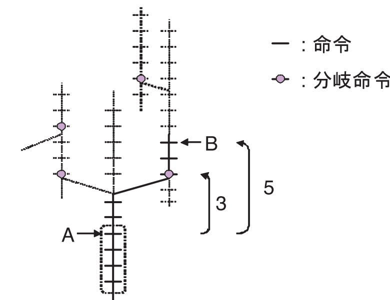


図 4　パスの表現
Fig. 4　Path description.

論理レジスタ番号で指定する場合，そのレジスタのプロデューサの位置が WS 以上前であること，すなわち $(WS-1)$ 前までの命令がそのレジスタに書き込まないことを保証する必要があるため，パスは $(WS-1)$ 必要であることに注意する．

3.5　パスの表現

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

たとえば，**図 4** のように，ある命令 A で始まるトレースに至るパスは複数存在する．

今，ある命令の次の命令について次のことがいえる：

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

よって：

- パスの先頭の命令のアドレス H
- パスの長さ L
- パス内の条件分岐の成否 $pBFlag$
- パス内の間接分岐のターゲット $jTarget$ とその数 $jumpCount$

により，パスを表現することができる．また，これらの情報からこのパスの直後の命令，つまりトレースの先頭アドレスも一意に定まる．たとえば，図 4 の実線のパスは，命令 A か

© 2008 Information Processing Society of Japan

ら 5 命令前の命令 B で始まり，3 命令前の分岐が taken であるという情報により特定することができる．

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

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

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

3.6　トレースのインデックス生成

前述したように，トレースキャッシュのコンフリクト・ミスを軽減するため，パスの長さ L は一定の値 $L_{\min}$ を最短とする．パスの最短の長さを $L_{\min}$ とすると，すべてのパスには $L_{\min}$ 前までの命令が含まれる．このとき，インデックスの生成には，$L_{\min}$ 命令前の命令のアドレス，$L_{\min}$ 前までの条件分岐履歴 $pBFlag$，トレースのアドレスを用いることができる．ただし，3.5 節で述べた間接分岐数の超過によりパスの長さ L が $L_{\min}$ 未満になる場合は，L 命令前の命令のアドレス，L 前までの条件分岐履歴，トレースのアドレスを用いる．

図 5 に，インデックス生成の様子を示す．フェッチあるいはリタイアした命令のアドレスの履歴と分岐履歴を保持しておき，$L_{\min}$ 前までの履歴と，フェッチ PC・トレース PC からインデックスを生成する．

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

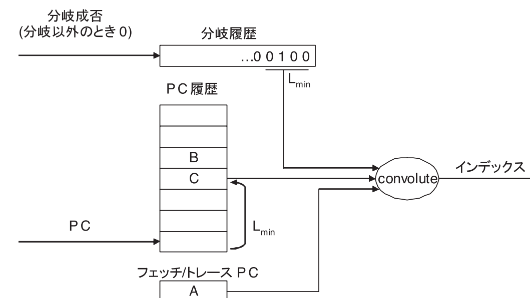


図 5　インデックス生成
Fig. 5　Index generation.

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

3.7　トレースキャッシュのタグ生成とタグ比較

3.4 節で述べたように，変換結果のトレースはパスによって区別する必要がある．よって，トレースキャッシュに格納する際のタグとして，通常のトレースキャッシュでも用いるトレース内の条件分岐の数 $bmask$ と成否 $bflag$ に，3.5 節で述べたパスを合わせたものを用いる．パスの生成は，**図 6** のように行うことができる．まず，やはりリタイアした命令の PC 履歴と分岐履歴を保持しておく．パスの長さ L を，トレース中の各命令がトレースの先頭命令から何命令前の命令を参照しているかを求め，その最大として計算する．PC 履歴と分岐履歴から L 命令前の命令のアドレスと，L 命令前までの分岐履歴を取り出し，トレースのタグとする．

トレースキャッシュから変換済みのトレースをフェッチする際には，通常のタグ比較に加え，タグのパス部分が，現在のフェッチ PC までのパス，すなわちこれまでフェッチした命令の履歴に一致するかどうかの比較が必要である．パス部分の比較は**図 7** のように行う．図の FTA，TA は，トレースの最後が分岐命令であったときの fall-through アドレスと taken

© 2008 Information Processing Society of Japan

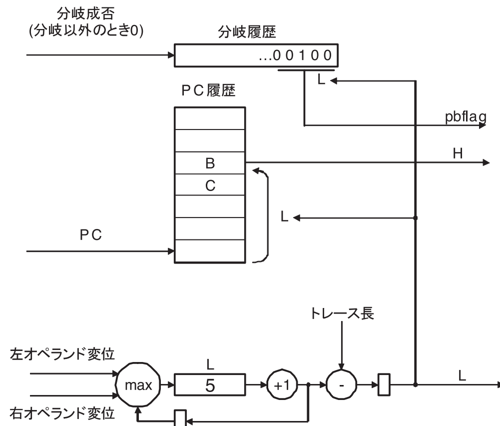


図 6　パス生成
Fig. 6　Path generation.

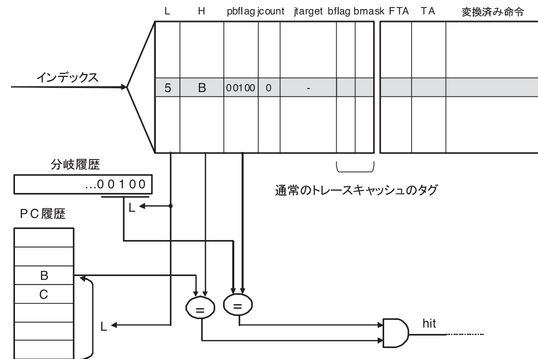


図 7　フェッチ時のタグ比較
Fig. 7　Tag comparison on fetching.

アドレスである．やはりフェッチした命令の PC 履歴，分岐履歴を保持しておく．まず，これらの履歴から，3.6 節で述べた方法でインデックスを生成し，フェッチ対象のトレースが格納されているであろうセットを特定する．このセットのタグ部分をトレースキャッシュから読み出せたら，各ウェイに対して，パスの長さ L を用いてアドレスの履歴から L 命令前の命令のアドレスを読み出し，タグ中のパス先頭アドレス H と比較する．また，条件分岐履歴の直近の L ビット，間接分岐ターゲットの履歴の直近の $jumpCount$ 個を，読み出したタグ中のパスの条件分岐成否 $pBFlag$，$jTarget$ とそれぞれ比較する．すべてが一致すればそれが使用すべき変換済みのトレースである．

このように，タグの比較を行うロジックは通常のトレース・キャッシュに比べて若干複雑になる．また，履歴からタグの比較に用いるパス先頭アドレスの読み出しを行うためには，トレースキャッシュからタグ中のパスの長さ L を読み出せていなければならない．よって，L を格納する部分を分けて配置するなどにより L が早期に得られるような工夫が必要である．また，タグが大きくなるためにトレースキャッシュのヒット・ミスが判明するタイミングは遅くなる．

3.8　逆 dualflow アーキテクチャの効果

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

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

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

4. 評　　価

逆 dualflow アーキテクチャでは，高価な処理であるレジスタ・リネーミングを，dualflow 形式への変換とキャッシングに置き換える．この変換は，トレースキャッシュ・ミス時のみ行われる．また，パスでトレースを区別することによりトレースキャッシュ・ミス率が増加し性能低下の要因になる．よって，特にトレースキャッシュ・ミス率に着目して評価を行った．

以降，まず逆 dualflow アーキテクチャの基本的なパラメータであるパス中の間接分岐数と変換幅に関する評価を行い，その後トレースキャッシュ・ミス率に関する評価を行う．

4.1　評 価 環 境

評価は，本研究室で開発したプロセッサ・シミュレータ「鬼斬弐」[6] を用いたシミュレーションにより行った．ベンチマークには，SPECCPU CINT2000 のプログラムから，164.gzip，175.vpr，176.gcc，181.mcf，186.crafty，197.parser，252.eon，253.perlbmk，254.gap，255.vortex，256.bzip2，300.twolf の 12 本を用いた．入力セットには train を用

© 2008 Information Processing Society of Japan

表 1　ベース・モデルの主要なパラメータ
Table 1　Principal parameters of base model.

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

い，最初の 1 G 命令をスキップし直後の 100 M 命令を実行した．

表 1 にベース・モデルの主要なパラメータを示す．ベース・モデルは Base と呼び，逆 dualflow アーキテクチャは DF と呼ぶ．DF の各モデルは DF-Jj-t-Ww と表す．ただし，j はパスに用いる関節分岐数 J_{max}，t はトレースキャッシュのエントリ数，w はウェイ数である．

通常の命令キャッシュの使用タイミングについては，トレースキャッシュにヒットしている間は通常の命令キャッシュへ同時にアクセスを行わず，トレースキャッシュ・ミスが判明してから通常の命令キャッシュにアクセスするものとした．トレースキャッシュ・ミスが判明するタイミングは，Base ではフェッチの 2 サイクル目，DF では 3.7 節で述べたようにタグ比較が複雑になるためフェッチの 3 サイクル目とした．

DF におけるパスの長さの最小値 L_{min} は 11 とした．変換幅は特に明記しない限り 1 としてある．また，DF では Rename ステージなし（0 サイクル）とした．

参考までに，**図 8** に Base における絶対 IPC を示す．

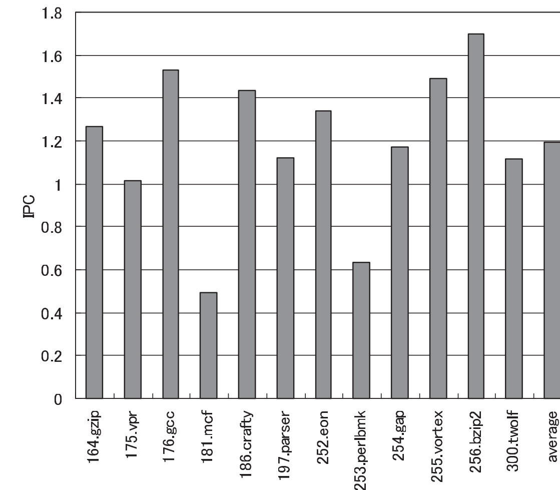


図 8　Base の絶対 IPC
Fig. 8　Absolute IPC of Base.

4.2　パスに用いる間接分岐数

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

そこで，パス情報に用いる間接分岐数 J_{max} の性能への影響を測定した．トレースキャッシュ・ミスの影響を排除するため，DF のトレースキャッシュは，64 k エントリ（1 MB）とした．**図 9**，**図 10** に測定結果を示す．グラフの横軸がベンチマーク（右端は平均）であり，ベンチマークごとの 4 本の棒グラフは，左からそれぞれ，J_{max} が 0，1，2，3 の場合の DF に対応する．図 9 の縦軸は，Base を 1 として正規化した IPC である．図 10 の縦軸は，プログラムの総実行サイクル数に対する，間接分岐数の超過によりフェッチがストールしたサイクルの割合である．

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

© 2008 Information Processing Society of Japan

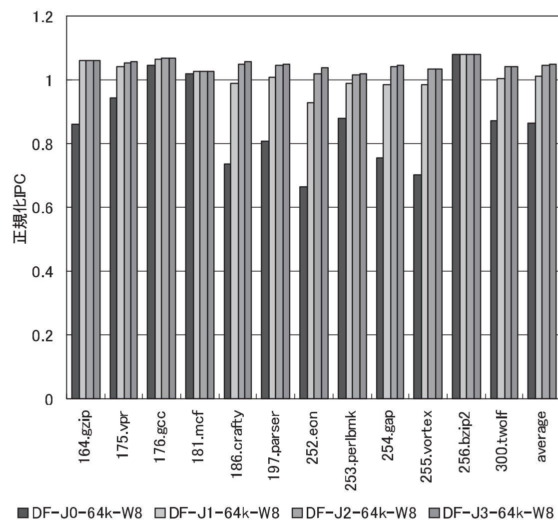


図 9　$J_{\max}$ と，Base=1 とした正規化 IPC
Fig. 9　Normalized IPC (Base=1) vs. $J_{\max}$.

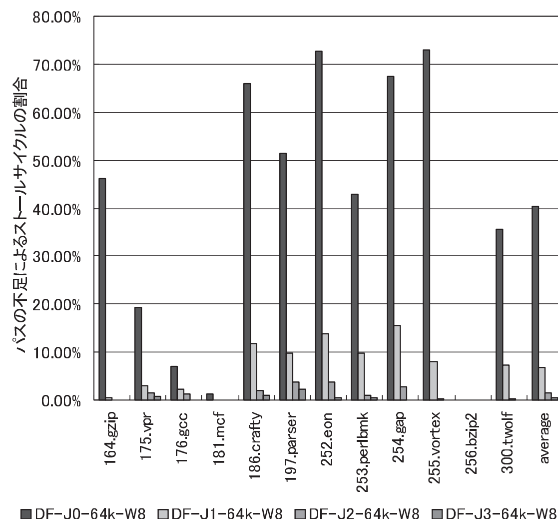


図 10　総実行サイクル数に対する，間接分岐数超過によってフェッチがストールしたサイクル数の割合
Fig. 10　The ratio of fetch-stalled cycles caused by excess of jumps to total execution cycles.

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

$J_{\max}$ が 2 の場合は，フェッチのストールは総実行サイクル数の 1.4%に抑えられており，$J_{\max}$ が 1 の場合に比べて 3.2%IPC が向上している．これに対し，$J_{\max}$ が 3 の場合は $J_{\max}$ が 2 の場合に対して平均 IPC が 0.3%しか向上していない．これは，3 個目の間接分岐ターゲットが格納されないことが多い，もしくはパス情報に間接分岐を 3 個しか含めない場合にパスとして表現できなくなる部分の長さが十分に小さいためと考えられる．

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

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

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

そこで，トレースキャッシュの容量とウェイ数を変化させてトレースキャッシュ・ミス率，IPC を測定した．**図 11**，**図 12** にウェイ数 8 の場合の，**図 13**，**図 14** にウェイ数 4 の場合の測定結果を示す．グラフの横軸がベンチマーク（右端は平均）であり，ベンチマークごとの 4 本の棒グラフは，左からそれぞれ，Base，トレースキャッシュのエントリ数が 2k，4k，8k の場合の DF に対応する．図 11，図 13 の縦軸は，すべてのリタイアした命令に対するトレースキャッシュからフェッチできなかった命令の割合で測ったトレースキャッシュ・ミス率である．図 12，図 14 の縦軸は，それぞれウェイ数 8 の Base，ウェイ数 4 の Base を 1 として正規化した IPC である．

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

一方，ウェイ数が 8 の場合に比べ，ウェイ数が 4 の場合には，Base に対してトレースキャッシュ・ミス率がより高くなってしまっている．トレースキャッシュの容量が 2k エントリでは，Base に対する IPC の低下は 1.7%と大きくなっている．トレースキャッシュの容量を 4 倍の 8k エントリにした場合でさえ，トレースキャッシュ・ミス率は回復せず 4.0%と Base の 2.4 倍にもなっており，IPC 向上率も 3.2%と小さくなっている．このように，DF では Base に比べコンフリクト・ミスが多く起こっていることが分かる．3.7 節で述べたよ

ⓒ 2008 Information Processing Society of Japan

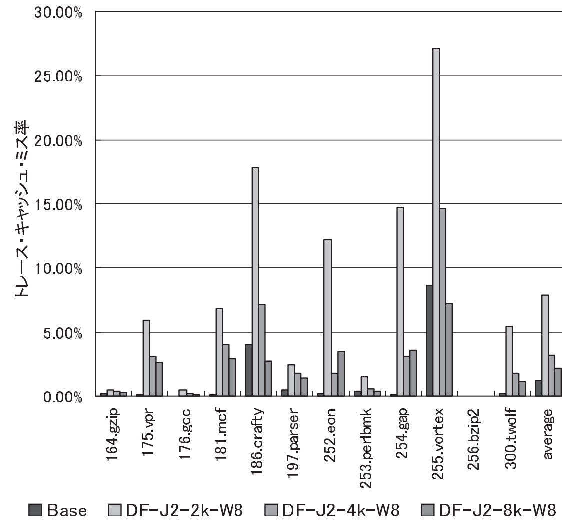


図 11　ウェイ数 8 の場合における DF のトレースキャッシュ容量とミス率
Fig. 11　DF: tracecache miss rate vs. capacity (8-way).

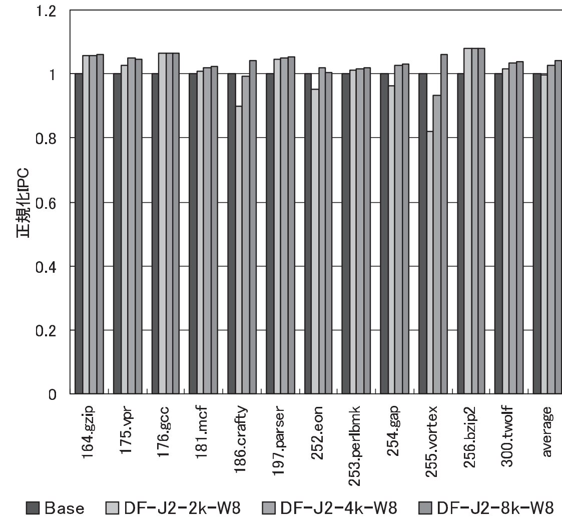


図 12　ウェイ数 8 の場合における DF のトレースキャッシュ容量と Base=1 とした正規化 IPC
Fig. 12　DF: normalized IPC (Base=1) vs. tracecache capacity (8-way).

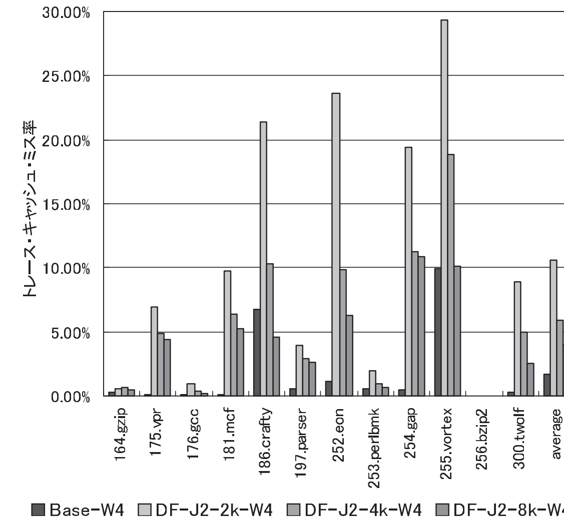


図 13　ウェイ数 4 の場合における DF のトレースキャッシュ容量とミス率
Fig. 13　DF: tracecache miss rate vs. capacity (4-way).

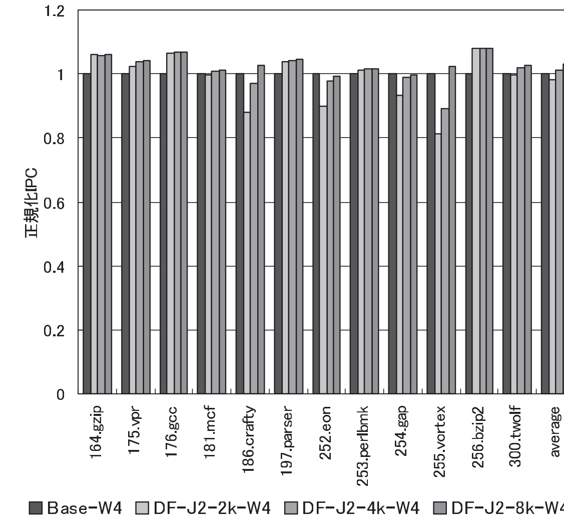


図 14　ウェイ数 4 の場合における DF のトレースキャッシュ容量と Base=1 とした正規化 IPC
Fig. 14　DF: normalized IPC (Base=1) vs. tracecache capacity (4-way).

© 2008 Information Processing Society of Japan

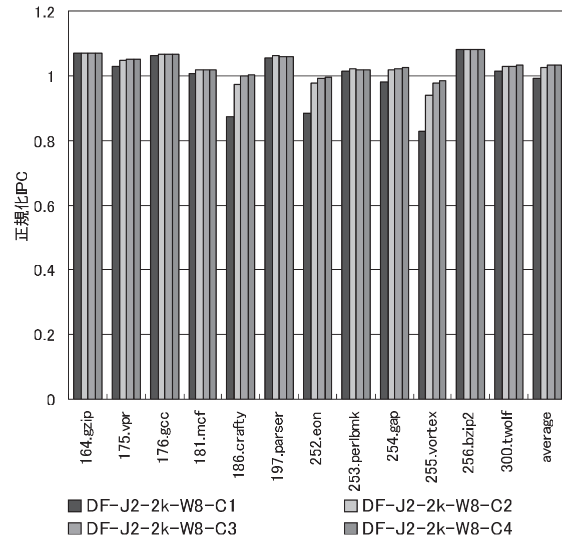


図 15　トレースキャッシュ2k エントリの場合における DF の変換幅と Base=1 とした正規化 IPC
Fig. 15　DF: normalized IPC (Base=1) vs. translation width (2k entries).

うなインデックス生成の工夫を行っても，このようにコンフリクト・ミスが起こりがちであることは問題である．

4.4　変　換　幅

DF-J2-2k-W8 モデルについて，変換幅を変化させて IPC を測定した．**図 15** に測定結果を示す．グラフの横軸がベンチマーク（右端は平均）であり，ベンチマークごとの 4 本の棒グラフは，左からそれぞれ，変換幅 1，2，3，4 の場合の DF-J2-2k-W8 モデルに対応する．トレースキャッシュ・ミス率は，図 11 の DF-J2-2k-W8 モデルである．

3.3 節で述べたように，変換が行われるのはトレースキャッシュ・ミス時のみであるため，トレースキャッシュに十分にヒットしている限りは変換幅が小さくても性能への影響は少ないことが確かめられる．6%前後のトレースキャッシュ・ミス率がある vpr，mcf，twolf においても，変換幅 1 に比べて変換幅 2 の場合で 2%程度の性能向上しかない．しかし，トレースキャッシュ・ミス率が 23–27%と非常に高い crafty，eon，vortex においては，変換幅 1 に比べて変換幅 2 の場合には 10–13%の性能向上が見られ，変換幅を 2 にすると性能低下を最大 6%に抑えることができる．

トレースキャッシュヒット率が十分に高い場合には変換幅は 1 で十分であるが，トレースキャッシュ・ミス率が高い場合には性能低下が大きい．変換幅を 2 にすることによりトレースキャッシュ・ミス率が非常に高い場合でも性能低下を抑えることができるものの，今後の研究によりトレースキャッシュ・ミス率を下げることが課題である．

4.5　典型的な追加ハードウェア量

本節では，逆 dualflow アーキテクチャで追加が必要なハードウェアの量を見積もる．典型的なパラメータとして，4-way のスーパスカラ・プロセッサ，PC を表現するのに必要なビット数 $L_a = 62$，ウィンドウ・サイズ $WS = 32$，物理レジスタ数 $N_p = 64$，トレースの長さ $L_t = 4$，変換幅 $TW = 1$，間接分岐数 $J_{\max} = 2$ を仮定する．また，トレースキャッシュのウェイ数 $W_t = 8$ とする．

削減されるハードウェアは，2 章で述べた RMT であり，整数系・浮動小数点系それぞれ，6 ビット ×32 エントリ，16 ポートの RAM である．

トレースキャッシュには，各命令の各ソース・オペランドに対して論理レジスタ番号で指定することを示すための RL/R と，3.5 節で述べたパスに関するタグが追加される．以下のとおり 1 エントリあたり 171 ビットが追加される：

RL/R　1 命令につき 2 ビット：$2 \times L_t = 8$ ビット
L　$\lceil \log_2 WS \rceil = 5$ ビット
H　$P = 62$ ビット
$pBFlag$　$WS = 32$ ビット
$jTarget$　$P \times J_{\max} = 124$ ビット
$jumpCount$　$\lceil \log_2 J_{\max} \rceil = 2$ ビット
トレースの先頭アドレス　3.5 節で述べたようにパスにより定まるため不要なので，62 ビット削減

また，トレースのフェッチ時，生成時それぞれのための PC 履歴と分岐履歴，間接分岐ターゲット履歴が追加される．フェッチ時の PC 履歴は，ポート数が書き込み 1 ポート，読み出し $W_t = 8$ ポートの合計 9 ポート，$WS \times 2 = 64$ エントリのシフトレジスタで実装できる．分岐履歴は $WS \times 2 = 64$ ビットの RAM，間接分岐ターゲット履歴は $J_{\max} \times 2$ エントリのシフトレジスタで，ポート数はいずれも読み書き 1 ポートずつの 2 ポートである．$WS \times 2$ エントリ用意するのは分岐予測ミス時の回復のためである．フェッチ時のものに比べてリタイア時のものは，いずれもエントリ数が $WS = 32$ エントリになり，PC 履歴はポート数が読み書き 1 ポートずつになる．

© 2008 Information Processing Society of Japan

5. まとめと今後の課題

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

今後の課題としては次のことがあげられる：

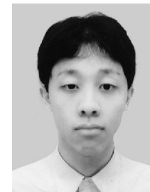
- 変換したトレースをパスによって区別する方法を工夫することにより，トレース数の増大を抑えトレースキャッシュ・ミス率を下げる．
- 逆 dualflow アーキテクチャではパスによって区別された同じアドレスのトレースのインデックスが衝突しがちである，すなわちコンフリクト・ミスが増加してしまうので，これを軽減させる．
- 応用として，逆 dualflow アーキテクチャではレジスタ・リネーミングが不要であるため，フロントエンドで行う処理に命令間での依存がない．これを利用して，トレースの長さとフロントエンドの幅を 2 倍にし，クロックを半分にすることにより，さらなる低消費電力を狙う．
- 消費電力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

謝辞　本論文の研究の一部は，文部科学省科学研究費補助金 No.20300015 による．

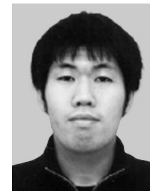
参　考　文　献

1) 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, Issue 1 (2001).
2) 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).
3) 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).
4) 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，京都大学 (2004).
5) Rotenberg, E., Bennett, S. and Smith, J.E.: Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching, *Proc. 29th MICRO*, pp.24–35 (1996).
6) 渡辺憲一，一林宏憲，五島正裕，坂井修一：プロセッサ・シミュレータ「鬼斬」の設計，先進的計算基盤システムシンポジウム SACSIS 2007（ポスター），pp.194–195 (2007).

(平成 20 年 1 月 29 日受付)
(平成 20 年 5 月　1 日採録)



一林　宏憲
1982 年生．2008 年東京大学大学院情報理工学系研究科修士課程修了．



塩谷　亮太（学生会員）
1981 年生．2008 年東京大学大学院情報理工学系研究科修士課程修了．同研究科博士課程に在学中．コンピュータアーキテクチャの研究に従事．

入江　英嗣（正会員）
1999 年東京大学工学部電子情報工学科卒業．2004 年同大学院情報理工学系研究科博士課程修了．博士（情報理工学）．同年より 2008 年まで科学技術振興機構 CREST “ディペンダブル情報基盤” 研究員．2008 年より東京大学大学院情報理工学系研究科助教．プロセッサアーキテクチャの研究に従事．電子情報通信学会，ACM 各会員．

© 2008 Information Processing Society of Japan

五島　正裕（正会員）

1968 年生．1992 年京都大学工学部情報工学科卒業．1994 年同大学院工学研究科情報工学専攻修士課程修了．同年より日本学術振興会特別研究員．1996 年京都大学大学院工学研究科情報工学専攻博士後期課程退学，同年より同大学工学部助手．1998 年同大学大学院情報学研究科助手．博士（情報学）．2005 年東京大学大学院情報理工学系研究科助教授，2007 年同大学院同研究科准教授，現在に至る．コンピュータ・システムの研究に従事．著書に『ディジタル回路』．2001 年情報処理学会山下記念研究賞，2002 年同学会論文賞受賞．IEEE 会員．

坂井　修一（正会員）

1981 年東京大学理学部情報科学科卒業．1986 年同大学院工学系研究科修了，工学博士．電子技術総合研究所，MIT，RWC，筑波大学を経て，1998 年東京大学助教授．2001 年より東京大学大学院情報理工学系研究科教授．現在同研究科副研究科長．計算機システムおよびその応用の研究に従事．特に並列処理アーキテクチャ，相互結合網，最適化コンパイラ，省電力アーキテクチャ，ディペンダブルシステム等の研究を進めている．情報処理学会研究賞（1989），情報処理学会論文賞（1991），IBM 科学賞（1991），市村学術賞（1995），IEEE Outstanding Paper Award（1995），Sun Distinguished Speaker Award（1997）等受賞．電子情報通信学会，人工知能学会，ACM，IEEE 各会員．日本学術会議連携会員，日本学術振興会専門委員．

© 2008 Information Processing Society of Japan