Vol.2010-ARC-190 No.10
2010/8/3

過渡故障耐性を持つ Out-of-Order スーパスカラ・プロセッサのコミット方式

有馬 慧[†1] 岡田 崇志[†1] 塩谷 亮太[†1,†2]
五島 正裕[†1] 坂井 修一[†1]

半導体プロセスが微細化するにつれて，ばらつきの問題が深刻化してきている．ばらつきを吸収するために，近年，動的なタイミング・フォルトを動的に検出・回復する技術が提案されている．そのひとつに，リセットをベースにした回復手法がある．本研究では，Out-of-order スーパスカラ・プロセッサのコミット・ステージに着目し，この手法をより信頼のあるものへ改良する．

Commit Scheme for Transient-Fault-Tolerant Out-of-Order Superscalar Processors

Satoshi ARIMA,[†1] Takashi OKADA,[†1]
Ryota SHIOYA,[†1,†2] Masahiro GOSHIMA[†1]
and Shuichi SAKAI [†1]

The feature size of LSI is getting smaller year by year, increasing random variation between the elements. To overcome the problem of the variation, there are researches about detecting and recoverying from runtime timing-faults, and one of them is based on reset function. We focus attention on commit stage of out-of-order superscalar processors, and improve this reset-based scheme.

†1 東京大学 情報理工学系研究科
Dept. of Information and Communication Eng, the Univ. of Tokyo
†2 日本学術振興会特別研究員 (DC2)
Research Fellow of the Japan Society for the Promotion of Science(DC2)

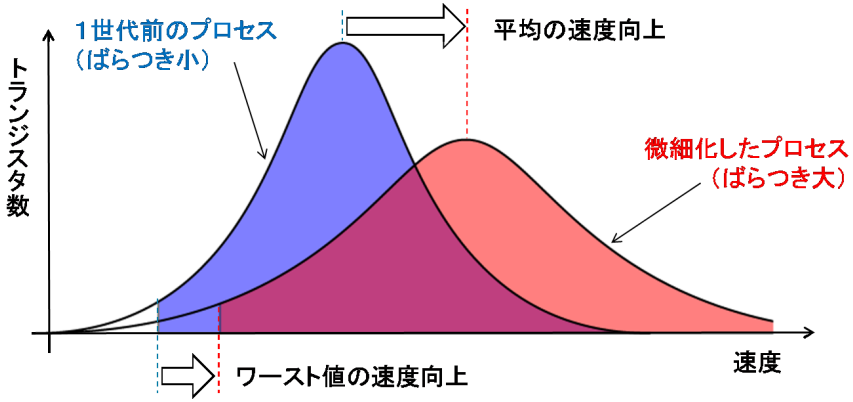

図 1 プロセスの微細化と製造ばらつき

1. はじめに

近年のハードウェア設計において，半導体素子の微細化に伴うばらつきの増加が大きな問題となっている．例えば，半導体素子の製造の過程で，微細化による露光精度の低下，エッチングや平坦化などの工作精度の低下によって，各個体の加工寸法，不純物密度分布がばらつく．微細化された素子が原子のサイズに近いことにより，不純物原子１つあたりの相対的な影響が大きく，例えばこれらの要因により，素子の性能（遅延等）が大きくばらつく (製造ばらつき (図 1))．他にも，製造時に静的に生じるばらつきだけでなく，動作時の局所的な電源電圧や温度の違いによるばらつきといった，動作環境の違いで生じるばらつきもある．

図１は２つの異なる半導体製造プロセスで製造されるトランジスタの速度による分布を示している．微細化によって速度の平均値は向上しているが，ばらつきの増大が原因で，ワースト値の速度向上は平均値の速度向上より小さい．

図２は，この様子をプロセスの世代に沿って見たものである．微細化が進むことで，平均性能は向上しているが，ばらつきが増大することで，ワースト値の向上はだんだんと見込めなくなりつつある．

従来，プロセッサはワースト値に基づくワースト・ケース設計を行っている．ワースト値に基づくことで，動的なタイミング・フォールト (動的 TF) を起こさない．

動的 TF とは，回路遅延の動的な変化によって，回路の遅延制約を満たさない誤った出力値がサンプリングされる過渡故障である．たとえば図 3 では，遅延の動的な増加によって動

© 2010 Information Processing Society of Japan

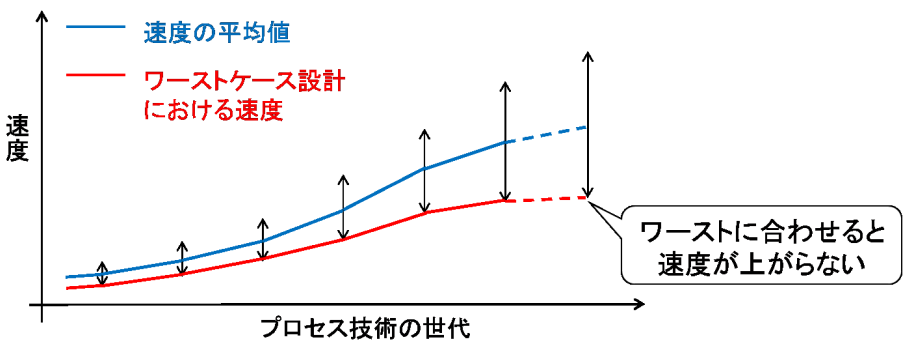

図 2 プロセス世代とワースト・ケース設計

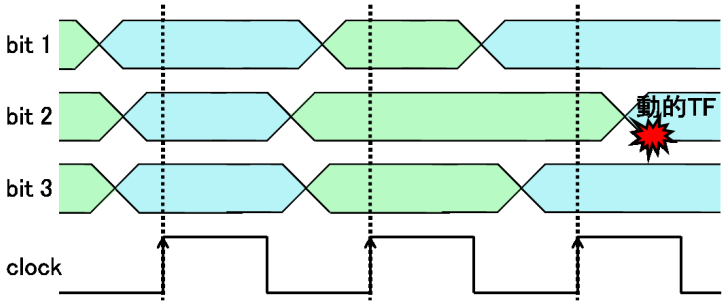

図 3 動的なタイミング・フォールト

的 TF が生じている．

ワースト・ケース設計では，最悪な素子が最悪な環境 (電源電圧や温度) にあってもプロセッサが正しく動作することを保証する (動的 TF が起こる個体は出荷検査で取り除かれ，出荷後の個体では熱暴走などを起こさない限り動的 TF が起きない) が，確率的にほとんど起きないようなワースト環境のために，ほとんどの動作には大きすぎるタイミング・マージンを確保するため，悲観的すぎる．

そこで，ばらつきの増大に対応するために，ワースト値に基づかない，実際の遅延に基づいた動作を行う手法が提案されている．たとえば，統計的静的タイミング解析 (SSTA:Statistical Static Timing Analysis)[1] や出荷検査時のスピード・グレード選別などがある．

SSTA では，ばらつきを統計的に取り扱うことで，悲観的になりすぎない見積りをし，スピード・グレード選別では，出荷時検査で実際の個体の性能を評価することで，製造ばらつきを吸収する．しかしこれらの手法は，動作時におこるばらつきに関しては依然としてワースト・ケースで見積もることになる．

動作時のばらつきまで含めて，実際の遅延に基づく手法として，DVFS(Dynamic Voltage and Frequency Scaling) と動的 TF 検出/回復・予報技術とを協調させる手法が提案されている[2]．

DVFS は動的に動作周波数や電源電圧を変化させる技術であり，通常は動的 TF が起こらない範囲で周波数や電圧を設定する．この範囲を超えて動作周波数を上げたり電源電圧を下げたりしていくと，どこかで動的 TF が発生する．

動的 TF 検出/回復・予報技術の代表的な既存の研究として，Razor[3),4)] やカナリア FF[5)] がある．Razor は動的 TF が起こると，それを FF で検出し，エラー信号を出す．また，レジスタやキャッシュへの書き込みステージにエラー信号を伝達することで，誤った書き込みを防ぎ，その上で命令を再実行することで回復を図る．詳しくは”2” で説明する．カナリア FF は動的 TF が起こりそうな状態を検出し，動的 TF が起こることを未然に防ぐ手法である．

これらの技術を協調させることで，動作時のばらつきを吸収することができる．具体的には以下のようにする．従来の DVFS での動的 TF が起こらない許容範囲を超えて，動作周波数を高く (または，電源電圧を低く) する．しばらく動的 TF が起こらなかったら，さらに動作周波数を高く (または，電源電圧を低く) していく．動的 TF の検出 (または予報) があったら，動作を止め，正しく回復して，少し動作周波数を下げて (または，電源電圧を高くして) 動作を再開する．これを繰り返すことで，見積りではない実際の遅延に合わせた動作が可能になり，高クロック化や省電力を達成することができる (図 4)．

動的 TF からの回復に関する既存の研究では，データ・パス上での動的 TF からの回復を行っているが，制御系パス上での動的 TF からの回復が考慮されていない．近年の Out-of-order スーパスカラ・プロセッサでは，クリティカルな制御系パスも数多く存在し，この点は大きな問題となる．

我々は制御系パス上での動的 TF を考慮した回復手法を提案してきた[6),7)]．この手法では，プログラム順でこの命令までは正しく動作をした，というプロセッサ状態を記憶しておき，動的 TF が起こると，正しいプロセッサ状態以外をリセットすることで動的 TF による誤りを排除し，その後そのプロセッサ状態から動作を再開することで回復を図る．

しかし，プロセッサ状態を更新するロジックで起こる動的 TF を考慮しておらず，回復の

© 2010 Information Processing Society of Japan

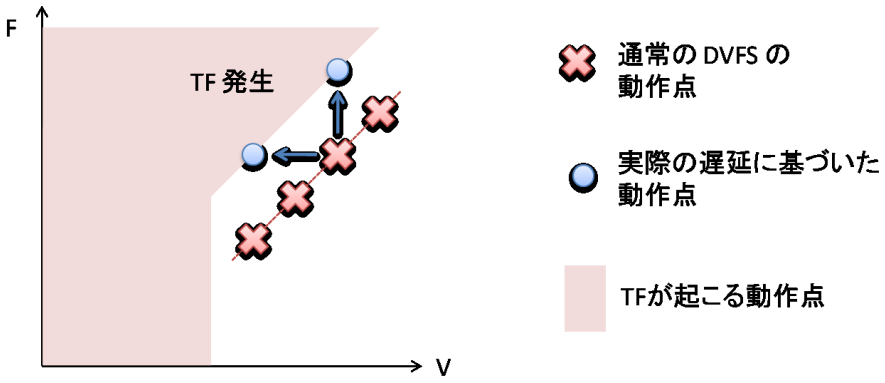

図 4 DVFS と動的 TF

要となる正しいプロセッサ状態が実現できない可能性が残っており，これが問題となっている．

本稿では，プロセッサ状態の更新部分に着目し，この部分で動的 TF が発生した場合にも正しいプロセッサ状態を実現する手法を提案する．

2. 関連研究：RazorII

動的 TF の検出/回復技術の代表的な既存研究として，前節で少し述べた Razor[3),4)] についてここで紹介する．

2.1 動的 TF 検出

RazorII FF(図 5) は，単相ラッチをベースに動的 TF 検出機構を付加している．このラッチは，CLK のポジティブ・エッジで入力 (D) をインターバル・ループ (N) に取り込む．同時に DCgenerator によって DetectionClock(DC) パルスが発生する．DC が再び High になる前に N の状態遷移が完了すれば，これは正常な遷移とみなされる (図左下)．しかし，DC が High になった後に N の状態遷移が完了すると，これは動的 TF として検出される (図右下)．この際，ERRORbit がアサートされ，検出を報せる．

CLK が Low のときは N の状態が保持される．つまり，DC のポジティブ・エッジから CLK のネガティブ・エッジまでが動的 TF の検出期間である．RazorII FF のタイミング制約は，$T_{ON} + h \sim 1 + DC \leq 1 + T_{ON} + h$ となる．ただし，T_{ON} は CLK が High である時間，h は hold-time，DC は DC パルス幅を表す．遅延の検出範囲には限界があり，

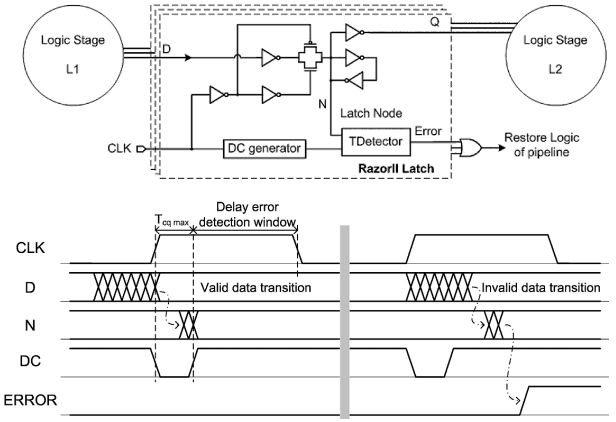

図 5 RazorII flip-flop and conceptual timing diagrams[4)]

$1 + T_{ON} + h$ を超える遅延は検出できない．

この検出ロジックの本質は，正しい状態遷移期間を限定することである．そのため，動的 TF だけでなく SEU(Single Event Upset:宇宙線によって bit が反転する過渡故障) による不正な状態遷移も同じロジックで検出できる．

2.2 動的 TF からの回復

動的 TF が起こると，プロセッサは想定した計算を正しく行えない．動的 TF が検出されたら，その誤った状態から正しい状態に回復する必要がある．動的 TF からの回復方法は，その検出方法に依存する．ここでは，RazorII FF による検出を利用した動的 TF からの回復手法を説明する．

アーキテクチャ・レベルの回復手法

動的 TF 耐性を持たせる対象をプロセッサに限定すると，プロセッサの性質を利用した回復手法が考えられる．これをアーキテクチャ・レベルの回復手法とよぶ．RazorII を利用した回復手法はアーキテクチャ・レベルである．

In-order のプロセッサは通常，レジスタ・ファイル (RF)，プログラム・カウンタ (PC)，データメモリ (データ・キャッシュを含む)，によって定義されるアーキテクチャ・ステートを持つ．並列実行や分岐予測による投機実行を行うプロセッサは，各命令の最後のステージでアーキテクチャ・ステートを更新する．

その際，確実に正しく終了した命令がプログラム順に反映される．例外を起こした命令

© 2010 Information Processing Society of Japan

や，分岐予測ミスによる不要な命令は反映されない．すなわち，命令をプログラム順にどこまで正しく処理したか，その時の RF やデータ・メモリの状態はどうなっているか，といったことがアーキテクチャ・ステートに保持される．原理的には，プロセッサが動作を停止しても，アーキテクチャ・ステートがあれば正しく動作を再開できる．

例外が起こった場合，プロセッサはアーキテクチャ・ステートをもとに逐次的に回復する．動的 TF を例外の一種ととらえれば，この考え方を利用して回復を行うことができる．具体的には以下のようにする．

(1) アーキテクチャ・ステートの更新を行うステージ (ライトバック・ステージ) では，動的 TF が発生しないことを保証する．
(2) プロセッサ上で発生した動的 TF の影響が，アーキテクチャ・ステートに反映されない (影響を含む書き込みが行われない) ことを保証する．
(3) 動的 TF が検出されたらアーキテクチャ・ステートへの書き込み　　を停止し，アーキテクチャ・ステートに保持される状態から再実行することで回復する．

フォールト通知ネットワーク

動的 TF の影響がアーキテクチャ・ステートに反映されないためには，プロセッサ上のあらゆる RazorII FF から動的 TF 検出情報を収集し，ライトバック・ステージに検出を伝える必要がある．そのため，この伝達を担うフォールト通知ネットワークが必要である．

このネットワークに必要な条件は以下の２つである．

(1) すべての動的 TF 検出情報が正しくライトバック・ステージに伝わる．
(2) 動的 TF の影響よりも先に，検出情報がライトバック・ステージに伝わる．

最も単純な構成法は，RazorII FF が検出をしたサイクル内に，その情報がライトバック・ステージに伝える方法である．これはすべての RazorII FF の ERRORbit をすべて or しなければならないため，あきらかにクリティカル・パスになる．そのためプロセッサの性能を著しく落としてしまう．

動的 TF 検出情報は，検出されたサイクル内にライトバック・ステージ伝わらなくてもよく，その影響よりも先に伝えられれば，アーキテクチャ・ステートは保護される．これは，RazorII FF の ERRORbit をパイプライン化することで実現できる．

各段の ERRORbit は他のデータと同様に，１段ずつ先のステージへ伝わっていく．この情報は ERROR の影響と同時にライトバック・ステージに伝わる．

ERRORbit は各段で or されて次の段に進んでいくが，その or 入力数は大した量ではないため，フォールト通知ネットワークはクリティカル・パスにならない．

ライトバック・ステージの直前のステージで動的 TF が発生した場合，それを検出する次のサイクルでは，その影響の書き込みが始まってしまうため，アーキテクチャ・ステートに誤りが反映されてしまう．これを回避するために，ライトバック・ステージの直前に，ただ同じ信号を伝えるだけの，動的 TF が絶対に起きないステージを設ける．これをスタビライジング・ステージと呼ぶ．

プロセッサ構成

動的 TF 耐性を持つ RazorII のプロセッサのパイプラインを図 6 に示す．

このプロセッサは，動的 TF が起きてもよい Speculative 領域と，起きてはいけない Non-speculative 領域とに分かれている．Speculative 領域のパイプライン・レジスタは，動的 TF が発生する可能性があるため，RazorII FF を利用する．

アーキテクチャ・ステート (レジスタ・ファイル，データ・キャッシュ) では動的 TF が起こらないように設計される．

このプロセッサでは，どこかで動的 TF が発生すると，それを RazorII FF で動的に検出して ERROR 信号を出す．その通知をパイプラインでライトバック・ステージに伝搬させる．検出通知がライトバック・ステージに届くと，プロセッサはアーキテクチャ・ステートへの書き込みを停止し，パイプライン・フラッシュによって後続の命令を無効化する．その後，アーキテクチャ・ステートに含まれる PC から再実行することで回復する．

ところで，ライトバック・ステージまでに SEU 等の何らかの原因で bit 反転が起こり，誤った ERROR 検出情報が伝わってもいけない．これは，Speculative 領域では RazorII によって検出でき，Non-speculative 領域では TMR(Triple Module Redundancy) という手法で保護されている．

3. リセットによる回復手法

RazorII の回復手法は，データ・パス上の動的 TF にのみ着目している．単純な In-order のパイプライン・マシンであればこの手法でも良いが，近年の Out-of-order スーパスカラ・プロセッサのように，クリティカルな制御系パスが多いプロセッサであれば，この手法では正しく回復されない．

我々は，データ・パス上のみならず制御系パス上の動的 TF にも対応できる，リセットによる回復手法を提案してきた．本節ではこの手法を紹介する．

なお，対比のために前節で紹介した RazorII の回復手法を，パイプライン・フラッシュによる回復手法，と呼ぶことにする．

© 2010 Information Processing Society of Japan

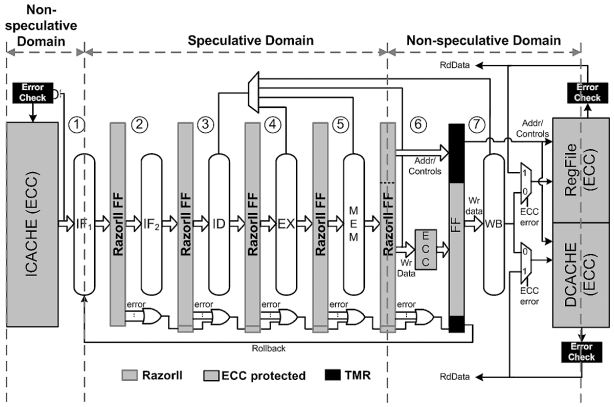

図 6 RazorII processor Pipeline design[4)]

3.1 制御系パス上の動的 TF

制御系パス上で動的 TF が発生すると，データ・パス上の動的 TF の場合に比べて，以下の点で回復が困難である．

(1) 制御系の状態が正常でないため，プロセッサが設計者の想定していない動作をする可能性がある．命令を正しく再実行できるかどうかも不明である．

(2) 再実行できたとしても，その後のプロセッサ制御系の状態が設計者の想定している正常な状態ではない可能性が残る．

たとえば，FIFO バッファ上での動的 TF によって，リード/ライト・ポインタとカウンタの値との間に齟齬が生じた場合，バッファ・アンダランやバッファ・オーバランが発生する．このような状態は，パイプライン・フラッシュをしても正常な状態には戻らず，動的 TF の影響が制御系に残ることになる．

このため，パイプライン・フラッシュによる手法では，制御系パス上の動的 TF に対応できない．

近年の Out-of-order スーパスカラ・プロセッサには，あらゆる制御に上記の FIFO バッファが使われている．さらに，それらの FIFO も含め，制御系パスには遅延の大きなものも多く，ここでの動的 TF は無視できるものではない．

3.2 リセットによる動的 TF からの回復

我々が提案しているリセットによる回復手法は，制御系上の動的 TF からも回復できる．この手法のプロセッサ構成はパイプライン・フラッシュの手法と非常によく似ている．以下の点は同じである．

(1) 動的 TF 検出には，RazorII FF 等の検出機構を付加した FF を利用する．

(2) アーキテクチャ・レベルの回復である．

(3) フォールト通知ネットワークはパイプライン化する．

本手法が対象としているプロセッサは，Out-of-order スーパスカラ・プロセッサであるため，アーキテクチャ・ステートとその更新方法が In-order パイプライン・マシンの場合と少し異なる．

アーキテクチャ・ステートは，論理レジスタ・ファイル (LRF)，プログラム・カウンタ (PC)，データメモリ (データ・キャッシュを含む)，少数の制御レジスタ，から成る，

Out-of-order に実行終了した命令を In-order に並び変えて，アーキテクチャ・ステートを更新することを，コミット，と呼び，それはコミット・ステージで行われる．

この手法では，以下のように回復する．

プロセッサのあらゆる場所で動的 TF を検出し，その検出信号をパイプラインでコミット・ステージに届ける．コミット・ステージに動的 TF 検出が伝えられると，直ちにコミットを停止し，アーキテクチャ・ステートを保護する．

Out-of-order で動作するため，動的 TF の影響自体は必ずしも最短でコミット・ステージに届くとは限らないが，動的 TF の検出信号が必ず最短のステージ数でコミット・ステージに届くように設計することはできる．そのため，アーキテクチャ・ステートに動的 TF の影響は反映されない．

コミットを停止した後，パイプライン・フラッシュではなく，アーキテクチャ・ステートを残してプロセッサをリセットする．

リセットを行った後は，正しいアーキテクチャ・ステート＋その他のレジスタの初期状態，となり，制御系を含めて一貫した状態に回復する．その後，アーキテクチャ・ステートの PC から動作を再開する．

4. リセットによる手法の問題点

リセットによる手法では，コミット・ステージ直前のリオーダ・バッファ(ROB) やロード/ストアキュー (LSQ) での動的 TF を深く考慮しておらず，これによって正しいアーキテクチャ・ステートが保たれない可能性があり，これが問題となっている．

この問題について述べる前に，まず ROB と LSQ について，その機能と仕組みを簡単に

© 2010 Information Processing Society of Japan

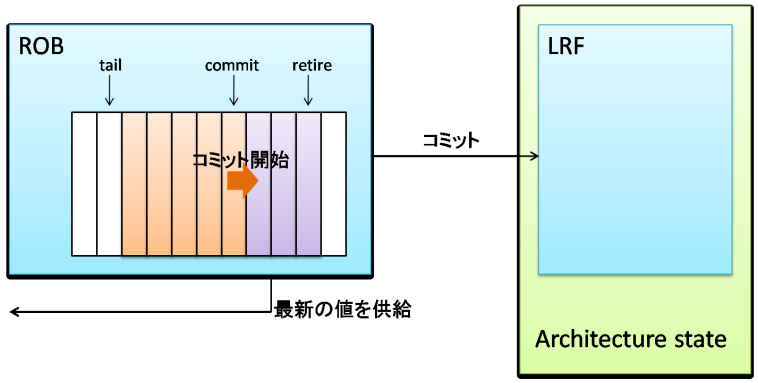

図 7 リオーダ・バッファ

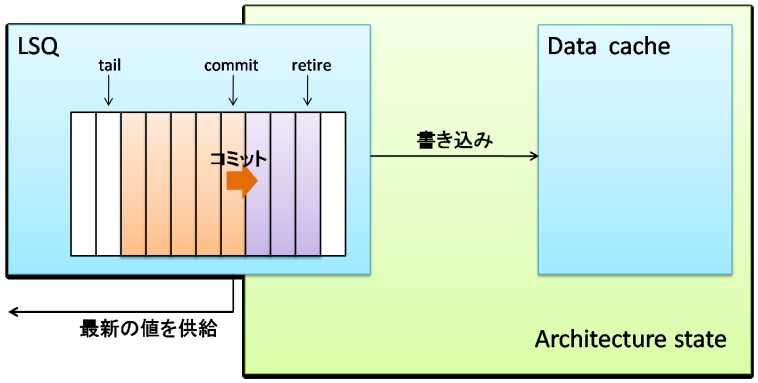

図 8 ロード/ストアキュー

示す.

4.1 リオーダ・バッファ

Out-of-order スーパスカラ・プロセッサには，Out-of-order に実行終了した命令を In-order に並び替えてコミットするモジュールがあり，これがリオーダ・バッファ(ROB) と呼ばれる.(図 7)

すべての命令は，ディスパッチ時に In-order で ROB のエントリが割り当てられる．Out-of-order で実行終了した命令の結果は，終了のタイミングで該当エントリに書き込まれる．ROB では，先頭エントリから順に，命令が終了していればコミット (LRF への書き込み) をしていく．こうすることで In-order でのコミットを実現している．

この方式では，必ずしも最新の実行結果が LRF に存在しない．そのため，ROB は最新の結果を後続の命令に供給する役割を持つ.

ここで，コミットを開始する命令について考える．LRF への書き込みは数サイクルかかるため，前述した実行結果供給のために，LRF への書き込みが終わるまで ROB がこの命令のデータを保持する必要がある．しかし，書き込みが完了してから先頭ポインタをカウント・アップしていては，コミットのスループットが小さくなってしまう.

この問題を回避するために，ROB は 3 種類のポインタを持つ．ここでは，先頭から，retire ポインタ，commit ポインタ，tail ポインタ，と呼ぶ.

エントリ割り当て時には tail ポインタの指すエントリを割り当てる．次にコミットを開始する命令は commit ポインタの指すエントリである．コミットを開始したら即座に commit ポインタをカウント・アップする．LRF に実行結果が書き込まれたら retire ポインタをカウント・アップする.

こうすることで，LRF への書き込み完了を待つことなく，次々と命令のコミットを開始することができ，コミット開始したがまだ LRF にないという実行結果も ROB から供給することができる.

4.2 ロード/ストアキュー

Out-of-order スーパスカラ・プロセッサにおいて，メモリ・アクセス命令を Out-of-order に実行するためのモジュールが，ロード/ストアキュー (LSQ) である (図 8)．メモリ・アクセス命令のプログラム順を管理するため，FIFO バッファで構成され，各エントリはメモリ・アドレスとストア・データを持つ.

LSQ も ROB と同様にポインタを 3 つ持つ．先頭から順に，retire ポインタ，commit ポインタ，tail ポインタ，と呼ぶことにする.

エントリ割り当ては ROB に該当命令が割り当てられるときに行われ，tail ポインタの指すエントリが割り当てられる．ROB でロード/ストア命令のコミットが開始されると同時に，commit ポインタをカウント・アップする．ストア命令の結果がメモリ (データ・キャッシュ) に書き込まれたら，retire ポインタをカウント・アップする．retire ポインタのカウント・アップではロード命令をとばすようにする.

ストア命令に関しては，アーキテクチャ・ステートを In-order に更新する必要性から，キャッシュへの書き込み要求の発行はコミット開始後に行う．ロード命令に関しては，先行

© 2010 Information Processing Society of Japan

するすべてのストア命令と同一アドレスでなければ，いつ読み出し要求をしても良い．

こうすることで，ROB と同様に，アーキテクチャ・ステートを保ちつつも，データ・キャッシュへの書き込み完了を待つことなく，次のコミットを開始することができる．

ここで，commit ポインタと retire ポインタとの間のデータは，ROB の場合と同様に，後続のロード命令へ供給されるためにも存在するが，もう 1 つ他に大きな役割がある．

通常，ロード命令はそれより後の命令に依存しているため，ストア命令よりも優先される．ストア命令は，実行できるロード命令が無く，キャッシュのポートが空いているときに，実際に書き込みを行う (サイクル・スチール)．そのため，コミットを開始したストア命令は，ただちにキャッシュへのアクセスを始めるわけではなく，キャッシュのポートが空くのを LSQ で待つ．

retire ポインタと commit ポインタとの間のデータは，キャッシュへの書き込みが始まるまでは LSQ 上にしか存在しない (これに対し，ROB の場合は，コミットが始まった命令が直ちに先のパイプラインにも進むため，ROB 上だけに存在するわけではない)．もし例外や動的 TF が起きたら，実行再開の前に，ここのデータを確実にキャッシュへ書き込まなければならない．すなわち，これらのデータはアーキテクチャ・ステートに含まれることになる．この意味で，commit ポインタのカウント・アップは，単なるコミットの開始点ではなく，実際にアーキテクチャ・ステートを更新する役割を担う．

4.3 LSQ 上の動的 TF

前節で述べたリセットによる手法は，LSQ 上での動的 TF を考慮していない．

たとえば，図 9 に示すように，commit ポインタが動的 TF でずれてしまった場合を考える．このとき，まだコミットされていない ins1,ins2 が commit ポインタと retire ポインタとの間に位置することになる．すなわち ins1,ins2 はともにコミットされているとみなされ，アーキテクチャ・ステートに含まれる．これにより，正しいアーキテクチャ・ステートが保持されていないことになり，問題である．

5. 提案コミット方式

LSQ 上での動的 TF によるアーキテクチャ・ステートの誤った更新を防ぐために，リタイア・バッファを提案する (図 10)．リタイア・バッファは FIFO のバッファであり，従来のコミット方式おいて LSQ に存在するアーキテクチャ・ステートを，LSQ から分離したものである．ただし，LSQ のポインタ構成は特に変更しない．これは，リタイア・バッファに後続の命令への値供給をさせないためである．これについては後述する．

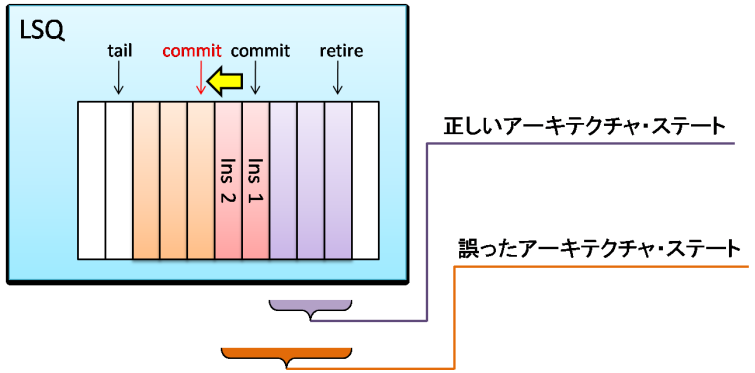

図 9 ロード/ストアキュー上の動的 TF

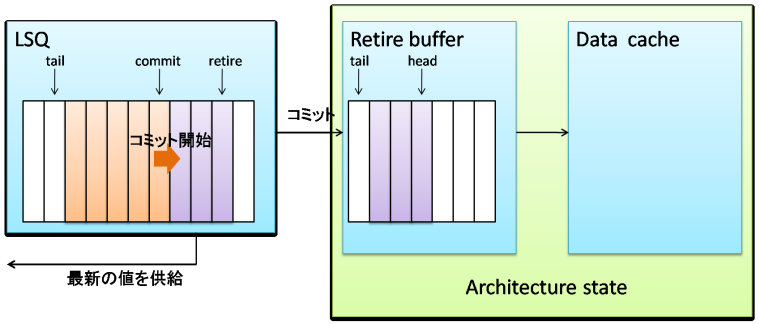

図 10 リタイア・バッファ

また，従来の方式ではコミットの開始を commit ポインタで行っていたため，スタビライジング・ステージが考慮されていなかったが，正しいアーキテクチャ・ステートを保持するためには必要な概念であるため，ここで再度しっかり位置付けることにする．提案するコミット方式は図 11 のようになる．

このコミット方式の動作を示す．すべての命令は ROB によって In-order にコミットを開始し，次のスタビライジング・ステージに進む．これがストア命令であった場合，該当の命令が LSQ から同時に次のステージに進む．スタビライジング・ステージを抜けた命令は，LRF とリタイア・バッファに書き込まれる．LRF の書き込みが完了した命令は，ROB のエントリが削除され，リタイアする．リタイア・バッファに書き込まれたストア命令は，サ

© 2010 Information Processing Society of Japan

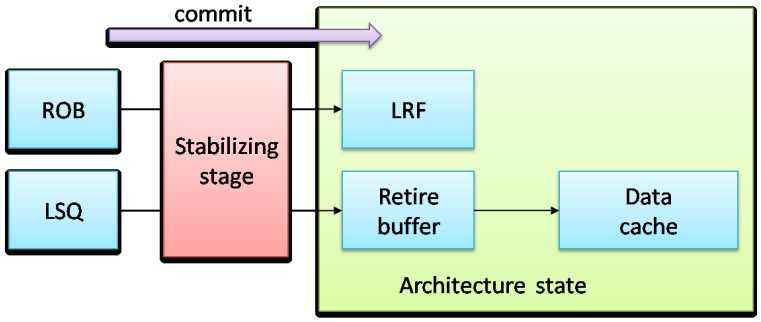

図 11 提案コミット方式

イクル・スチールによってデータ・キャッシュへの書き込まれる．書き込みが完了した命令は，LSQ のエントリが削除され，リタイアする．

LRF の状態とリタイア・バッファ，データ・キャッシュの状態とは，ROB によってプログラム順が保証され，スタビライジング・ステージによって動的 TF の影響がないことが保証されるので，LRF とデータメモリに関する正しいアーキテクチャ・ステートが実現できる．

前節でみた LSQ 上での動的 TF が起こっても，ここでのアーキテクチャ・ステートは正しく保たれ，この問題は解消される．

5.1 スタビライジング・ステージ

コミットが開始されて最初に行われるのは，LRF をアクセスするためのアドレス・デコードである．このステージをスタビライジング・ステージとみなすことを提案する (図 12)．

ここでは，フォールト通知ネットワーク (図の TF) の出力値を利用して，ワード線の最終的なアサートを制御している．動的 TF の通知があれば，どのワード線もアサートされない．すなわち，次段で誤った書き込みが行われない．

スタビライジング・ステージとしての役割を果たすためには，このステージで動的 TF が起こらないことが必要である．アドレス・デコードで動的 TF が起こる可能性があるかどうかは，まだはっきりと結論が出ていないため，今後しっかりとした議論をする必要がある．

このように，既存のステージをスタビライジング・ステージとみなしたいと考えることには，理由がある．スタビライジング・ステージとして実質何もしないステージを新たに設けると，パイプライン・ステージが 1 段分延びることになり，各モジュールの命令のリタイアが少し遅くなる．リタイアが遅くなると各モジュールの資源の解放が遅れ，後続の命令が利

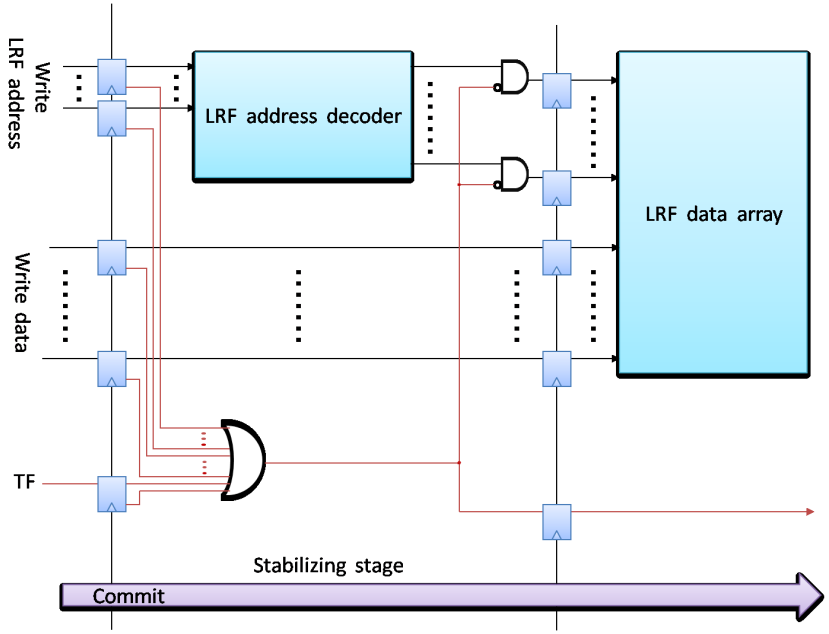

図 12 スタビライジング・ステージ

用できる資源が少なくなることで，性能 (IPC 等) が下がる．

さて，アドレス・デコードをスタビライジング・ステージにできると仮定した場合，LSQ 側のパイプラインとの同期を考えると，図 13 のようになる．

リタイア・バッファの書き込み先は tail ポインタによってあらかじめデコードされているので，LSQ を出たストア・データはただちにリタイア・バッファに書き込むことが可能である．しかし，ROB からの書き込みデータがスタビライジング・ステージを過ぎるところでプログラム順の同期をとるために，すぐにはリタイア・バッファに書き込まず，一段分何もしないステージをとることにする．

6. 議　　論

6.1 リタイア・バッファの構成について

前節で提案したリタイア・バッファは，その内容が LSQ の一部 (retire ポインタと commit

© 2010 Information Processing Society of Japan

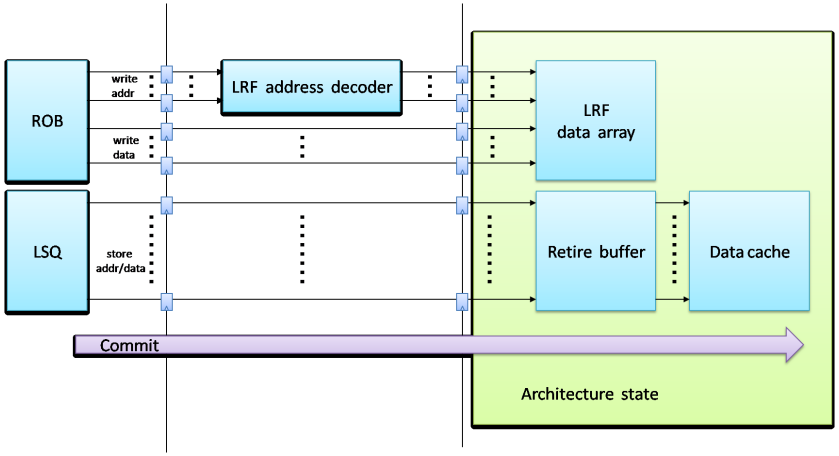

図 13 スタビライジング・ステージ

ポインタとの間) と同じである．そのため，LSQ の一部を排除し，LSQ のサイズをより小さくする，という構成も考えられる (図 14)．

この構成ではリタイア・バッファにのみ存在するデータがあるため，リタイア・バッファから後続命令に対して最新の値を供給しなければならない．その場合，リタイア・バッファは CAM(連想記憶メモリ) になる．提案の手法であれば CAM ではなく RAM になる．CAM は RAM よりも面積が大きい．さらに，LSQ とリタイア・バッファで同じアドレスの値が存在した場合に備えて，優先度をつけて新しい LSQ 側のデータを選択しなければならず，処理が複雑になる．

6.2 リタイア・バッファとアーキテクチャ・ステート

提案手法では，リタイア・バッファに移ったストア命令を，アーキテクチャ・ステートの一部とみなしたが，そうではなく，LSQ の一部をそのままアーキテクチャ・ステートとみなし，そこが動的 TF で狂ったときのためのバックアップとしてリタイア・バッファを位置付ける方法も考えられる (図 15)．

この方法では，LSQ のポートがリタイア・バッファ用に一つ増加するため，好ましくない．さらに，LSQ のアーキテクチャ・ステートが一瞬でも正しくなくなることを可能性として想定しており，最初からアーキテクチャ・ステートを正しく保護する手法に比べて，アーキテクチャ・ステートを回復する手間がある分，無駄が多い．

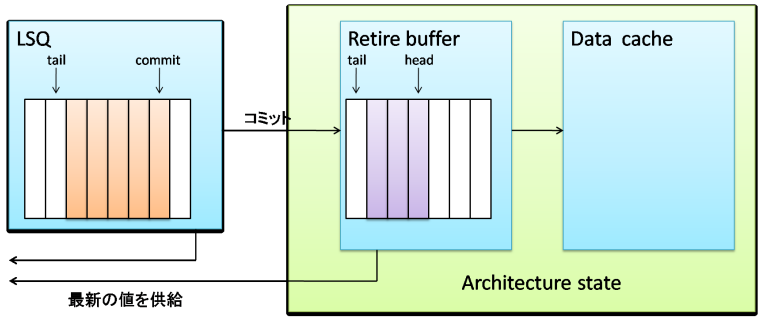

図 14 リタイア・バッファ(CAM)

6.3 コミットメント・パイプライン

一般に，コミットする，と一言で表されることが多いが，コミットする命令が ROB で決まるところから，実際に LRF やデータ・キャッシュへ書き込みが終わって命令がリタイアするまで，長い時間がかかる．

これは一種の独立したパイプラインとみなすことができる (図 16)．ROB でコミットする命令を選択するステージは，Out-of-order な命令から終了している命令を wakeup し，In-order に select するスケジューリングのステージ，スタビライジング・ステージは，アーキテクチャ・ステートのモジュール間での同期を確定し，再び Out-of-order への道を開くステージ，LRF とリタイア・バッファへの書き込みのステージは，実行ステージ，といった具合である．

このパイプラインはまだ深く考慮されていないが，ROB での命令の選択や，各モジュールでのリタイア (資源の解放) のタイミングを考えるための一助になる可能性を秘めている．たとえば，資源の解放のタイミングが，コミットメント・パイプラインで事前にしっかりと分かれば，フロントエンドでの命令フェッチをより無駄なく行う手法を考えやすくなる．

7. ま と め

リセットによる回復手法は制御系パス上の動的 TF からも回復することができる．本稿では，この手法において考慮が不十分であったコミット周辺を問題視し，耐過渡故障方式としての信頼度を上げるため，コミット方式を提案した．

本方式では，タイミング・フォールトが起こりうるモジュールで管理しているアーキテクチャ・ステートを，別のモジュールに移し，その保護をより確かなものにする．また，スタ

© 2010 Information Processing Society of Japan

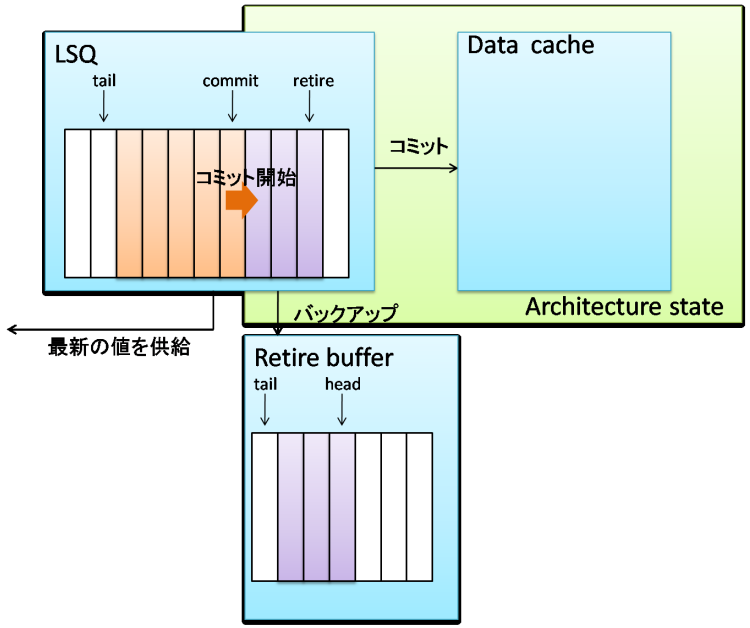

図 15 リタイア・バッファ(バックアップ)

ビライジング・ステージを既存のパイプライン・ステージの内部に組み込むことで，プロセッサに耐故障性をもたせる上でのオーバヘッドを軽減できる可能性を示した．

謝辞 本論文の研究は，一部 JST CREST による．

参 考 文 献

1) 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. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.24, No.12 (2005).
2) S.Das, D.Roberts, S.Lee, S.Pant, D.Blaauw, T.Austin, K.Flautner, T.Mudge: A Self-Tuning DVS Processor using Delay-Error Detection and Correction, *IEEE Journal of Solid-State Circuits (JSSC)* (2006).
3) D.Ernst, N.Kim, S.Das, S.Pant, T.Pham, R.Rao, C.Ziesler, D.Blaauw, T.Austin and T.Mudge: Razor: A Low-Power Pipeline Based on Circuit-Level Timing Specification, *Int'l Symp. on Microarchitecture (MICRO)*, pp.7–18 (2003).
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 Symp. on Solid-State Circuits Conference (ISSCC)* (2008).
5) 佐藤寿倫：カナリア・フリップフロップを利用する省電力マイクロプロセッサの評価，先進的計算基盤シンポジウム SACSIS，pp.227–234 (2007).
6) 杉本　健：タイミング・フォルト耐性を持つスーパスカラ・プロセッサ，東京大学修士論文 (2009).
7) 有馬　慧：Out-of-order スーパスカラ・プロセッサの耐過渡故障方式の改良，東京大学卒業論文 (2010).

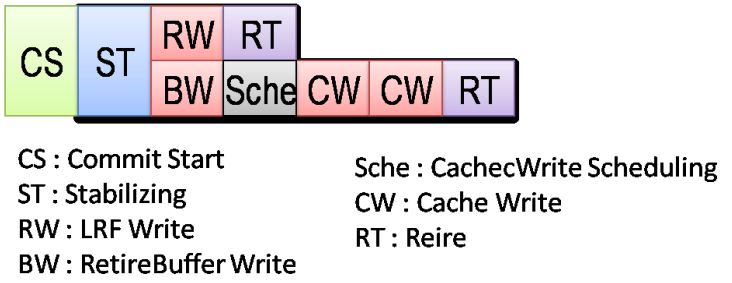

図 16 コミットメント・パイプライン

© 2010 Information Processing Society of Japan