逆 Dualflow アーキテクチャ

一林 宏憲[†] 入江 英嗣[††,†]
五島 正裕[†] 坂井 修一[†]

LSI が微細化されるにつれ，ロジックの遅延に占める配線遅延の割合が大きくなってきている．このため，単に LSI を微細化するだけでは，プロセッサの動作周波数を高くすることはもはや不可能である．動作周波数を高めるためには，より深いパイプラインを採用する必要があるが，パイプラインを深くすると，各部のレイテンシの増加によって IPC (Instructions Per Cycle) は低下してしまう．本研究では，特にレジスタ・リネーミングに着目し，レジスタ・リネーミング・ステージ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，命令を動的にレジスタ・リネーミングの不要な形式に変換してトレース・キャッシュに保存することにより，レジスタ・リネーミング・ステージを省略する．

Anti-Dualflow Architecture

HIRONORI ICHIBAYASHI,[†] HIDETSUGU IRIE,[††,†] MASAHIRO GOSHIMA[†]
and SHUICHI SAKAI [†]

Since the wire delays are dominating the delays of logics as the process rule becomes smaller, it can not be accomplished to increase the frequency of processors only by shrinking the process rule. To increase the frequency, deeper pipelines are necessary. But deep pipelines increase the latency of processor logics, diminishing the IPC (Instructions Per Cycle). In this paper, we focus on register renaming. We propose a method to eliminate the register renaming stage — anti-dualflow architecture. In anti-dualflow architecture, instructions are dynamically converted to a register renamed form and stored in trace cache.

1. はじめに

LSI の製造プロセスの微細化が進むにつれて，ロジックの遅延に占める配線遅延の割合が大きくなってきている．このため，単に LSI を微細化するだけでは，プロセッサの動作周波数を高くすることはもはや不可能である．動作周波数を高めるためには，より深いパイプラインを採用せざるを得ない．しかし，パイプラインを深くすると，動作周波数は向上するものの，各部のレイテンシの増加によって IPC (Instructions Per Cycle) は低下してしまう．

レジスタ・リネーミング

本研究では，特に，out-of-order スーパスカラ・プロセッサのレジスタ・リネーミングに着目する．

レジスタ・リネーミング・ステージは，命令パイプラインのフロントエンドに位置する．レジスタ・リネーミング・ステージに割り当てられるサイクル数が増加した分だけ，分岐予測ミス・ペナルティが増加することになる．

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 Mapping Table)** を読み出す．このテーブルの遅延も配線遅延に支配されており，LSI の微細化，パイプラインの深化とともに，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]．

Dualflow アーキテクチャ

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

† 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
†† 科学技術振興機構
Japan Science and Technology Agency

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 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，分岐命令を超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:

- 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，命令パイプラインからレジスタ・リネーミング・ステージを省略することができる．

以下，2 章で，レジスタ・リネーミングの解釈とその遅延について述べる．3 章では，dualflow アーキテクチャについてまとめる．次いで，4 章において，提案手法について説明する．5 章で，予備的な評価の結果についてまとめる．

2. レジスタ・リネーミング

レジスタ・リネーミングでは，1 つの命令の生産するデータに対し，専用の物理レジスタを 1 つ割り当てる．命令の生産するデータに 1 対 1 で対応したこの物理レジスタは，最初にただ 1 回だけ書き込まれ，後は読み出されるだけである．これによって，ある物理レジスタを読み出す命令は，その物理レジスタに書き込んだ命令の生産するデータを使用するとしてよくなる．つまり，レジスタ・リネーミングとは，レジスタを介してデータを受け渡すことによって生じる偽の依存関係を解消し，真の依存関係，すなわちデータ・フローを明らかにする操作であると言える．

次に遅延について考える．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 Mapping Table)** を読み出すことで行う．RMT は，通常 RAM で実装される．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．フェッチ幅を FW とすると，FW 個の命令を同時にリネームするため，この RAM は読み出しポート数 $2 \times FW$，書き込みポート数 FW の RAM で実装される．この RAM の遅延が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・ロジックをパイプライン化する必要が生じるのである．

3. Dualflow アーキテクチャ

本章では，**dualflow アーキテクチャ**[2)~4)] について簡単に説明し，その問題点について述べる．Dualflow アーキテクチャ自体の詳細については文献[2)~4)] を参照されたい．

3.1 Dualflow アーキテクチャの命令

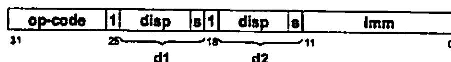

図 1　Dualflow アーキテクチャの命令フォーマット例

図 1 に，dualflow アーキテクチャの命令フォーマット例を示す．前述したように，通常の制御駆動型の命令セット・アーキテクチャと異なり，ソース・オペランドやデスティネーション・オペランドをレジスタ番号で指定しない．かわりに，コンシューマを指定するフィールド ***d1/d2*** がある．

d1/d2 フィールド中の *disp* フィールドは，この命令の生産したデータが *disp* 命令後の命令のオペランドとして使用されることを示す．そのデータが左オペランドになるか右オペランドになるかは s フィールドで指定する．

3.2 レジスタ・リネーミングとの関係

dualflow アーキテクチャにおいて命令に明示されたプロデューサとコンシューマの関係は，データ・フローそのものである．この意味において，dualflow アーキテクチャの命令列は「レジスタ・リネーミング済み」であると言える．よって dualflow アーキテクチャではレ

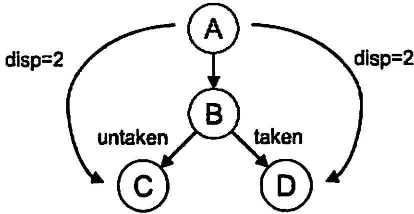

図 2 分岐を挟んだデータの授受

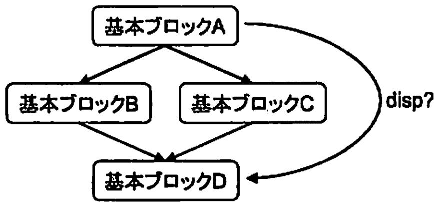

図 3 基本ブロックを挟んだデータの授受

ジスタ・リネーミングが不要である.

3.3 問 題 点

dualflow アーキテクチャでは，データの授受に関する次の 3 種類の制約がコードに課せられる:

数と距離 コンシューマを指定するフィールドは 2 つしかないので，2 つ以上の命令にデータを送りつけるには，データを受け取ってそのまま後の命令に送るだけの命令 (転送命令) を挿入する必要がある.

条件分岐 コンシューマの位置はコンパイル時に静的に決定しなければならないので，条件分岐の前の命令が分岐の後の命令にデータを送る場合であっても，分岐の結果によってデータを送る位置を変えることはできない．いずれかの条件分岐先で使用されるデータであれば，両方の分岐先に送ることになる．分岐先においては，同じ順番で，データを受けとるか nop で捨てる必要がある．図 2 に例を示す．命令 A から untaken 側の命令 C にデータを送る場合，同時に taken 側の命令 D も命令 A のデータを受け取ることになる．このため，命令 D でも命令 A のデータを使用するか捨てる必要がある

基本ブロック 1 つ以上の基本ブロックを越えてデータを送る場合，プロデューサからコンシューマへの変位が静的に定まらないため，直接データを送ることができない．データを送るためには，メモリを介する必要がある．図 3 に if-then-else 構造における例を示す．一般に，基本ブロック B と C の大きさは異なるので，基本ブロック A から基本ブロック D への変位を静的に決定することはできない.

op-code	D	destReg	1	RL	srcRegL / dispL	1	RR	srcRegR / dispR	Imm

ビット位置: 33, 27, 21, 14, 7, 0

図 4 dualflow 形式の内部表現

これらの制約によって，本来不要な命令が増加してしまう問題がある．これは，静的にプロデューサとコンシューマとの距離を決定していることが主な原因である．分岐の例に見るように，データ・フローは制御フローに依存するので，静的にデータ・フローを決定することには無理がある.

4. 逆 Dualflow アーキテクチャ

3.3 節で述べたように，dualflow アーキテクチャでは，命令セット・アーキテクチャとして dualflow 形式の命令を用いるため，プロデューサとコンシューマの関係を静的に決定しなければならず，データの受け渡しに制約が課せられる問題がある．本章では，dualflow アーキテクチャのこの問題を解決するアーキテクチャとして**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，制御駆動型の命令列を動的に dualflow 形式の内部表現に変換してキャッシュする．さらに，データの授受を指定する方向を逆にする，すなわち，後続の命令が先行する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ようにする．逆にすることにより，後述するように，レジスタ・リネーミングと同様な方法で命令を動的に dualflow 形式に変換することができる．一方，逆にしない場合，つまり先行する命令が実行結果の送り先を指定する方法だと，後続の，すなわち未来の命令を知らなければ送り先を判断することができず，動的に変換するのは現実的でない.

4.1 内 部 表 現

図 4 に，逆 dualflow アーキテクチャにおける命令の内部表現を示す．基本的にはソース・オペランドを Dualflow アーキテクチャと同様に命令間の距離で指定するが，距離が *disp* フィールドに収まらない場合がある．距離が *disp* フィールドに収まる命令を**距離指定範囲**と呼ぶことにする．距離指定範囲外の命令の実行結果を参照する場合にはレジスタ番号で指定することにする..

D, destReg *D* フィールドが 1 のとき，この命令は *destReg* 番のレジスタをデスティネーション・オペランド・レジスタとすることを示す.

		(untaken)		(taken)
imm	$1 = a	imm	$1 = a	imm	$1 = a
imm	$2 = b	imm	$2 = b	imm	$2 = b
blt	$1 < $2 then A	blt	[-2] < [-1] then A	blt	$1 < $2 then A
mov	$1 = $2	mov	$1 = [-2]	A: st	(...) = [-3]
A: st	(...) = $1	A: st	(...) = [-1]		

図 5　経路と命令変換

RL(R) ***dispL(R)/srcRegL(R)***　*dispL(R)*/*srcRegL(R)* フィールドは，*RL(R)* フィールドが 0 のとき *dispL(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 *srcRegL(R)* 番の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．

4.2 dualflow 形式への変換

制御駆動型の命令列を dualflow 形式に変換するのは，レジスタ・リネーミングと同様に行う．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レジスタに最後に書き込んだ命令の番号の対応を記憶するテーブル (以後 RMT と呼ぶ) を用意し，命令がフェッチされるたびに更新する．RMT からソース・オペランド・レジスタに最後に書き込んだ命令の番号を読み出し，今変換中の命令の通し番号との差を取れば，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．ただし，距離指定範囲より前の命令の実行結果を使用する場合にはレジスタ番号でオペランドを指定する．

ここで重要なのは，命令の変換結果は経路に依存するということである．経路が変われば，各論理レジスタに最後に書き込んだ命令の位置も変わる，つまり命令のソース・オペランドとなるデータを生産した命令の位置が変わることになる．図 5 の max(*a*, *b*) のコードを例として示す．左が変換前の命令列，中央が untaken，右が taken の経路にそって変換を行った結果である．[-*disp*] は，*disp* 命令前の命令の実行結果を表す．最後に$1 に書き込んだ命令が，untaken では 4 命令目の mov であり，taken では 1 命令目の imm であるので，最後の st 命令のソース・オペランドである$1 の変換結果が異なる．

命令の変換結果が経路に依存するので，経路の異なる命令は異なるものとして保存する必要がある．変換後の命令は，経路をタグに付加してキャッシュに格納する．ここで，命令を動的な順番に並べたものを格納するには，トレース・キャッシュが適している．

4.3 経路の表現

次に経路の表現について説明する．距離指定範囲より前の命令の実行結果はレジスタ番号で参照するので，距離指定範囲より前の命令がどうなっているかは命令の変換結果に影響しない．よって，経路としては距離指定範囲内の命令列だけを考えればよい．以下,「経路」というときには距離指定範囲内の命令列を指すことにする．また，経路の先頭の命令のことを**経路先頭**，距離指定範囲内の命令数を**経路長**と呼ぶことにする．

経路を表現するには，最も単純な方法として,「経路中にある命令のアドレスの列」が考えられる．しかし，この方法ではどんな経路であってもアドレスを経路長個保持することになり，効率が悪い．

分岐命令でない命令の次の命令は一意に決められるので，経路先頭の命令が分かれば，経路中の最初の分岐命令までの命令列が分かる．さらにその分岐命令のターゲットが分かれば，同様にして次の分岐命令までの命令列が分かる．これを繰り返すと経路中の全ての命令が分かる．よって「経路先頭のアドレスと経路中にある分岐命令のターゲットの列」で経路を表現できる．

さらに，条件分岐のターゲットは，条件分岐が taken であるか untaken であるかによって決めることができるので，条件分岐についてはターゲットを保持する代わりにその taken/untaken を保持すればよい．よって「経路先頭と経路中にある条件分岐の taken/untaken，間接分岐のターゲットの列」で経路を表現できる．条件分岐の taken/untaken は高々経路長 bit であるし，間接分岐が出現する頻度は高くないので，このような表現をすることによって経路を保持するために必要な記憶量を小さくできる．

4.4 まとめと効果

逆 dualflow アーキテクチャでは，通常の制御駆動型の命令セット・アーキテクチャを動的に dualflow 形式の命令に変換し，その命令までの経路とともにキャッシュに格納する．キャッシュから dualflow 形式に変換済みの命令を取り出すことができれば，その命令はレジスタ・リネーミングせずに実行することが可能であり，分岐予測ミス・ペナルティを削減できる．

5. 評　　価

逆 dualflow アーキテクチャでは，同じ命令列であっても経路が異なれば異なる dualflow 形式の命令列に変換され，経路もタグとしてキャッシュに格納される．つまり，経路ごとに命令が複製されてキャッシュに格納されるため，キャッシュへの負荷が増大する．

本章では，予備的な評価として，基本ブロック (Basic Block, 以下 BB と呼ぶ) 単位で読み書きを行うキャッシュ(以下，**BB-Cache** と呼ぶ) を仮定し，BB-Cache の

表 1 測定に使用したプログラム

プログラム	入力セット	プログラム	入力セット
164.gzip 176.gcc 181.mcf 186.crafty 197.parser	train	254.gap 255.vortex 256.bzip2 300.twolf	train

性能を評価することによって，逆 dualflow アーキテクチャにおけるトレース・キャッシュミス率を評価する．ただし，ここで用いる BB は，本来の意味での BB ではなく，分岐命令のターゲット命令から次に現れる分岐命令までの命令列とする．

5.1 評価環境

Alpha プロセッサ向けにコンパイルしたプログラムを実行したトレース・データから，BB-Cache のシミュレーションを行うプログラムを作成し，BB-Cache のミス率を測定した．ベンチマークには，SPEC2000 より表 1 に示す 9 本のプログラムを使用した．

BB-Cache は BB をエントリとし，BB の先頭アドレスと経路をタグとして用いる，真の LRU を用いた 4-way セット・アソシアティブ・キャッシュで実装した．次の 3 つのモデルに対し，BB-Cache のエントリ数を 1K，2K，4K，8K と変化させて BB-Cache のミス率を測定した．これらのモデルは，それぞれ通常の制御駆動型，経路長 16 の逆 dualflow アーキテクチャ，経路長 32 の逆 dualflow アーキテクチャに対応する．

NoPath 経路なし

Path16 経路として 16 命令を含む BB の列を用いる

Path32 経路として 32 命令を含む BB の列を用いる

ミス率の算出方法は，BB-Cache からの BB 読み出しがミスした場合，その BB 中の命令全てがミスしたと数える方法と，通常の命令キャッシュと同様，ミスを引き起こした命令，つまり BB の先頭の命令のみをミスとし，残りの命令はヒットと数える方法が考えられる．今回の評価では後者を採用した．

5.2 評価結果

表 2 に，各プログラムの平均 BB 長と経路に含まれていた BB 数の平均を示す．図 6 に，BB-Cache エントリ数が 1K のときの各モデルでの BB-Cache ミス率を示す．BB の長い列で表される長い経路を付加することによって，前述したようにキャッシュへの負荷が高くなっていることが分かる．

ここで，特にミス率の悪化している gcc，crafty，gap，vortex，twolf について，BB-Cache エントリ数を増加させたときのミス率の改善を見る．図 7 に Path32 での，図 8 に Path16 での BB-Cache ミス率を，NoPath，BB-Cache エントリ数 1K のときのミス率で正規化したものを示す．Path32 においては，gcc，crafty，gap，vortex について BB-Cache エントリ数を 4 倍にすることで，経路を付加したことによるミス率の悪化が解消されることが分かる．twolf については，BB-Cache エントリ数を 8 倍にすることでミス率の悪化が解消される．Path16 においては，gcc と voretx について

表 2 BB 長と経路に含まれる BB 数

プログラム	平均 BB 長	平均 BB 数 (Path16)	平均 BB 数 (Path32)
164.gzip	9.33	2.63	4.40
176.gcc	6.88	3.11	6.04
181.mcf	4.91	3.98	7.52
186.crafty	8.54	3.24	5.39
197.parser	6.50	3.50	6.13
254.gap	7.27	3.39	5.96
255.vortex	6.81	3.32	5.85
256.bzip2	10.41	2.28	4.16
300.twolf	8.34	3.01	5.29

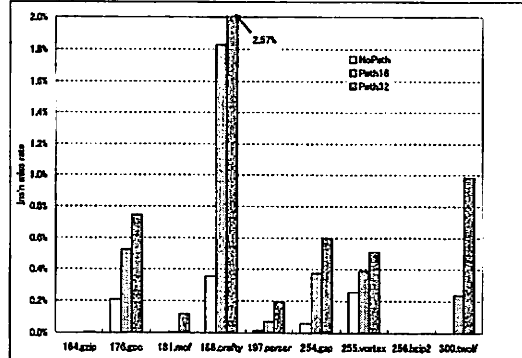

図 6 BB-Cache エントリ数が 1K のときの各モデルでの命令ミス率

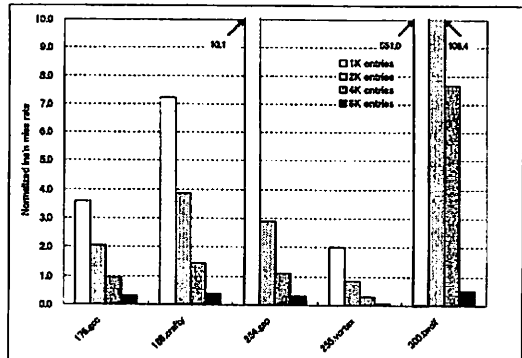

図 7 Path32 における BB-Cache エントリ数に対する正規化命令ミス率

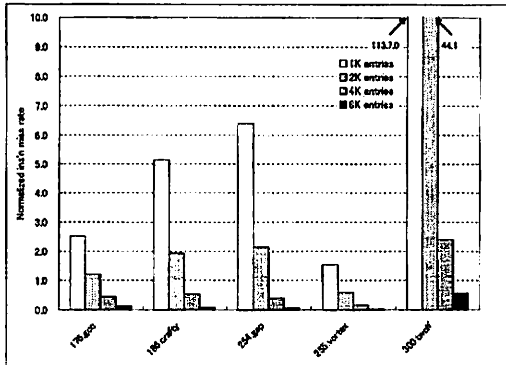

図 8　Path16 における BB-Cache エントリ数に対する正規化命令ミス率

BB-Cache エントリ数を 2 倍，crafty と gap について BB-Cache エントリ数を 4 倍にすることでミス率の悪化が解消される．twolf についてはミス率の悪化を解消するには 8 倍のエントリ数が必要であった．

6. おわりに

本研究では，レジスタ・リネーミング・ステージを省略することによりパイプライン段数を削減する手法として，逆 dualflow アーキテクチャを提案し，その予備的な評価を行った．逆 dualflow アーキテクチャでは，命令を動的に dualflow 形式に変換することにより，レジスタ・リネーミングを省略することができる．変換した命令は経路とともにキャッシュに格納されるので，経路の種類の分だけキャッシュの負荷が増大し，ミス率が悪化するが，経路長が 16 のときはキャッシュのエントリ数をを 2 倍から 4 倍弱にすることで，経路長が 32 のときは 4 倍程度にすることでミス率の悪化を解消できる．

今後の課題として，次のことがある:

- 距離指定範囲外の命令の実行結果はレジスタ番号を用いて参照することにしたが，具体的にどのようにデータを受け渡せばよいかが分かっていない．dualflow 形式で依存関係が明示されていないので，何らかの方法で依存関係を解決する必要がある．
- 現在はどんな経路も正確に表現できるとしている．間接分岐のターゲットを保持する個数には制限が必要であり，間接分岐の多い ― 最悪で経路の全てが間接分岐である経路については経路が正確に表現できない．正確に表現できなかった場合への対処を考える必要がある．

謝　辞

本研究の一部は，文部科学省科学研究費補助金 基盤研究 B(2) #16300013，CREST プロジェクト「ディペンダブル情報基盤」，21 世紀 COE プログラム「情報科学技術戦略コア」の支援により行った．

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).