情報処理学会論文誌　コンピューティングシステム　Vol.6　No.1　17–30 (Jan. 2013)

タイミング・フォールト耐性を持つOut-of-Orderプロセッサ

五島 正裕[1,a)]　倉田 成己[1]　塩谷 亮太[2]　坂井 修一[1]

受付日 2012年7月4日, 採録日 2012年9月7日

概要：半導体プロセスの微細化にともなって増大するランダムばらつきに対する対策の 1 つに，タイミング・フォールトを動的に検出/回復する技術がある．しかし既存の回復技術は，単純なスカラ・プロセッサにしか対応することができなかった．本稿では，複雑な out-of-order スーパスカラ・プロセッサにも対応できる回復技術を提案する．Out-of-order スーパスカラ・プロセッサでは特に，リオーダ・バッファ（ROB）やロード/ストア・キュー（LSQ）の内部で発生するフォールトへの対処が問題となる．そのためまず，out-of-order スーパスカラ・プロセッサのコミット周りを詳細に検討した．コミットの PNR（Point-of-No-Return）という概念を導入し，コミットの開始ではなく，PNR を通過することが命令のコミットであるという設計規約を設けた．そのうえで，通常は LSQ の内部にあるストア・バッファを PNR の下流へと別体化する．そして，フォールト発生時には，PNR の上流を初期化することによってフォールトの影響を取り除く．これらのことによって，ROB/LSQ を含む，PNR より上流のいかなる部分に発生するフォールトにも対応できるようになる．ストア・バッファを LSQ から別体化することによってコミットの遅延が伸びるが，それによる IPC の低下は平均で 0.7%と，十分に低いことを確認した．

キーワード：プロセスばらつき，タイミング・エラー，フォールト・トレランス，Out-of-Order スーパスカラ・プロセッサ，命令コミットメント

Timing-Fault-Tolerant Out-of-Order Processor

Masahiro Goshima[1,a)]　Naruki Kurata[1]　Ryota Shioya[2]　Shuichi Sakai[1]

Received: July 4, 2012, Accepted: September 7, 2012

Abstract: A decrease of the feature size of LSIs leads to an increase of the effect of random variation. Techniques that detect and recover from timing faults can solve this problem. Existing techniques, however, can only be applied to simplest scalar processors. This paper proposes a new technique that can be applied to complex out-of-order superscalar processors. The point is how to handle the faults that occurs inside of the Reorder Buffer (ROB) and the Load/Store Queue (LSQ). We examine these commitment modules in detail, introduce the notion of Point-of-No-Return (PNR), and make a more general design rule that says not start of the commitment but passing through PNR is the actual commitment of an instruction. Then, we separate the store buffer from inside of the LSQ to below the PNR. And, the effect of the faults is removed by initializing the pipeline above the PNR. This scheme enables handling of faults that occurs any part above the PNR including the ROB and the LSQ. The latency of the commitment is prolonged by the separate store buffer. However, the simulation results show that IPC degradation caused by it is no more than 0.7% on average.

Keywords: process variation, timing error, fault tolerance, out-of-order superscalar processor, instruction commitment

1. はじめに

半導体プロセスの微細化にともない，素子のランダムなばらつきの問題が顕在化しつつある [1], [2]．微細化によって，素子性能の平均（typical）値は向上するものの，ばらつ

[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, Bunkyo, Tokyo 113–8656, Japan
[2] 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University, Nagoya, Aichi 464–8603, Japan
[a)] goshima@mtl.t.u-tokyo.ac.jp

きの増大によって，ワースト値の向上は減殺されてしまう．

従来の LSI 設計は，このワースト値に基づくワースト・ケース設計である．したがって，このまま微細化を進めても従前のような性能向上は期待できなくなってしまう．

ばらつきへの対策としては，様々なレベルの技術が考えられる [3] が，回路〜アーキテクチャ・レベルの技術として，タイミング・フォールトの検出と回復がある．

タイミング・フォールト検出/回復

タイミング・フォールトとは，回路遅延の動的な変動によって生じる過渡故障である．本稿ではタイミング・フォールト以外のフォールトを扱わないので，単にフォールトとあった場合にはタイミング・フォールトのことを指すと解されたい．

ワースト・ケース設計では，ワースト・ケースにおいてもタイミング・フォールトが発生しないように設計を行う．そのため，フォールトが生じるのは，サーモ・センサの故障による熱暴走などの想定外の状況においてのみである．一方で，ワースト・ケースにおいてもフォールトが発生しないような見積りは，ばらつきが増大したプロセスでは悲観的になりすぎる．

タイミング・フォールトを検出/回復する技術は，特に DVFS – Dynamic **V**oltage and **F**requency Scaling と組み合わせて用いることで，このような問題に対処することができる．すなわち，ワースト・ケースよりも低い電圧（V），高い周波数（F）で動作させ，その結果生じるフォールトを検出し，回復すればよい．回復にはペナルティが付随するから，その影響が十分に小さくなるように，フォールトの発生確率が十分に低い V/F の組みを見つける．このようにすれば，個体差や動作環境に応じた実際の遅延に基づく動作が可能となり，ワースト・ケース設計の悲観的すぎる見積りから脱却することができる．

このような技術のうち，プロセッサを対象とするものは，一般に，タイミング・フォールトを検出する回路レベルの技術と，検出後に回復を行うアーキテクチャ・レベルの技術の 2 つからなる．

Razor Ⅱ

プロセッサを対象とするタイミング・フォールト検出/回復技術としては，**Razor Ⅱ** [4] が有名である．Razor Ⅱでは，回路レベルの検出技術としては，パイプラインの各ステージで生じたフォールトを検出する Razor Ⅱ FF を；アーキテクチャ・レベルの回復技術としては，パイプライン・フラッシュによる回復を，それぞれ提案している．すなわち，Razor Ⅱ FF によってフォールトが検出されると，その命令が例外を起こしたのと同様に考え，パイプライン・フラッシュによって当該命令（と後続の命令）をパイプラインから取り除くのである．

Razor Ⅱを構成する 2 レベルの技術のうち，回路レベルの検出技術には大きな問題はないが，アーキテクチャ・レベルの回復技術には不十分な点がある．端的にいえば，Razor Ⅱは，データ・パス上で生じるフォールトについては考慮されているが，それ以外の部分——制御系で生じるフォールトについてはほとんど考慮されていない．その結果，文献 [4] で紹介されているようなごく単純なスカラ・プロセッサにしか適用できないのである．

提案技術の概要

そこで本稿では，2 レベルの技術のうち，アーキテクチャ・レベルの回復技術について新たな提案を行う．すなわち，フォールトを検出する回路レベルの技術については，Razor Ⅱなどと同様と考えてよい．

提案する回復技術は，Razor Ⅱとは異なり，複雑な out-of-order スーパスカラ・プロセッサにまで適用可能であることを特長とする．最近では，Cortex-A のように，従来組み込み系と考えられてきた分野においても out-of-order スーパスカラ・プロセッサが用いられるようになってきている．したがって，より複雑なプロセッサにまで適用可能であることは，今後ますます重要となるであろう．

本稿の構成は以下のとおりである：続く 2 章では，まず Razor Ⅱを含むタイミング・フォールト検出/回復技術についてまとめる．その後 3 章で，out-of-order スーパスカラ・プロセッサの実際について説明し，Razor Ⅱの考え方では不十分であることを示す．続く 4 章と 5 章で，提案の回復技術について述べる．6 章では，IPC の評価の結果を示す．

2. タイミング・フォールト検出/回復技術

本章では，プロセッサを対象としたフォールト検出/回復技術の一般的なことがらから始めて，Razor Ⅱ [4] の回復技術に特有のことがらについてまとめる．

前述したように，フォールト検出/回復技術は，一部の例外 [5] を除いて，フォールトを検出する回路レベルの技術と，検出後に回復を行うアーキテクチャ・レベルの技術の 2 つからなる [6], [7]．以下では，まず 2.1 節で回路レベルの検出技術について述べた後，2.2 節と 2.3 節でアーキテクチャ・レベルの回復技術について述べる．ここまでは一般的な検出/回復技術についての説明であって，我々の以前の提案 [8] や，Razor Ⅱ，そして，本稿の提案にもあてはまる．最後に 2.4 節において，Razor Ⅱの回復技術に特有の点についてまとめる．

2.1 フォールト検出技術

フォールトを検出する回路レベルの技術としては，Razor では **Razor FF** [9], [10], [11] が，Razor Ⅱ [4] では Razor Ⅱ FF が，それぞれ提案されている．

本節では，理解が容易であるので，Razor Ⅱ FF ではなく，Razor FF について概説する．これら 2 つは，フォールト検出の原理が異なり，Razor Ⅱ FF の方が面積などの

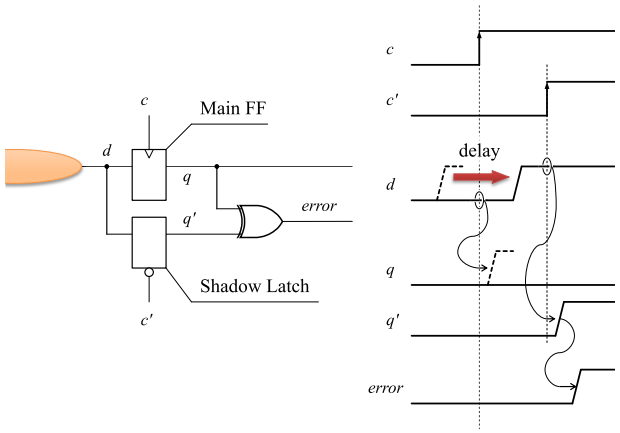

図 1　Razor FF の回路図（左）と動作例（右）
Fig. 1　Circuit diagram (left) and timing chart (right) of Razor FF.

点で有利であるとしている．しかし，回復技術にとって重要なフォールト検出のタイミングは同様であり，次節以降の議論には影響を及ぼさない．Razor FF と Razor Ⅱ の回復技術の組合せも可能である．

図 1 に，Razor FF の回路図と動作例を示す．Razor FF は，メインの FF と，それより少し遅れた位相でサンプリングを行う**シャドウ・ラッチ**によって構成されている．同図の動作例では，d の変化が遅れ（図中の赤い矢印），その結果メイン FF には誤った値（L）がサンプリングされている．この時点ですでにフォールトは発生していることに注意されたい．これより少し遅れて，シャドウ・ラッチは正しい値（H）をサンプリングすることになる．メインの出力（L）とシャドウの出力（H）が異なるので，*error* がアサートされ，フォールトが検出される．

このようにフォールトの検出は事後的であり，次節で述べるように，このことが回復技術に大きな影響を及ぼす．

2.2　アーキテクチャ・レベルの回復技術

フォールトからの回復の技術には，回路レベルのもの [5] とアーキテクチャ・レベルのものが考えられる．ただしプロセッサの場合に限れば，アーキテクチャ・レベルの技術が比較的容易に実現可能である．それは，プロセッサにはアーキテクチャ・ステートが定義されているからである．

アーキテクチャ・ステート

アーキテクチャ・ステートは通常，命令セット・アーキテクチャにおいて定義され，PC（を含む PSW）と（論理）レジスタ・ファイル，および，その他の制御レジスタからなる．アーキテクチャ・ステートは，主に OS とのインタフェースとして，コンテクスト・スイッチ時にセーブ/レストアの対象となる．

一方，マイクロアーキテクチャにおいては，PC や（論理）レジスタ・ファイルに加えて，主記憶もアーキテクチャ・ステートに含めると都合がよい．そのようにすると，命令の**コミット**を，命令（の実行結果）によるアーキテクチャ・ステートの不可逆的な更新と定義できるからである．より具体的には，ストア命令のコミットを，ストア・データによる 1 次データ・キャッシュの不可逆的な更新とすることができる．

以下，本稿では，この拡張された定義を採用する．

アーキテクチャ・レベル回復技術の概要

プロセッサにおけるフォールト検出/回復は，アーキテクチャ・ステートを利用して，次のようにすればよい：

(1) **コミットの停止**　コミットを停止することによって，アーキテクチャ・ステートをフォールトから保護する．Razor FF などでフォールトを検出すると，エラー情報をコミット・モジュールへと伝え，フォールトの影響を受けた命令がコミットされる前にコミットを停止する．

(2) **フォールトの影響の除去**　何らかの方法によってパイプラインからフォールトの影響を取り除く．その後，保護されたアーキテクチャ・ステートから実行を再開する．

後で詳しく述べるが，フォールトの影響を除去する方法としては，Razor Ⅱ ではパイプライン・フラッシュ，我々の以前の提案 [8] や 4 章以降で述べる本稿の提案ではパイプラインの初期化を用いる．

アーキテクチャ・レベル回復技術のパイプライン

図 2 (a) に，アーキテクチャ・レベルの回復技術のパイプラインの概略を示す．

パイプライン・ラッチは，原則的には Razor FF などで構成されるとする[*1]．

エラー通知ネットワークは，各ステージにおいて発生したエラー信号を，パイプラインに沿って下流へと通知する．各ステージでは，Razor FF から出力されるエラー信号の OR をとり，次のステージへと出力する．出力されたこのエラー信号は，次のステージにおける OR の入力に含められる．

こうすることによって，各ステージの OR の fan-in を当該ステージの Razor FF の数 +1 に抑えることができる．

エラー通知ネットワークによってエラー信号がパイプラインの最下流にあるコミット・モジュールへと伝えられ，コミット・モジュールはコミットを停止する．その後，フォールトの影響の除去を行い，保護されたアーキテクチャ・ステートから実行を再開する．

2.3　アーキテクチャ・レベルの回復技術の包括性

耐故障性や信頼性では，手法の包括性が肝要である．包括性の観点からは，これらの技術は，以下のようにいえる：

[*1] フォールト発生の可能性が低いものに関しては，通常の FF に置き換え可能である．

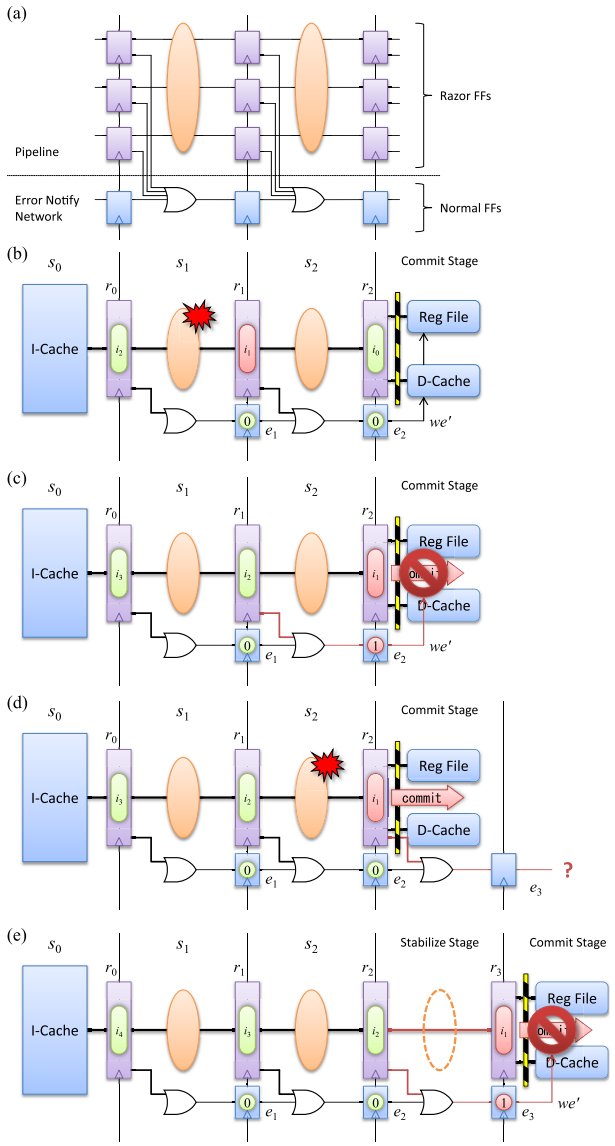

図 2　アーキテクチャ・レベルの回復技術のパイプライン
Fig. 2　Pipeline of architecture-level recovery scheme.

- コミット・モジュールより上流で発生するフォールトによってアーキテクチャ・ステートが破壊されることがない．さらに，何らかの包括的な方法によって，コミット・モジュールより上流のフォールトの影響は除去される．
- したがって，問題があるとすればコミット・モジュールだけであり，この部分がフォールト・フリーであることを確かめればよい．

このように，これらの回復技術はプロセッサ上のフォールトについて包括的に扱っているということができる．

2.4　Razor II の回復技術

前節までの説明は一般的なもので，我々の以前の提案 [8] や，Razor II の回復技術，そして，4 章以降で述べる本稿の提案でも本質的な違いはない．本節からは，Razor II の回復技術に特有な点についてまとめる．

図 2 (b)〜(e) にも端的に表れているように，Razor II の回復技術はスカラ・プロセッサ（パイプライン・マシン）を意識したものといえる．その特徴は以下の 2 点にまとめられる：

(1) 各命令と，その命令が起こしたフォールトのエラー信号がパイプラインを並んで下っていく．
(2) 例外や投機ミスに対するパイプライン・フラッシュによってフォールトの影響を取り除く．

以下，図 2 (b)，(c) を例に Razor II のパイプラインの動作を説明する：

図 2 (b) は，命令 i_1 がステージ s_1 を通過中にフォールトが発生した（図中，爆発のアイコン）場合を示す．同図は，そのサイクルの終わりを表している．サイクルの終わりにパイプライン・ラッチ r_1 に格納された命令 i_1 はフォールトの影響を受けており，コミットしてはならない．この時点では，r_1 と同相のレジスタ e_1 はセットされていないことに注意されたい．

図 2 (c) は，その次のサイクルの終わりを表している．このサイクルの終わりには，i_1 はパイプライン・ラッチ r_2 に格納される．エラー信号はこのサイクルにおいて OR されて，i_1 が格納されたレジスタ r_2 と同相のレジスタ e_2 がセットされる．

さらにその次のサイクルには，e_2 によってライト・イネーブル we' が制御され，i_1 のレジスタ・ファイル/データ・キャッシュへの書き込みが抑制される，すなわち，i_1 のコミットを停止することができる．

このように，エラー信号はフォールトを起こした命令と並んでパイプラインを下っていくことになる．したがって，パイプラインを下っていく各命令に 1-bit のエラー・フラグを付加したものと考えてよい．

スタビライズ・ステージ

コミット直前のステージでフォールトが発生した場合にもコミットを停止するためには，余分なステージが必要となる．以下，図 2 (d)，(e) を例に動作を説明する：

図 2 (d) は，コミット直前のステージ s_2 でフォールトが発生した場合を表している．この場合，次のサイクルにエラー信号は OR されて e_3 がセットされるが，そのときには i_1 によるレジスタ・ファイル/データ・キャッシュへの書き込みが終わってしまっている．すなわち，フォールトの影響を受けた命令がコミットされることを防げない．

図 2 (e) は，このような状況を避けるために余分なステージ s_3 を追加したものである．Razor II では，このステージ s_3 を**スタビライズ・ステージ**と呼んでいる．i_1 がスタビライズ・ステージをただ通過している間に，エラー信号は OR されて e_3 がセットされる．

次のサイクルには，e_3 によってライト・イネーブル we' が制御され，i_1 のレジスタ・ファイル/データ・

キャッシュへの書き込みが抑制される，すなわち，i_1 のコミットを停止することができる．

スタビライズ・ステージはロジックを含まないので，このステージにおけるフォールトの発生確率は十分低いと見なすことができる．

パイプライン・フラッシュ

Razor Ⅱでは，ゼロ除算などの例外からの回復と同様に，パイプライン・フラッシュによってフォールトから回復する．フォールトが検出されると，フォールトを起こした命令のエラー・フラグがセットされる．何サイクルか後にエラー・フラグがセットされた命令がコミット・ステージにたどり着くと，当該命令のコミットを停止するとともに，パイプライン・フラッシュによって当該命令と後続の命令をパイプラインから取り除くのである．

次章で詳しく述べるが，このパイプライン・フラッシュが，Razor Ⅱの適用範囲をスカラ・プロセッサに限定する主な要因となる．

3. Out-of-Order プロセッサと Razor Ⅱの限界

前章で述べたように，プロセッサを対象とするフォールト検出/回復技術は一般に，回路レベルの検出技術とアーキテクチャ・レベルの回復技術とからなる．Razor Ⅱは，この 2 つのうち，検出技術には大きな問題はないが，回復技術に不十分な点がある．Razor Ⅱの回復技術は，ごく単純なスカラ・プロセッサでは正しく動作するが，実用的なプロセッサには適用できない．

本章では，特に out-of-order スーパスカラ・プロセッサのコミットについて説明し，Razor Ⅱの技術では限界があることを示す．以下，3.1 節からコミットについて詳しく述べ，3.5 節から Razor Ⅱの問題点について述べる．

3.1 コミットに関わるモジュール

Out-of-order スーパスカラ・プロセッサの基本的な構成は，コミットの観点からは，以下の 2 つの方式に大別できる：

(1) 命令の実行結果を，一律に物理レジスタ・ファイルに書き込み，物理レジスタ番号の制御によってコミットを実現する方式

(2) 命令の実行結果を，いったんリオーダ・バッファに書き込み，論理レジスタ・ファイルへの移動によってコミットとする方式

コミットについて理解するには後者の方が容易であるため，本稿ではそちらを仮定する．ただし，前者に対する変更も可能である．

図 3（上）に，後者のモデルの概観を示す．コミットに関わるモジュールは 2 系統あり，同図中では上下に描かれている：

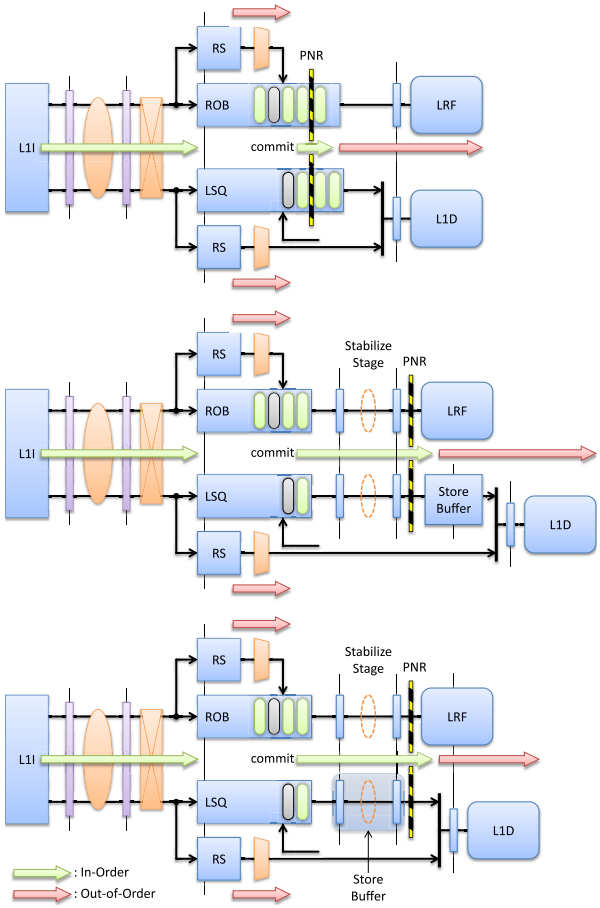

図 3　前提モデル（上）と提案モデル（中，下）

Fig. 3　Base model (top) and proposed models (middle, bottom).

上　**ROB**：リオーダ・バッファ　→
　　LRF：論理レジスタ・ファイル
下　**LSQ**：ロード/ストア・キュー　→
　　L1D：1 次データ・キャッシュ

同図中の **PNR** は，Point-of-No-Return の略で，ここを超えた命令は 2 度と超える前の状態に戻ることはできない点である．PNR については，後で詳しく述べる．

Out-of-order スーパスカラ・プロセッサでは，命令の実行は out-of-order に行われるが，その結果は in-order に実行した場合と同一でなければならない．ROB/LSQ の役割とは，端的にいえば，in-order な結果を保証しつつ out-of-order 実行を実現することである．具体的には，以下の 2 つにまとめられる：

コミット　コミット，すなわち，アーキテクチャ・ステートの不可逆的な更新は，in-order に行われなければならない．そのため ROB/LSQ は，基本的には FIFO で構成され，各エントリは in-order に割り当てられる．LRF/L1D の更新は，実行が終了した命令から順にではなく，in-order に行われる．

フォワーディング　命令がコミットされ，実行結果が実際に LRF/L1D から読めるようになるまでの間，後続

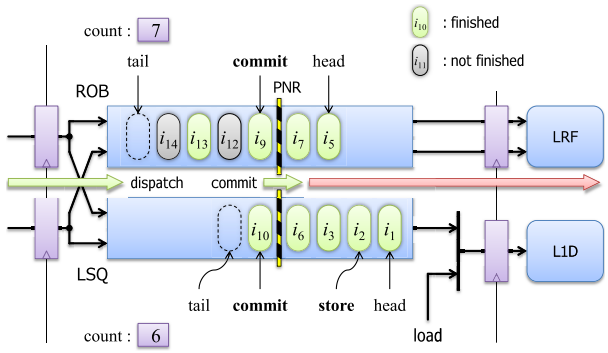

図 4　リオーダ・バッファ（ROB）とロード/ストア・キュー（LSQ）
Fig. 4　Re-Order Buffer (ROB) and Load/Store Queue (LSQ).

の命令には ROB/LSQ が依存元の命令の結果を供給する．ただし，ROB に関しては，フューチャ・ファイルなど，別のモジュールがこの役割を担うことが多い．一方，LSQ に関しては，実際に LSQ がこの役割を担うことが多い．LSQ のフォワーディング機能については，3.3 節で詳しく述べる．

3.2　リオーダ・バッファ

図 4（上）に，ROB の典型的な構成例を示す [12]．この例では，ROB は 3 つのポインタを持つリング・バッファとして構成されている．**head/tail** ポインタは，通常のリング・バッファと同様，有効なエントリの先頭と末尾のエントリを指す．そのほかに，有効なエントリ数を数えるカウンタ **count** がある．

commit ポインタ

ROB に特徴的であるのが，第 3 のポインタ，**commit** である．commit ポインタは，次にコミットの対象となるエントリを指す．毎サイクル，commit ポインタの指すエントリから下流に向かって命令を探し，終了していない命令が見つかったら，そのエントリへと commit ポインタを進める．コミット幅以内に終了していない命令がない場合には，コミット幅だけ進められる．

したがって，head〜commit のエントリは，コミットされ，LRF へ書き込むことが確定しているが，ROB からのデキューが終わっていない命令のエントリとなる．commit ポインタの更新により head〜commit の範囲に入った命令は，通常，次のサイクルに読み出され，LRF に送られる．

同図中，PNR（Point-of-No-Return）は，commit ポインタの指す命令の直前に位置している．すなわち，commit ポインタが更新されて PNR を超えた命令は，2 度と超える前の状態には戻すことができず，必ず LRF を更新しなければならない．PNR については，次章でさらに詳しく述べる．

3.3　ロード/ストア・キュー

図 4（下）に，LSQ の典型的な構成例を示す [12]．LSQ は，ROB と同じく，リング・バッファとして構成されている．

また ROB と同様，commit ポインタを持ち，ROB の commit ポインタと同期して更新される．

ロード/ストア・キュー内のストア・バッファ

ただし，ROB の場合と異なり LSQ の場合には，head〜commit にあるストア命令はただちに L1D への書き込みを開始できるわけではない．L1D のポートは，通常，ロード命令が優先して使用するため，ストア命令による L1D への書き込みは**サイクル・スチール**によって行われる，すなわち，コミットされたストア命令の読み出しは，L1D のポートの空きを待たなければならない．

図 4 の構成では，LSQ の head〜commit の部分が，いわゆる**ストア・バッファ**として，以下の 2 つの役割を果たすことになる：

バッファリング　L1D のポートの不足により LSQ からの読み出しが停止している間も，commit ポインタの更新を続けることができる．

フォワーディング　前述した ROB/LSQ のフォワーディング機能と原則変わりない．以下でより詳しく述べる．

ストア・バッファのフォワーディング機能

最近 commit ポインタの head 側に移ったストア命令は，まだ L1D に反映されてはいない．したがって，後続のロード命令に対しては，アドレスの一致/不一致を調べ，（もしあれば）依存元のストア命令の値を供給する必要がある．

このため，ストア命令が LSQ からデキューされるのは，L1D へと値を読み出した後**ではなく**，後続のロード命令が L1D に書き込まれた値を読めるようになった後でなければならない．

store ポインタ

LSQ には，ストア・バッファ機能のため，第 4 のポインタ **store** がある．store ポインタは，次に L1D へと送り出すストア命令を指す，すなわち，head〜commit のエントリのうち，どこまで L1D へ送り出し，次にどこから送り出すかを，store ポインタで管理する．

したがって，この store ポインタの要・不要は，書き込み先のポートがつねに空いているかどうかによる．前節で述べた ROB においては通常，ROB からの書き込みのための専用のポートが LRF にあり，専用なので必ず空いている．前述したように，commit ポインタの更新により head〜commit の範囲に入った命令は，必ず次のサイクルに読み出され（LRF に送られ）るので，store ポインタは別途必要ないのである．

3.4　コミットの考慮点

前章の図 2 のような単純なスカラ・プロセッサの場合，LRF/L1D の更新がコミットであった．一方，図 4 のよう

な構成においては，LRF/L1D の更新ではなく，ROB/LSQ の commit ポインタの更新こそがコミットとなる．それは，先にフラッシュについて考えると分かりやすい．

パイプライン・フラッシュ

例外や投機ミスによるパイプライン・フラッシュは，commit～tail のエントリを削除することによって実現される．具体的には，tail = commit と tail ポインタを commit ポインタの内容で上書きすると同時に，count を tail − commit の分だけ減らす[*2]．

コミットに関する設計規約

上述のように，フラッシュでは，commit～tail のエントリを削除して，それらの命令を実行する前の状態に巻き戻すことができる．逆に，head～commit のエントリは，巻き戻す必要がないものとして設計するのが定石である．もしまだ巻き戻す必要があるのなら，その命令を head～commit に入れなければよいからである．すなわち，「ROB/LSQ の commit ポインタの更新がコミットとなる」とは，単なる設計上の規約であって，通常そのような規約の下で各部の設計が行われるという意味である．

この規約に従えば，commit ポインタより head 側に入ったエントリは，もはや巻き戻されことがなく，必ず LRF/L1D に反映させなければならない．すなわち，commit ポインタの指す命令の直前を PNR（Point-of-No-Return）とすることになる．

3.5　Razor Ⅱ の回復技術の問題点

前節までで述べた ROB/LSQ に対して，2.4 節でまとめた Razor Ⅱ の回復技術を適用しようとすると，以下のような様々な疑問や問題が生じる：

(1) フォールトが検出されると，フォールトを起こした命令のエラー・フラグがセットされる．
- とすると，ROB/LSQ のエントリにエラー信号のビットを付加するのか？　そして，たとえば，フロントエンドでフォールトが生じた場合，当該命令のこのビットをセットするのか？
- しかし，たとえば ROB/LSQ のポインタのいずれかでフォールトが発生した場合など，どの命令のフォールトと見なせばよいのか？　ROB/LSQ 内のすべての命令のエラー・フラグをセットするのか？

(2) フォールトを起こした命令が，コミット・ステージの直前に設けられたスタビライズ・ステージで待つ間に，コミットが停止される．
- コミット・ステージとはどこなのか？　あるいは，commit ポインタがフォールトを起こした命令を指すことをもって，コミット・ステージにたどり着いたと解せばよいだろうか？
- だとしても，スタビライズ・ステージを設ける場所がない．特に，ROB/LSQ の commit ポインタでフォールトが生じた場合，その瞬間にアーキテクチャ・ステートは正しくなくなる．その後，コミットを停止，すなわち，commit ポインタの更新を停止しても手遅れである．

(3) パイプライン・フラッシュによって回復する．
- 制御系に発生したフォールトからはパイプライン・フラッシュによっては回復できない．この点が特に致命的であるので，次節以降で詳しく述べる．

3.6　パイプライン・フラッシュによる回復の困難

制御系に発生したフォールトからは，パイプライン・フラッシュによっては回復できない．まず，最も分かりやすい例として，前述の ROB の 3 つのポインタやカウンタのいずれかにフォールトが起こった場合を考えよう．

ROB/LSQ におけるタイミング・フォールト

前節で述べたように，パイプライン・フラッシュでは，tail = commit と tail ポインタを commit ポインタの内容で上書きすると同時に，count を tail − commit の分だけ減らす．このような操作を行っても，以下のように，正しい状態には戻らない：

- head，tail や count にフォールトが生じた場合，お互いに齟齬が生じる．以降は誤った値に基づいて更新されるだけで，齟齬のない状態には戻らない．このような状態では，有効なエントリの書きつぶしや，コミット漏れ，二重のコミットなどが起こり，致命的である．
- 特に commit ポインタにフォールトが起こった場合には，（前述したように，そもそもコミットの停止が間に合わないということもあるが）tail ポインタが誤った commit ポインタの内容によって上書きされるだけで，正しい状態にはならない．

制御系のタイミング・フォールトと個別的アプローチ

このような説明をすると，個別的に対処すればよいという意見をよく耳にする．すなわち，フォールトが起こると対処が困難なロジックを残らず列挙し，フォールトが起こらないように 1 つ 1 つ最適化すればよいというわけである．そのような個別的アプローチは，まったく不可能というわけではないが，以下の理由により，現実的とはいえない：

- まず，そのようなロジックは数多い．Out-of-order スーパスカラ・プロセッサでは，リネーム・ロジック，スケジューリング・ロジック，そして例にあげたコミット・ロジックなどの主要なモジュールが，それぞれ複数のキューやテーブルによって構成されている．これらのキューやテーブルにほぼ例外なく付随するポインタ/カウンタも，パイプライン・フラッシュによっては回復されない．そして，これらのキューやテーブルの方が，演算器などよりはるかに設計量が多い．

[*2] 当然，循環について考える必要がある．

- さらに，これらのロジックのうちのいくつかはシステムのクリティカル・パスを含む可能性が高い．そのようなロジックでフォールトが起こらないようにするためには，ディープ・パイプライニングによって遅延を大幅に短縮したうえでレイテンシの増加による性能低下を緩和する方策を施すなど，マイクロアーキテクチャの大幅な変更が必要となる．
  しかし，すべてのロジックについてそのような方策が知られているわけではない．少なくとも，通常のデザインからフォールト対応済みのデザインへと自動的に変換することは不可能となる．
- たとえ可能であったとしても，このような個別的アプローチは手法の包括性を著しく毀損する．耐故障性や信頼性では手法の包括性が肝要であり，2.3 節では，アーキテクチャ・レベルの回復技術は包括性が高いことがメリットであると述べた．このような個別的アプローチは，この手法の包括性を著しく毀損するものである．

3.7　考察：データ・パスと制御系のフォールト

データ・パス上に発生したフォールトは確かに，ゼロ除算などと同様の**例外**として扱うことができる．プロセッサを 1 つの状態機械と考えると，データ系にフォールトが生じたとしても，設計者の想定した既知の状態の 1 つと考えることができるからである．たとえば，64 bit の実行結果に 1 bit のエラー・フラグを加えて 65 bit のデータと見なせばよい．65 bit 目が 0 であっても 1 であっても，設計者の想定の範囲内である．Razor Ⅱ の回復技術の考え方は，このようなもとの解することができる．

一方，例にあげたポインタのような**制御系**のレジスタにフォールトが生じた場合，それはもはや設計者の想定した状態ではない．パイプライン・フラッシュは，当然のことながら，既知の状態から別の既知の状態への遷移させる操作であって，未知の状態に陥ったものを既知の状態へ遷移させることはできないのである．

実際には，スカラ・プロセッサであっても，例にあげたポインタのような制御系レジスタを持つ．実用的なスカラ・プロセッサには，ストア・バッファなどのバッファが不可欠であり，そのようなバッファには制御系レジスタが存在する．結局，Razor Ⅱ の回復技術は，少なくともそのままでは，文献 [4] で例示されているようなストア・バッファすら持たないごく単純なプロセッサにしか適用できないのである．

4.　コミット

前章では，Razor Ⅱ の回復技術は実用的なプロセッサに適用できないことを述べた．本章以降では，前章までの問題を解決し，複雑な out-of-order スーパスカラ・プロセッサにも適用できる技術を提案する．

Razor Ⅱ の回復技術の問題点は，「ROB/LSQ の commit ポインタの更新がコミットとなる」という規約に起因するといえよう．そこでまず本章では，コミットについてさらに深く考察を加え，これに代わる合理的な規約を探す．

提案手法の詳細については，次章でまとめる．

4.1　コミット・パイプラインの概要

近年の out-of-order スーパスカラ・プロセッサのコミットにおいては，commit ポインタの更新から，LRF/L1D のデータ・アレイの更新まで，数サイクルかかるのが普通である．この部分は，**コミット・パイプライン**と呼ぶことができる．通常，スーパスカラ・プロセッサは，フロントエンドとバックエンドの 2 つのパイプラインを持つと説明されるが，フロントエンド，バックエンド，そして，コミットの，3 つのパイプラインを持つと理解した方がよい．

コミット・パイプラインは，一般に，**開始**～**PNR**（Point of No-Return）～**終了**に区切ることができる．開始，PNR，終了は，ステージではなく，ステージとステージとの境界であることに注意されたい：

開始　は，一般に，ROB/LSQ の commit ポインタの更新と考えてよい．

PNR　は，以下のように定義される：名前のとおり，ここを超えた命令は，もはや超える前の状態に戻すことができず，必ず LRF/L1D を更新しなければならない．

終了　は，LRF/L1D を構成する RAM のデータ・アレイへの書き込みの終了である．

前章までで述べたような一般的な ROB/LSQ の構成では，commit ポインタの更新が，コミットの開始と同時に PNR ともなっていると解することができる．これは，「commit ポインタの更新をコミットとする」という規約に対応するものである．図 3（上），および，図 4 の中に示した PNR は，このことに対応する．

本稿では，「commit ポインタの更新は単なる開始であって，PNR の通過を本当のコミットとする」という，より一般化された規約を導入し，実際に開始と PNR を分離する方法を探す．

4.2　コミットの開始，PNR，終了とプログラム・オーダ

コミットの終了は out-of-order である．一方で，すべての命令はコミットの開始と PNR を in-order に通過しなければならない．

コミットの終了

コミットの終了は out-of-order である，すなわちコミットの終了は，以下のように，ROB → LRF 系と LSQ → L1D 系で異なる：

- LRF と L1D を構成する RAM の書き込みに要するサ

イクル数は一般に異なる．

- ストア命令は，ストア・バッファにおいて待たされる．

より厳密には，後続の命令がLRF/L1Dから値を読み出せるようになった時点をもってコミットの終了と定義すると便利であろう．そのように定義すると，「ROB/LSQのエントリを開放し，命令をリタイアするのは，コミットの終了以降」ということができる．

コミットの開始

コミットの開始，すなわち，commitポインタの更新は，3.2節で述べたように，必ずin-orderに行われる．これは，コミット可能な命令をプログラム・オーダ上で探す操作だからである．

コミットのPNR

コミットのPNRを超えた命令は絶対にLRF/L1Dを更新しなければならないという定義から，すべての命令はPNRをin-orderに通過しなければならない．

4.3　命令パイプラインと命令グループ

提案の第1のポイントは，コミットの開始とPNRを分離することにある．そのためにはまず，命令パイプラインについて精密に理解する必要がある．

フロントエンド・パイプラインとフェッチ・グループ

同時にフェッチされたn個の命令からなるグループは**フェッチ・グループ**と呼ばれる．フェッチ幅をnとすると，フロントエンド・パイプラインはn本の**レーン**からなると考えてよいであろう．フェッチ・グループのn個の命令は，n本のレーンにそれぞれ投入される．フロントエンド・パイプラインでは，このフェッチ・グループが**分解**（**break**）されることなく，保たれたまま下流へと運ばれる．すなわち，n本のレーンはすべて互いに同期しており，いずれかのレーンだけ先に進む，あるいは，停止するなどいうことはない．すなわち，フェッチ・グループはin-orderに形成され，そのままin-orderに保たれる．

フェッチ・グループを保つ性質は，特に，レジスタ・リネーミングのために重要である．レジスタ・リネーミングが可能であるのは，リネーミング・ステージにおいてフェッチ・グループが保たれているからである．その結果，グループ内，グループ間の命令の並びから，プログラム・オーダを容易に再現することができるのである．

バックエンド・パイプラインと発行グループ

バックエンド・パイプラインにおいて，同時に発行された命令からなるグループは**発行グループ**と呼ばれる．フェッチ・グループとは異なり，発行グループはout-of-orderである．

バックエンド・パイプラインも，フロントエンド・パイプラインと同様，発行グループを保つ性質を持つ．バックエンド・パイプラインでは，この性質がout-of-order命令スケジューリングを可能としているといえる．端的にいえば，依存元と依存先の命令のレーン上での相対位置関係が，スケジューリング時と実行時で変わらないからである[13]．

コミット・パイプラインとコミット・グループ

同様にして，commitポインタの更新により，あるサイクルに新たにcommitポインタよりhead側に移った命令からなるグループは，**コミット・グループ**と呼ぶことができる．コミット・グループは，in-orderである．

しかし，コミット・パイプラインにはコミット・グループを保つ性質がない．それは，コミット・パイプラインのLSQ → L1D側にはストア・バッファがあるからである．ストア・バッファの役割は，L1Dのポートの不足のためストア・バッファの下流が停止したときにも，ストア・バッファの上流は停止させないことである．まさにこのとき，ROB → LRF側だけが先に進み，コミット・グループは分解されてしまう．すなわち，コミット・パイプラインのストア・バッファより下流ではコミット・グループはin-orderに保たれないのである．

4.4　コミットの開始とPNRの分離

すべての命令はPNRをin-orderで通過しなければならないが，ストア・バッファより下流ではin-orderに保たれない．したがって，PNRはストア・バッファの入り口（もしくは，その上流）となる．そして，コミットの開始で形成されたin-orderなコミット・グループは，PNR＝ストア・バッファ入り口まで保持されなければならない．

これらの制約を満たしたうえで，コミットの開始とPNRを分離するには，以下のようにすればよい．図3（中）に，提案手法のパイプラインの全体像を示す：

(1) まずストア・バッファをLSQから別体化する．
(2) 次に，LSQからstoreポインタを廃して，ROBと同様の制御とする．すなわち，LSQ上でcommitポインタよりhead側に移された次のサイクルに必ずLSQから読み出すと，ROB/LSQのcommitポインタの更新によって形成されたコミット・グループは，グループが保たれたまま，次のサイクルにROB/LSQから読み出されることになる．
(3) 別体化されたストア・バッファまでコミット・グループを転送するパイプラインを構成すれば，そのパイプライン上ではコミット・グループは保たれる．
(4) そこで，そのパイプライン上，ストア・バッファへのエンキューの直前をPNRとし，その直前にスタビライズ・ステージを設ければよい．PNRを超えたコミット・グループ内の命令はすべてコミットすれば，アーキテクチャ・ステートはin-orderに保たれる．ROB/LSQのポインタなどにフォールトが発生した場合にも，影響を受けたコミット・グループがスタビライズ・ステージを通過する間に，LRF/L1Dへの書き込みを停止することができる．

ただし，別体化されたストア・バッファは，PNR より下流であるため，ここでフォールトを起こしてはならないという制約が課せられる．次章以降で詳しく述べるが，ストア・バッファは実際には 2 エントリ程度の単純なシフト・レジスタで構成しても性能低下はほとんどない．

5. 提案手法の詳細

前章では，「コミットの PNR の通過が本当のコミットとなる」というより一般化された規約を導入し，ストア・バッファを ROB/LSQ から別体化することで PNR を ROB/LSQ の外部に移動できることを述べた．これを受けて本章では，提案手法の詳細について述べる．

5.1 提案手法の特徴

Razor Ⅱ の回復技術との違いを際立たせる提案手法の特徴は，以下の 3 つの “I” によって表すことができる：

In-Order Passing through PNR　前章で述べた方法でストア・バッファを別体化すれば，すべての命令は ROB/LSQ の外部にある PNR を in-order で通過することが保証される．

Imprecise Cancellation　フォールトの影響を受けた命令を正確に特定する必要はなく，影響を受けた可能性のある命令が PNR を超えないようにすれば，アーキテクチャ・ステートはフォールトから保護される．

Initialization of Pipeline above PNR　フラッシュではなく，PNR より上流を初期化することによってフォールトの影響を取り除く．

前章では PNR を ROB/LSQ から外部化する方法について述べた．このことによって，In-Order Passing through PNR が可能になる．以下，5.2 節と 5.3 節で，Imprecise Cancellation と Initialization of Pipeline above PNR について述べる．別体化ストア・バッファについては，5.4 節で詳述する．

5.2 Imprecise Cancellation

3.2 節で述べたバッファのポインタの例などを考えれば，out-of-order スーパスカラ・プロセッサではフォールトの影響を受けた命令を正確に特定することは原理的に不可能である．実際には，正確に特定する必要などそもそもない．影響を受けた可能性のある命令が PNR を超えないようにするだけで，アーキテクチャ・ステートはフォールトから保護される．

コミットの停止と命令のキャンセルは，Razor Ⅱ の回復技術のようにフォールトを受けた命令の到着を待って行う必要はない．フォールトの発生が判明したら，PNR を超えていない命令は，フォールトの影響を受けたか受けないかにかかわらず，速やかにキャンセルしてしまってよい．

実際 out-of-order スーパスカラ・プロセッサの場合，フォールトの影響を受けた命令よりも，エラー信号の方が先に PNR に到達することになる．これは，エラー通知ネットワークが単純なパイプラインで構成されるのに対し，命令は命令ウィンドウでバッファリングされるからである．

アーキテクチャ・ステートを保護するための条件は，フォールトの影響を受けた可能性のある命令がエラー信号より先に PNR に到達しないことのみである．パイプライン上で最後にエラー信号が生成されるのは，ROB/LSQ のポインタの更新時と，ROB/LSQ の読み出し時であるから，この部分の前後関係だけに注意すればよい．

エラー信号が PNR に到達したら，コミット・グループの PNR の通過を停止する．すでに PNR を通過した命令は数サイクル以内に LRF/L1D の更新を終了する．これで，LRF/L1D の状態は，PNR を通過した命令まではすべて実行され，それ以降の命令は 1 つも実行されていない，in-order なアーキテクチャ・ステートとなる．

5.3 Initialization of Pipeline above PNR

フラッシュではなく，PNR より上流を初期化することによってフォールトの影響を取り除く．初期化のロジック自体は（パワーオン）リセットと同じでよい．初期状態のパイプラインが，LRF/L1D に保存された in-order なアーキテクチャ・ステートから実行を再開することになる．

フォールトの影響の除去

制御系のレジスタには，フラッシュ用と 初期化用に別々のロジックが存在する．たとえば，3.2 節，3.3 節で述べた ROB/LSQ のポインタの場合，以下のようになる：

フラッシュ　3.4 節で述べたように，フラッシュでは，tail = commit と tail ポインタを commit ポインタの内容で上書きすると同時に，count を tail − commit の分だけ減らす．

そのため，commit ポインタがフォールトによって誤った値となっていた場合，フォールトの影響はむしろ tail ポインタへと拡大するだけで，フォールトの影響を取り除くことはできない．

初期化　それに対して初期化は，すべてのポインタ，カウンタの値を 0 にすることになる．

当然のことながら，フォールトの影響は完全に除去できる．

初期化のコスト

初期化の対象となるレジスタはフラッシュと同じであるが，ロジックはフラッシュより簡単であるため，フラッシュと同程度以下のコストで実現できる．

Fan-out の低減のため，パイプラインの上流から順に初期化を行うネットワークを構築するとなおよい．逆に，このネットワークを（パワーオン）リセット時のパイプラインの初期化に用いることもできる．

5.4　フォールト・フリー・ストア・バッファ

前章で述べたように，別体化されたストア・バッファは，PNR より下流であるため，ここでフォールトを起こしてはならない．フォールト・フリーは，以下の 2 点によって達成される：

(1) フォワーディング機能の省略
(2) シフト・レジスタベースの構成

この 2 点について，以下で詳しく述べる．

(1) フォワーディング機能の省略

3.3 節において，ストア・バッファの役割にはバッファリングとフォワーディングがあると述べた．しかし，別体化されたストア・バッファでは，LSQ がフォワーディングの機能を果たすため，ストア・バッファはバッファリングの機能のみを果たせばよい．フォワーディングの機能は複雑であり，これをフォールト・フリーとすることは現実的ではない．しかし，バッファリングの機能のみであれば十分に単純である．

ストア・バッファ内にエントリを持つ命令は，フォワーディングのため，LSQ にもエントリが確保されたままになっていることに注意されたい．ストア・バッファ内のエントリは，L1D の更新を開始してもおらず，これらのストア命令の値は，L1D からではなく，LSQ 内のエントリから供給される．

今ここで，上流で別の命令のフォールトが検出されたとしよう．このときには，LSQ 内の命令は前述のとおり初期化によってキャンセルされる．一方で，ストア・バッファ内のエントリは，PNR を通過したのだから，キャンセルされることなく，L1D を更新することになる．

(2) シフト・レジスタベースの構成

さらに，データ・パスに関しては，シフト・レジスタをベースとする構成を提案する．図 3（下）に，提案の構成を示す．同図のように，データ・パスの構成は実質的にパイプライン・ラッチの並びと同じであるが，制御によってバッファの機能を果たす．すなわち，たとえば，L1D のポートがロード命令によって使用されるために出口が塞がっていたとしても，入り口側のレジスタが空いていればエンキューすることができる．

データ・パスの構成はパイプライン・ラッチの並びと等価であるので，フォールトの発生確率はスタビライズ・ステージと同程度であり，無視できる．さらに，図 3（下）のように，ストア・バッファ内の最初のステージをスタビライズ・ステージとすることができる．なお，この場合の最小のエントリ数は 2 となる．

ただしシフト・レジスタベースの構成では，エントリ数と同じサイクル数だけ，エンキューからデキューまでのレイテンシが長くなる．このことが IPC に与える影響についても，次章で評価する．

表 1　プロセッサの構成
Table 1　Configuration of simulated processor.

パラメータ	値
ISA	Alpha 21164A
fetch width	4 inst.
commit width	6 inst.
exec units	int: 2, fp: 2, mem: 2
inst window	int: 32, fp: 16, mem: 16
load/store queue	Unified, 32 entries
load/store ports	1-read + 1-read/write, cycle stealing
branch pred	8 KB g-share
miss penalty	10 cycles
BTB	2K-entry, 4-way
L1D	32 KB, 4-way, 64B/line, 3 cycles
L2C	4 MB, 8-way, 64B/line, 15 cycles
main memory	200 cycles

5.5　提案手法の包括性

最後に，提案手法の包括性をまとめる：

- PNR より上流で発生するフォールトが PNR を超えることはない．さらに，パイプラインの初期化によって，PNR より上流のフォールトの影響は完全に除去される．
- したがって，PNR より下流がフォールト・フリーであることを確かめればよい．別体化されたストア・バッファに関しては，フォールトが起こる確率は十分に低いことを示した．

このように，提案の回復技術は out-of-order スーパスカラ・プロセッサのフォールトについて包括的に扱っているということができる．

6.　IPC の評価

従来の LSQ 内部にストア・バッファが含まれるモデルに対して，提案手法のように LSQ からストア・バッファを別体化するとことによる IPC の低下を，シミュレーションにより評価した．

6.1　評価モデルと評価環境

プロセッサ・シミュレータ**鬼斬弐** [14] によって，SPEC CPU2006 [15] を用いて評価を行った．**表 1** に，ストア・バッファ以外のパラメータをまとめる．

ストア・バッファに関しては，以下で述べる 3 種のモデルを実装した：

BASE　3.3 節で述べた，LSQ の一部をストア・バッファとして用いるモデル．

RINGB　リング・バッファで構成された一般的な別体ストア・バッファを持つモデル．エンキューとデキューのためにそれぞれ 1 サイクル，合計 2 サイクル BASE よりレイテンシが長くなる．

SHIFTR　提案手法．5.4 節で述べた別体ストア・バッ

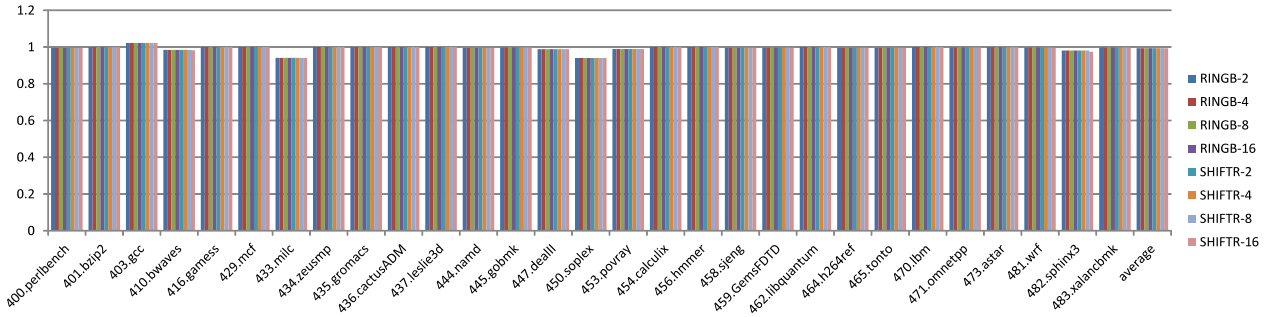

図 5　ベースラインに対する相対 IPC
Fig. 5　Normalized IPC to the BASE model.

ファを持つモデル．エントリ数と同じサイクル数だけ BASE よりレイテンシが長くなる．

ストア・バッファのサイズは，BASE では LSQ の一部であるため，最大で 32 エントリとなる．RINGB と SHIFTR では，2, 4, 8, 16 の場合についてそれぞれ評価した．5.4 節で述べたように，SHIFTR の最小サイズは 2 である．

6.2　評価結果

ストア・バッファの別体化は，以下のようにして性能低下を招く：

レイテンシの増加　レイテンシの増加の分だけ LSQ のエントリの解放が遅れ，資源の不足によりフロントエンド・パイプラインがストールする確率が上昇する．

スループットの低下　別体化によって，LSQ →ストア・バッファへのスループットが低下する．BASE では，LSQ 内部の移動であるため，スループットはコミット幅（= 6）によって決まる．RINGB と SHIFTR では，LSQ からの読み出しとストア・バッファへのエンキューが必要となる．

LSQ →ストア・バッファのスループットは，ストア・バッファ→ L1D の書き込みのスループットと，平均としては同じでよい．なお，ストア・バッファ→ L1D の書き込みのスループットは，L1D の書き込みポート数で与えられ，この評価では 1 である．

IPC

図 5 に，BASE に対する RINGB と SHIFTR の相対 IPC を示す．相対 IPC の低下は平均は 0.7%にとどまっており，ストア・バッファを別体化することによる性能低下が十分小さいことを示している．

性能低下が大きい milc，soplex では，それぞれ，5.9%，6.0%である．そこで，BASE において LSQ のコミット幅を 1 に制限したところ，性能低下はちょうど 5.9%，6.0%であった．したがって，RINGB，SHIFTR の性能低下は，もっぱら LSQ →ストア・バッファのスループットの低下によるものと推定される．前述したように，LSQ →ストア・バッファのスループットは，ストア・バッファ→ L1D の書き込みのスループットと，平均としては同じでよい．

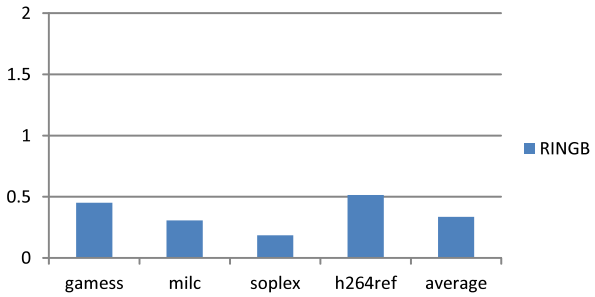

図 6　ロード命令のサイクルあたり平均ポート利用数
Fig. 6　Average utilization of the port per cycle by load instructions.

しかしこれらのプログラムでは，ストアを多数連続して実行する部分があり，そのためスループットの影響が生じたものと考えられる．

RINGB，SHIFTR におけるレイテンシの増加の影響はほとんど見られない．モデル SHIFTR-16 の，sphinx3 においてわずかな低下が見られるだけである．

それ以外には目立った性能低下はなく，SHIFTR のサイズは，最小である 2 エントリで十分である．

サイクル・スチール

以下では，SPEC CPU2006 の中で特徴的な 4 種のプログラムを抽出し，全 29 種のプログラムの平均値とともに示す．

図 6 では RINGB でストア・バッファエントリ・サイズが 2 の場合について，ロード命令のサイクルあたりの L1D ポート利用数を，それぞれ示す．ロード命令のサイクルあたりの L1D ポート利用数は平均でも 0.3，最大の h264ref でも 0.6 と低い値となっている．この結果が示すように，L1D のポートをロード命令が占有している（サイクルあたり 2 命令）確率は十分に低く，ほぼ毎サイクル，サイクル・スチールすることができるため，LSQ の資源が足りなくなるほど多くの命令がストア・バッファにとどまる確率は低い．

図 7 に，ストア命令のストア・バッファ内の平均滞留サイクル数を示す．RINGB，SHIFTR とも，ほとんどの場合において，ストア・バッファを通過するのに最低限必要なサイクル数程度しかかからないことを示している．たとえば，

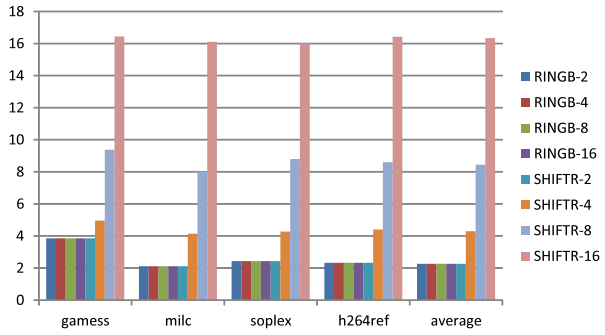

図 7　ストア・バッファの平均滞留サイクル数

Fig. 7　Average cycles that an instruction stays in the store buffer.

SHIFTR で 2 エントリのモデルでは，平均 2.3 サイクル，最悪でも 3.8 サイクル となっている．これは，1 ポートのサイクル・スチールで十分な性能が得られることを裏書きしている．

7.　おわりに

近年問題となりつつある LSI のばらつきへの対策として，タイミング・フォールトの検出/回復技術がある．しかし，既存の回復技術では，out-of-order スーパスカラ・プロセッサのコミットへの考慮が不十分であり，また，制御系のフォールトから正しく回復することができないため，ごく単純なスカラ・プロセッサに適用範囲が限られていた．本稿は，そのような技術の 1 つとして，複雑な out-of-order スーパスカラ・プロセッサにも適用可能な回復技術を提案した．

提案手法では，「ROB/LSQ の commit ポインタの更新をコミットとする」という規約に代えて，「コミットの PNR の通過を本当のコミットとする」というより一般化された規約を導入し，LSQ からストア・バッファを別体化することで PNR を ROB/LSQ の外部に移動する．また，別体化したストア・バッファは PNR の下流に位置し，フォールトが起こってはならないため，単純なシフト・レジスタでこれを構成する方法を提案した．シミュレーションによって，ストア・バッファの別体化による IPC への影響は十分に小さいことを確認した．

現在，研究室では，NORCS [13] など様々な技術を取り入れた高効率な out-of-order スーパスカラ・プロセッサの開発を行っており，これに提案手法を組み込んでいる．このプロセッサが完成すれば，実機によって提案手法のタイミング・フォールト耐性を確認することができるであろう．

謝辞　本研究の一部は，JST CREST「ディペンダブル VLSI システムの基盤技術」，および，文部科学省科学研究費補助金 No.23300013 による．

参考文献

[1] 岡田健一： 集積回路における性能ばらつき解析に関する研究，博士論文，京都大学 (2003).
[2] 平本俊郎，竹内　潔，西田彰男：MOS トランジスタのスケーリングに伴う特性ばらつき，電子情報通信学会誌，Vol.92, No.6 (2009).
[3] Mukhopadhyay, S., Mahmoodi, H. and Roy, K.: Modeling of Failure Probability and Statistical Design of SRAM Array for Yield Enhancement in Nanoscaled CMOS, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.24, No.12 (2005).
[4] Blaauw, D., Kalaiselvan, S., Lai, K., Ma, W.-H., Pant, S., Tokunaga, C., Das, S. and Bull, D.: Razor II: In-Situ Error Detection and Correction for PVT and SER Tolerance, *Int'l Solid-State Circuits Conference* (*ISSCC*) (2008).
[5] Fojtik, M., Fick, D., Kim, Y., Pinckney, N.R., Harris, D.M., Blaauw, D. and Sylvester, D.: Bubble Razor: An Architecture-Independent Approach to Timing-Error Detection and Correction, *ISSCC*, pp.488–490 (2012).
[6] Choudhury, M., Chandra, V., Mohanram, K. and Aitken, R.: TIMBER: Time borrowing and error relaying for online timing error resilience, *Design, Automation Test in Europe Conference Exhibition* (*DATE*), *2010*, pp.1554–1559 (2010).
[7] Tiwari, A., Sarangi, S.R. and Torrellas, J.: ReCycle: Pipeline Adaptation to Tolerate Process Variation, *ISCA*, pp.323–334 (2007).
[8] 杉本　健，入江英嗣，五島正裕，坂井修一：タイミングエラー耐性を持つスーパスカラプロセッサ，電子情報通信学会研究報告 CPSY2008-14 (SWoPP), pp.19–24 (2008).
[9] Ernst, D., Kim, N., Das, S., Pant, S., Pham, T., Rao, R., Ziesler, C., Blaauw, D., Austin, T. and Mudge, T.: Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation, *Int'l Symp. on Microarchitecture* (2003).
[10] Austin, T., Blaauw, D., Mudge, T. and Flautner, K.: Making Typical Silicon Matter with Razor, *IEEE Computer* (2004).
[11] Das, S., Roberts, D., Lee, S., Pant, S., Blaauw, D., Austin, T., Flautner, K. and Mudge, T.: A Self-Tuning DVS Processor using Delay-Error Detection and Correction, *IEEE J. of Solid-State Circuits* (2006).
[12] Shen, J.: *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill Science/Engineering/Math (2004).
[13] Shioya, R., Horio, K., Goshima, M. and Sakai, S.: Register Cache System not for Latency Reduction Purpose, *Int'l Symp. on Microarchitecture* (*MICRO-43*), pp.301–312 (2010).
[14] 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).（ポスター）.
[15] The Standard Performance Evaluation Corporation: SPEC CPU2006 suite, available from ⟨http://www.spec.org/cpu2006/⟩.

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．2004 年博士（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．コンピューティング・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．

倉田 成己　（学生会員）

1987 年生．2010 年東京大学工学部電子情報工学科卒業．2012 年同大学大学院情報理工学系研究科電子情報学専攻修士課程修了．同年同専攻博士課程に進学，現在に至る．プロセッサアーキテクチャの研究に従事．2010 年情報処理学会推奨卒業論文に認定．

塩谷 亮太　（正会員）

1981 年生．2006 年東京大学工学部電子工学科卒業．2008 年同大学大学院情報理工学系研究科電子情報学専攻修士課程修了．2009 年より日本学術振興会特別研究員．2011 年東京大学大学院情報理工学系研究科電子情報学専攻博士課程修了，博士（情報理工学）．同年より名古屋大学大学院工学研究科助教，現在に至る．コンピュータアーキテクチャの研究に従事．2011 年 IEEE Computer Society Japan Chapter Young Author Award 受賞．電子情報通信学会，IEEE 会員．

坂井 修一　（フェロー）

1958 年生．1981 年東京大学理学部情報科学科卒業．1986 年同大学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．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），大川出版賞（2012）等受賞．情報処理学会フェロー．電子情報通信学会フェロー．人工知能学会，ACM，IEEE 各会員．日本学術会議連携会員．