修士論文

タイミング・フォールト耐性を持つ Out-of-Order プロセッサ

Timing-Fault-Tolerant Out-of-Order Processor

平成24年2月8日提出

指導教員
五島正裕 准教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-106401 有馬慧

概 要

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．今後の半導体産業の発展には，ばらつきを吸収する回路技術が不可欠である．プロセッサを対象とするばらつき対策の１つに，タイミング・フォールト（TF）を動的に検出/回復するプロセッサを用いる手法が提案されているが，既存の手法では近年の複雑な Out-of-Order スーパスカラ・プロセッサに対応できない．本研究では，Out-of-Order スーパスカラ・プロセッサのコミット・モジュール周りを詳細に検討し，既存の手法では回復できなかったロード/ストア・キュー (LSQ) の制御系の TF からも回復できるコミット方式を提案する．これは，通常は LSQ 内にあるストア・バッファを LSQ から分離し，In-Order なコミットを TF から完全に保護することで可能となる．ストア・バッファを LSQ から分離することで，コミットのレイテンシが増加するが，これによる性能低下は平均で 0.7%，最低でも 6.0%と十分に低いことを予備評価で確認した．

目 次

図 目 次

表 目 次

第1章　はじめに

近年の LSI を牽引してきたのは，半導体製造プロセスの微細化である．微細化によって，性能向上，低消費電力，集積度向上を達成してきた．しかし，微細化に伴って半導体素子の特性がばらつくという新たな問題が発生している．[1, 2]

ばらつきは微細化が進むにつれて増大している．この様子を**図 1.1** に示す．微細化の効果で平均の性能が向上する一方，ばらつき増大の影響でワースト値の向上は見込めなくなりつつある．

従来の LSI 設計は，ワースト値に基づくワースト・ケース設計であるため，このまま微細化を続けても今後の性能向上が期待できない．今後も微細化の効果を得るためには，ばらつきへの対策が必須である．

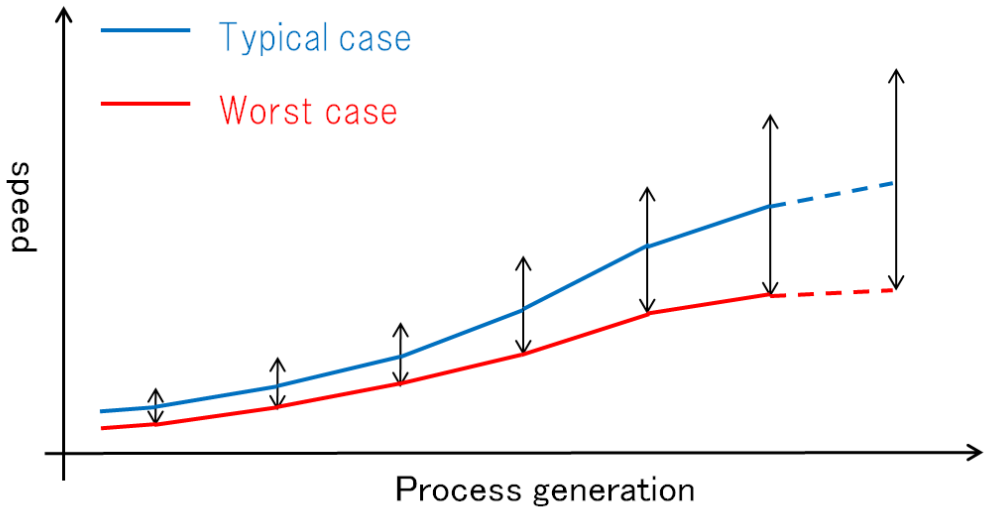

図 1.1: プロセスの微細化とばらつきの増大

ばらつきへの対策は様々な研究がなされている．[3, 4] プロセッサを対象とした対策の 1 つに，**タイミング・フォールト (TF:Timing Fault) の検出/回復**を利用する手法がある．

TF 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．ワースト・

ケース設計では，回路遅延の変動をワーストで見積もっても TF が発生しないように十分にタイミング・マージンをとる．そのため，ばらつきが増大したプロセスでは悲観的になりすぎる．

TF を動的に検出/回復することができれば，ばらつきによる悲観的なマージンの見積もりを回避し，実際の回路遅延に合わせた動作が可能となる．DVFS[5] と協調することで，個体差や動作環境に応じた実際的なタイミング・マージンで動作させるため，ワースト・ケース設計に比べて性能を向上させることができる．

このようなアプローチをとっている代表的な既存手法に，RazorII[6] がある．RazorII では，パイプラインの各ステージで生じた TF を一種の例外のように扱う．TF を起こした命令を検出し，それより上流のパイプラインをフラッシュすることによって TF から回復する．

しかし，RazorII の手法は，近年の複雑な Out-of-Order スーパスカラ・プロセッサや，制御系で生じる TF についての考慮が不十分なため，論文に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．

我々は， RazorII が考慮していない複雑な Out-of-Order プロセッサを対象とする手法を提案する．

基本的な考え方は RazorII の手法と同じであるが，フラッシュではなく初期化を用いることで制御系の TF から回復することができる．また，従来の基本的な Out-of-Order スーパスカラ・プロセッサの構成では，回復の起点となるプロセッサのコンテクストを確実に保護することが困難であるが，我々の手法ではそれが可能となる．

本稿の構成は以下の通りである．まず，ばらつき対策としての TF 検出/回復の利用について，既存手法の RazorII をからめて 2 章で述べる．次の 3 章では，近年の Out-of-Order スーパスカラ・プロセッサの実際について述べ，実際のプロセッサに TF 耐性を持たせるには，RazorII では不十分であることを述べる．続く 4 章では，複雑な Out-of-Order スーパスカラ・プロセッサにも適用できる TF からの回復手法を提案し，5 章で提案手法に関する予備評価について述べる．

第2章　既存手法：RazorII

1 章で簡単にふれたように，ばらつきへの対策として，TF の検出/回復技術を利用する手法がある．

本章では，プロセッサを対象とした場合の TF 検出/回復技術について，既存手法の Razor[7, 6] を例に説明し，最後に TF 検出/回復を利用したばらつき対策について述べる．

2.1　タイミング・フォールト

タイミング・フォールト （TF: Timing Fault）とは，回路遅延の動的な変動によって設計者の意図とは異なる動作をする過渡故障である．例を**図 2.1**（左）に示す．

この例では，bit1 の回路遅延が cycle1 で動的に増加している様子を表す．bit1 は cycle1 の時間内に正しい値に遷移せず，cycle2 へ入る CLOCK の立ち上がりで誤った値（正しい値へ遷移する前の値）がサンプリングされている．

従来のワースト・ケース設計では，回路遅延の動的な変動がワーストであっても TF が起こらないように，十分なタイミング・マージンを確保している．そのため，熱暴走などを起こさない限り，通常は TF は発生しない．

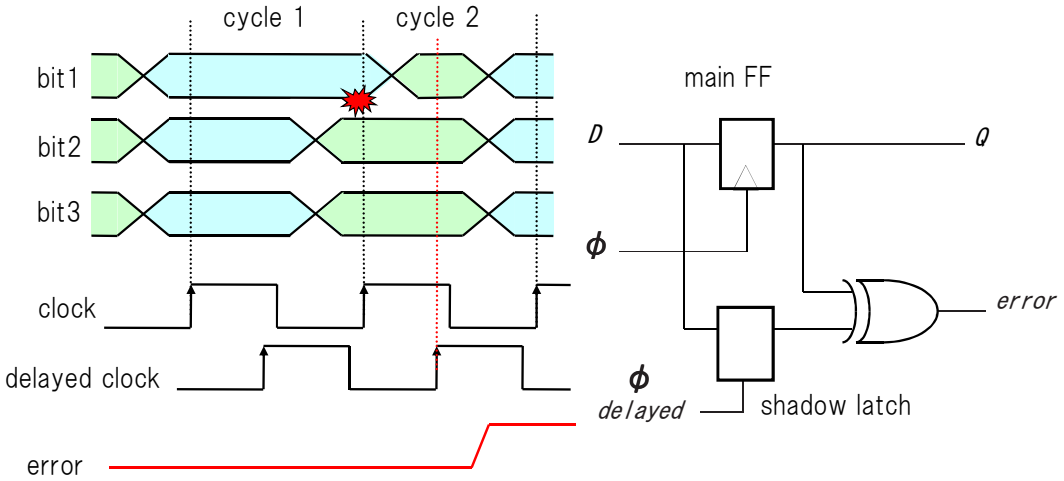

図 2.1: (左)TF，(右) RazorI FF

2.2 プロセッサにおける TF 検出/回復の概要

ここで，プロセッサにおける TF 検出/回復の概要を述べる．

回復の鍵となるのは，プロセッサの持つ**プロセッサ・ステート**と呼ばれるコンテキストである．

プロセッサ・ステートは，プログラム・カウンタ，論理レジスタ・ファイル（LRF: Logical Register File），その他少数の制御レジスタからなる．[1]

プロセッサ・ステートは，プログラムを逐次実行した際に観測される論理的なステートである．そのため，L1 データ・キャッシュ（L1D$: L1 Data Cache）からなる**メモリ・ステート**は，プロセッサ・ステートに同期して更新される．

本稿では，同期して更新されるプロセッサ・ステートと メモリ・ステートをあわせて，以後 **P/M ステート**(P/M State: Processor/Memory State) と呼ぶことにする．

P/M ステートは，プログラム・オーダで不可逆的に更新される．この不可逆的な更新は**コミット**と呼ばれる．[2]

原理的には P/M ステートの更新は逐次的であり，その過程はすべて一意に定まる．これは，どの時点の P/M ステートからプログラムを実行しても，そこからの実行および実行結果がすべて同じであることを意味する．

この性質を利用して，プロセッサにおける TF 検出/回復は次のように行われる．

1. P/M ステートを TF の影響のあるデータでコミットしない．
2. TF が検出されたら，プロセッサから TF の影響を取り除き，保護された正しい P/M ステートから再実行する．

TF の影響を除去した正しく動作するプロセッサで，TF の影響のない正しい P/M ステートから再実行するため，実行結果が正しいことが保証され，これは TF からの回復を意味する．

2.3 RazorII の TF 検出/回復

ここでは，既存手法である RazorII の TF 検出/回復について述べる．

[1]単純なスカラ・プロセッサでは LRF という呼び方はしないが，逐次的なプロセッサ・ステートを表すという意味で，レジスタ・ファイルが LRF に相当する．

[2]単純なスカラ・プロセッサではコミットという呼び方はしないが，不可逆的に P/M ステートを更新するという意味で，ライト・バックがコミットに相当する

2.3.1 TF 検出

Razor[7] では，TF を検出する RazorI FF（図 2.1 右）が提案されている．RazorI FF は，メインのフリップ・フロップに対して，少し遅れた位相で動作する シャドウ・ラッチ が付加されている．

この例では，bit1 で TF が発生しており，メインの FF には誤った値がサンプリングされていることは，2.1 節で述べた．

一方で，シャドウ・ラッチでは，メインのフリップ・フロップより少し遅れて正しい値をサンプリングしている．メインとシャドウの出力を比較することで，TF を検出することができる．

RazorII[6] では，原理は異なるが同様のタイミングで TF を検出する RazorII FF が提案されており，これは RazorI FF に比べて面積等の点でより優れている．

2.3.2 TF からの回復

RazorII における TF からの回復の流れは以下のようになる．全体像を**図 2.2** に示す．

1. プロセッサ上のどこかで TF が発生した場合，RazorI(II) FF でそれを検出する．
2. 検出されると，TF 通知ネットワーク（後述）を通して情報がコミット・ステージへ伝わり，コミットを停止する．
3. P/M ステートを残してパイプラインをフラッシュする．
4. 保護された P/M ステートから動作を再開することで回復する．

2.3.3 パイプライン構成

RazorII のパイプライン構成を述べる．パイプライン全体は，TF の発生を認める Speculative ドメインと，認めない Non-speculative ドメインとに大きく分類される．

(パイプライン・レジスタ) パイプライン・レジスタは RazorI(II) FF で構成される．Speculative ドメインである各ステージ上で TF が発生した場合，TF の検出を次のパイプライン・ステージへ事後的に出力する．

(TF 通知ネットワーク) パイプライン・レジスタから出力される TF 検出信号の論理和を各ステージでとり，次のステージへ出力する．次のステージでは，この出力を論理和の入力に含める．（図 2.2 下側）

(スタビライズ・ステージ)　TF は事後的に次のステージで検出される．そのため，コミット直前のステージで TF が発生した場合，Non-speculative ドメインである P/M ステートを，TF の影響を含むデータで更新してしまう．これを防ぐために，コミット直前に TF が発生しないような（実質的に何もしない）ステージが必要で，これをスタビライズ・ステージと呼ぶ.（図 2.2 右側）

プロセッサ上の TF 検出情報をまとめて，コミットする/しないを決定するステージとも言える．

(回復機構)　TF を起こした命令と，TF 通知ネットワークを通したその検出情報は，同時にコミット・ステージへ到達する．RazorII では，TF を一種の例外とみなすことで，投機ミスからの回復と同じ意味でのパイプライン・フラッシュを用いて TF から回復することができる.（図 2.2 上側）

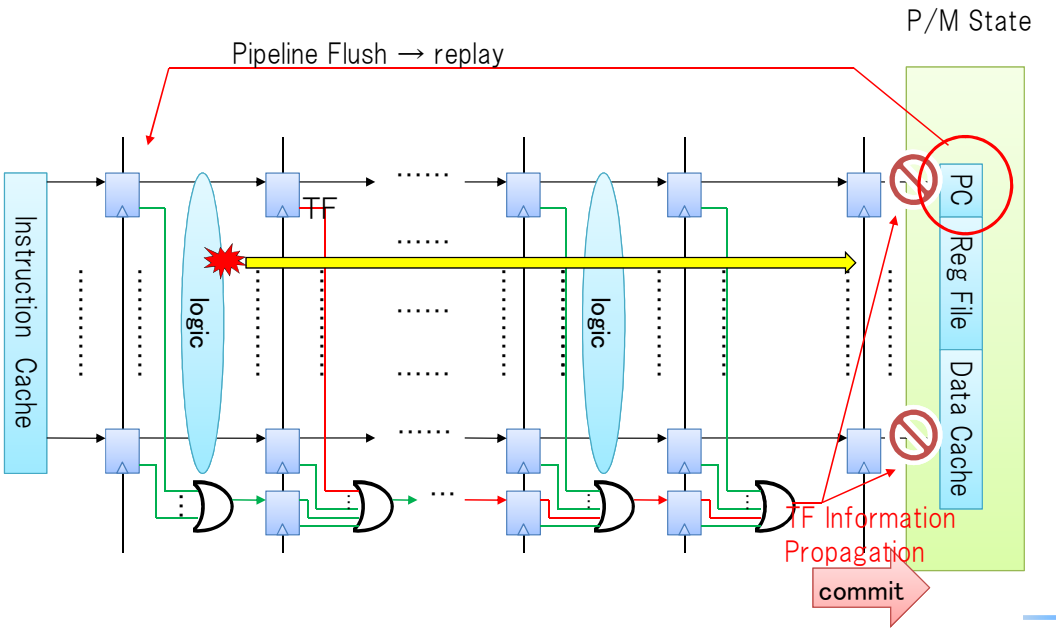

図 2.2: RazorII の TF 検出/回復

2.4　TF 検出/回復と DVFS との協調

TF 検出/回復技術と DVFS(Dynamic Voltage and Frequency Scaling) とが協調することで，ばらつきの影響をおさえることができる．本節では，その具体的な方法を述べる.（**図 2.3**）

DVFS とは，動作時に電源電圧や動作周波数を動的に変化させることのできる技術である．

動作時の電源電圧と動作周波数の組を動作点と呼ぶことにすると，従来のDVFS では，TF が絶対に起きないように十分にマージンを確保した動作点（図の×印）で動作する．

この動作点から徐々に高周波数や低電圧によせていくことで，いつかは TF が低確率で発生する動作点に到達する．

TF を検出したら回復を行い，しばらく TF が起こらなければ，また徐々に高周波数や低電圧によせていく．頻繁に TF が起こるようであれば，逆に低周波数や高電圧に少しだけ戻す．

TF の発生確率の閾値を設定することで，TF が低確率で発生するギリギリの動作点を実現することができる．

TF が発生しない間は，正しく動作する（ワーストではない）実際的な動作点で動作が可能なため，ワーストに合わせた悲観的なマージンを確保する場合と比べて，性能が向上する．

ここで考慮しなければならないことに，回復にかかるオーバヘッドがある．しかし，TF の発生確率が十分に低ければ，大きな影響はないといえる．

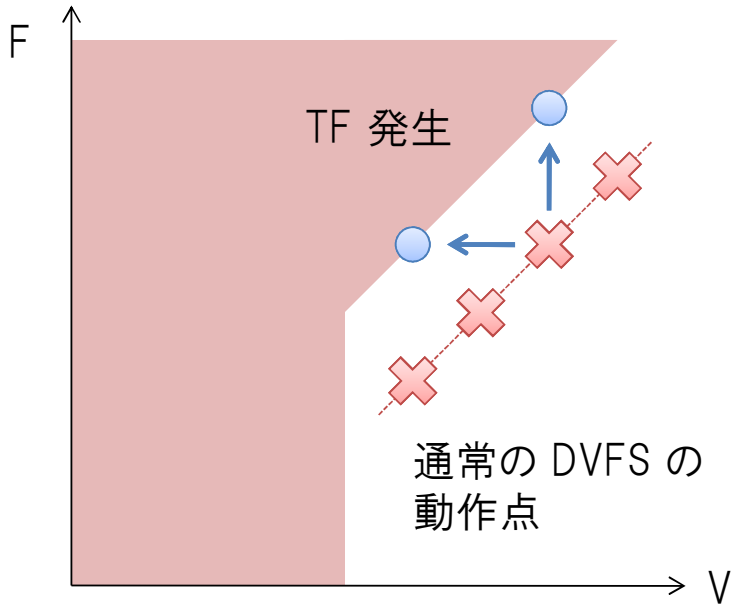

図 2.3: TF 検出/回復 と DVFS

第3章　プロセッサの実際と RazorII の限界

前章で述べた RazorII の手法は，ごく単純なスカラ・プロセッサを想定する限りでは正しく動作するが，実用的なプロセッサで起こりうる制御系の TF からは正しく回復できない．

本章では，我々が対象としている複雑な Out-of-Order スーパスカラ・プロセッサの実際について述べ，RazorII の手法では限界があることを述べる．

3.1　Out-of-Order スーパスカラ・プロセッサのコミット

本節では，Out-of-Order プロセッサのコミットについて説明する．

3.1.1　コミットを行うモジュール

Out-of-Order スーパスカラ・プロセッサには，コミットを行うモジュールが 2 つ存在する．1 つは**リオーダ・バッファ(ROB: ReOrder Buffer)**，もう 1 つは**ロード/ストア・キュー (LSQ: Load/Store Queue)** である．

ROB は，インフライトなすべての命令のエントリを保持しており，プログラム・オーダに PC や LRF への書き込みを行う．LSQ は，インフライトなすべてのメモリ命令のエントリを保持しており，プログラム・オーダに L1D$への書き込み（ストア）を行う．

P/M ステートをプログラム・オーダで更新するには，LRF への書き込みと L1D$ への書き込みとが同期している必要がある．ROB がメモリ命令を含めた全命令を管理していることを利用し，ROB と LSQ がこの同期を行っている．

3.1.2　ROB/LSQ

ROB，LSQ の主な役割は以下の 2 つである．

- 同期したコミットを行う．

- 後続の命令に最新の値を供給する．

コミットはプログラム順を意識した動作であるため，ROB/LSQ は FIFO のバッファで構成され，プログラム順にエントリを割り当てる．

また，Out-of-Order に終了した命令の結果が実際に LRF や L1D$へ反映されるまでには，時間がかかる．その間，最新の値は LRF や L1D$には存在しない．最新の値を必要とする後続の命令には，ROB や LSQ から値を供給する．[1]

ROB

ROB は３つのポインタを持つ FIFO として構成される．（**図 3.1**）

なお，ここでのポインタの名前は一般的なものではなく，本稿で便宜的に名づけたものである．

- tail ポインタ
  In-Order にディスパッチされるすべての命令に対して，In-Order にエントリを割り当てる．次にディスパッチされる新しい命令に割り当てるエントリを tail ポインタで管理する．

- commit ポインタ
  Out-of-Order に終了した命令を In-Order にコミットする．次にコミットする命令を commit ポインタで管理する．

- head ポインタ
  LRF への書き込みには数サイクル要する．LRF へ書き込みが完了するまでは，コミットした命令のエントリは削除できない．次に書き込みが完了して削除する予定のエントリを head ポインタで管理する．

commit ポインタの指すエントリが，次にコミットを開始する命令のエントリである．commit〜head は，コミットを開始しているが LRF への書き込みが終わっていない命令のエントリである．

ここで，コミットの開始とは commit ポインタを更新すること，および，該当命令を LRF へ送りだすことを意味している．

LSQ

LSQ は，ROB と同じ 3 つのポインタに加え，さらに store ポインタを持ち，４つのポインタを持つ FIFO となっている．（**図 3.2**）．

[1]ただし，フューチャ・ファイルを実装するデザインの場合，ROB は必ずしも後続の命令へ値を供給するわけではない．

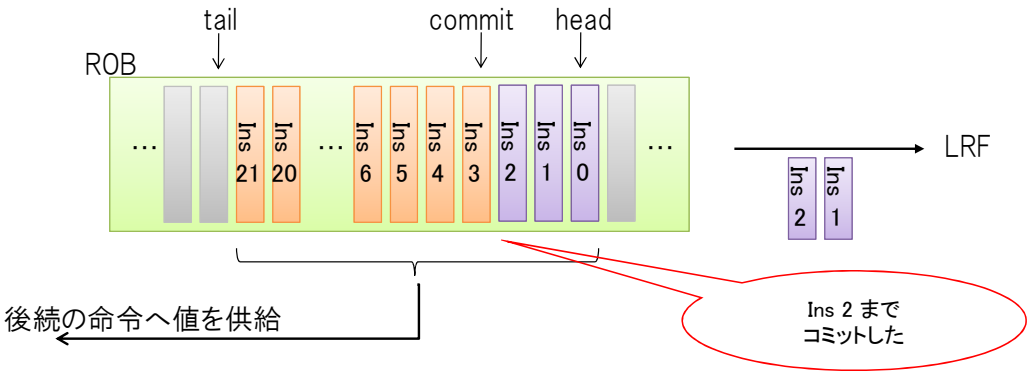

図 3.1: ROB

- tail ポインタ
  ディスパッチされる命令のうち，すべてのメモリ命令に対して，In-Order にエントリを割り当てる．次にディスパッチされる新しいメモリ命令に割り当てるエントリを tail ポインタで管理する．

- commit ポインタ
  Out-of-Order に終了した命令を In-Order にコミットする．次にコミットする命令を commit ポインタで管理する．commit ポインタの更新は，ROB の commit ポインタと同期して行う．

- store ポインタ
  コミットしたストア命令は，L1D$のポートが空いているとき（ロード命令がないとき）にL1D$へ送出される（サイクル・スチール）．コミットした命令の中で次に送出するストア命令を store ポインタで管理している．

- head ポインタ
  L1D$への書き込みには数サイクル要する．L1D$へ書き込みが完了するまでは，コミットした命令のエントリは削除できない．次に書き込みが完了して削除する予定のエントリを head ポインタで管理する．

LSQ のコミットは ROB に同期して開始し，commit ポインタを更新する．しかし，コミットを開始した命令がただちにL1D$への書き込みを開始するわけではない．この点で ROB と異なる．L1D$のポートはロード命令が優先して使用するため，ストア命令の書き込みはL1D$のポートが空いているときに行うからである（サイクル・スチール）．

つまり，head〜commit がいわゆるストア・バッファの役割を果たしており，どこまで書き込みを開始したか（どこまでL1D$ へ送りだしたか）を，store ポインタで管理している．

Out-of-Order なデータ・アレイの更新

コミットを開始したストア命令がいつL1D$へ送られるかは，サイクル・スチールのためわからない．これは，ROB と LSQ とがコミット開始（commit ポインタの更新）で同期したにも関わらず，実際にはLRF/L1D$のデータ・アレイの更新の足並みがずれてしまうことを意味する．

さらに，ROB から送り出された命令によって LRF の書き込みが完了までのサイクル数と，LSQ から送り出されたストア命令によってL1D$の書き込みが完了までのサイクル数とは，一般には異なるため，ここでも足並みがずれてしまう．

つまり，P/M ステートの更新は Out-of-Order になっている．

P/M ステートを論理的なプログラム・オーダで保持するためには，コミットしてストア・バッファに所属した命令を，いつかは必ずL1D$に反映させなければならない．

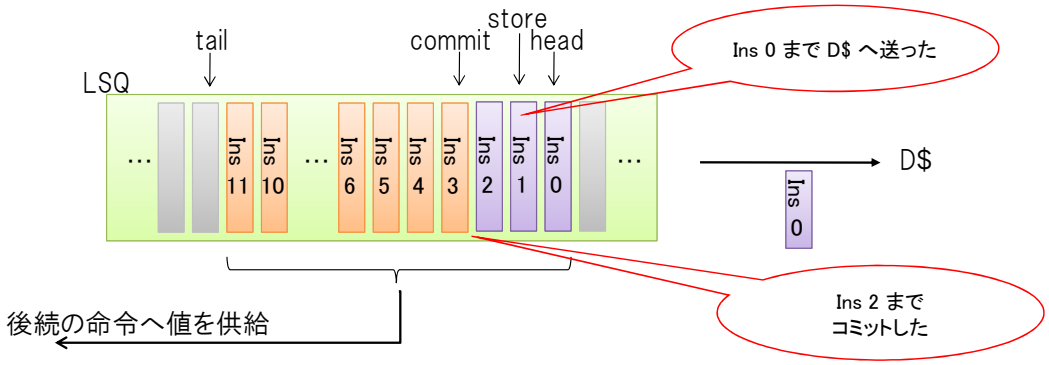

図 3.2: LSQ

3.2　コミットメント・パイプライン

近年の Out-of-Order スーパスカラ・プロセッサのコミットは，commit ポインタの更新から，LRF/L1D$のデータ・アレイの更新まで，幅がある．

そこで，コミットメント・パイプラインという考え方を導入する．

3.2.1　広義/狭義のコミット

まず，**広義のコミット**と**狭義のコミット**を定義する．

広義のコミットは，P/M ステートの更新に関わる一連の手続きを表す．ROB/LSQ の例では，commit ポインタの更新から LRF/L1D$の更新までにかかる数サイクルを指す．

狭義のコミットは，P/M ステートの更新の決心である．イメージとしては，コミットのコミットである．

狭義のコミットをした命令は，確実に P/M ステートに反映する．逆に，狭義のコミットをしていない命令は，コミットを中止することができる．幅のある広義のコミットとは異なり，狭義のコミットは瞬間的なものである．

この観点で言えば，１サイクルでコミットができた昔は，広義のコミット＝狭義のコミット，であった．また，3.1.2 の LSQ で考えると，広義のコミットの開始＝狭義のコミット，ということになる．

3.2.2　コミットメント・パイプライン

コミットメント・パイプラインとは，コミットの開始→狭義のコミット（コミットのコミット）→コミットの終了，の一連のパイプラインのことである．

コミットのコミットは，P/M ステートの不可逆的な更新を行う決心のことであり，これを過ぎた命令のコミットは中断できない．逆に，コミットのコミットを行うまでは，コミットを中断できる．

P/M ステートの更新は，LRF と L1D$とで足並みをそろえなければならない．そのため，コミットの開始で合わせた足並みは，コミットのコミットに至るまではそろえておく必要がある．コミットを中断する際も，足並みをそろえて中断する．

3.3　LSQ の問題点

LSQ において，Out-of-Order にデータ・アレイが更新されるなかで，逐次的に実行した場合と論理的に等価な P/M ステートを実現するには，コミットのコミット＝コミットを開始（commit ポインタの更新）したすべての命令を，いつか必ずデータ・アレイに反映させればよい．

しかし, この方法は LSQ のポインタでの TF に対して脆弱である．例を**図 3.3** に示す．

この図は，commit ポインタに TF が発生して，指すエントリが Ins3 から Ins5 へずれていることを表している．コミットを開始したとされる命令は必ず P/M ステートへ反映されるため，この例では，この後に Ins4 までを確実に L1D$へ書き込むことになる．

本来ならば Ins2 までの書き込みを行うことで P/M ステートが正しく保たれるはずであるが，TF の影響で LRF/L1D$ の足並みがそろわなくなる．

こうして，P/M ステートが TF の影響でコミットされてしまうという問題が生じる．

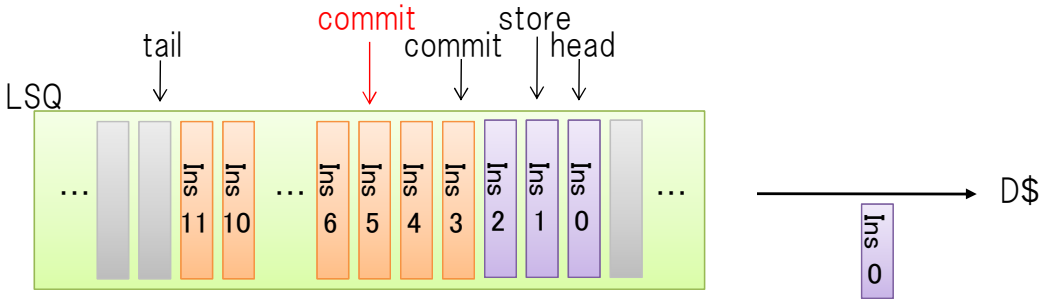

図 3.3: LSQ 上での TF

3.4 RazorII の限界

以上のようなコミットに対する考慮が不十分であるため，RazorII の手法は複雑な Out-of-Order スーパスカラ・プロセッサには適応できない．

さらに，図 3.3 の状態から回復しようとしてフラッシュを行ったとしても，FIFO のポインタとカウンタとの齟齬はもとには戻らない．

ポインタのような制御系のレジスタは実行プログラムとは独立に動作しており，フラッシュをしても以前と同じ正しい状態には戻らないからである．

さらに，数サイクル前の状態にしようとポインタを動かすため，カウンタとの齟齬は解消せず，TF の影響が残り続ける．

実際は，RazorII[6] で紹介されているようなごく単純なスカラ・プロセッサであっても，ストア・バッファなどのバッファを持っており，こういったバッファには制御系のレジスタが存在する．

そのため，RazorII の手法には限界があり，本当に限られた単純なプロセッサにしか適用できない．

第4章　提案手法

前章では，既存手法の RazorII は実用的なプロセッサに適用できないことをみた．我々は，前章までの問題を解決し，複雑な Out-of-Order スーパスカラ・プロセッサに適用できる手法を提案する．

4.1　前提モデル

Out-of-Order スーパスカラ・プロセッサの構成

Out-of-Order スーパスカラ・プロセッサの基本的な構成として，物理レジスタ・ファイル に対するリネーミングを行う方式と，ROB と LRF を組みわせる方式がある．

P/M ステートを扱うには後者の方がわかりやすいため，我々はそちらを仮定する[1]．**図 4.1** に概観を示す．

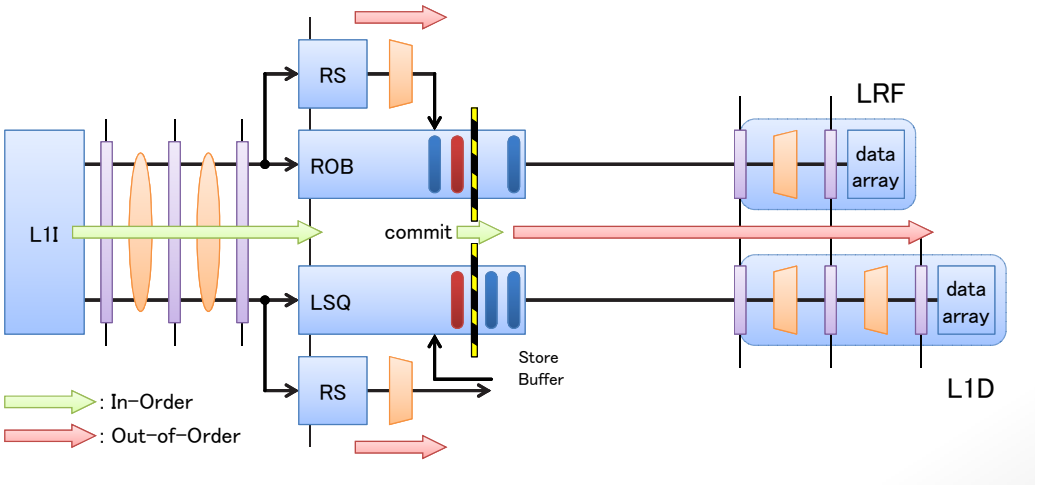

図 4.1: 前提モデル

[1]ただし，前者に対する変更もおそらく可能である．

書き込みのフォールト・モデル

P/M ステートの更新は RAM で構成される ROB や L1D$ への書き込みである．本稿では，書き込み時のフォールトを，書き込みの失敗と誤書き込みとに分けて考える．

書き込みの失敗とは，指定したエントリに異なる値を書いたり，正しい値が書き込まれないことを指す．誤書き込みとは，指定したエントリとは別のエントリを書きつぶすことを指す．

我々が仮定するモデルでは，書き込みの失敗は RazorFF のように事後的に検出される．たとえば，書き込み保証バッファ [8] などがそれを可能にする．また，RAM の回路構成上，原理的に誤書き込みは起こらないと想定している．

4.2 提案手法の概略

4.2.1 TF の検出/回復

提案手法では，プロセッサを Faulty ドメインと Fault-Free ドメインとに分ける．Faulty ドメインは，TF が発生することを認め，Fault-Free ドメインは，TF が発生することを認めない．

Out-of-Order スーパスカラ・プロセッサでは原則，以下のようにするのが妥当である．

- In-Order モジュール $\subset$ Fault-Free ドメイン
- Out-of-Order モジュール $\subset$ Faulty ドメイン

Out-of-Order は複雑で Fault-Free であることを保証することが困難だからである．

TF 検出/回復の基本的な考え方は，RaozrII の手法と同じである．すなわち，

1. Faulty ドメインでの TF を，RazorII FF 等の既存の手法で検出し，TF 通知ネットワークに送る．
2. 検出された情報は TF 通知ネットワークを通してコミット・ステージに送られる．
3. TF が検出されたらコミットを停止し，上流のパイプラインを無効化したあと，Fault-Free ドメインに含まれる P/M ステートから再実行する．

4.2.2　3つの”I”

既存の手法との違いは，一言でいうと３つの”I”である．まず，

1. **Initialization of Pipeline** ．フラッシュではなく，初期化によって無効化する．そのためには，
2. **In-Order Commitment** を実現，すなわち，プログラム・オーダ上で，ここより前はすべてコミット，それより後はすべてキャンセル，というルールの遵守．その結果，
3. **Imprecise Cancellation** ，すなわち，実際に TF の影響を受けた命令（特定できない場合もある）よりも前の命令もキャンセルすることができる．

以下，もう少し詳しく説明する．

(Initialization of Pipeline)

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．フラッシュが特定サイクル前の状態に戻す操作であるのに対し，初期化では初期状態に戻す．

初期状態には TF の影響は含まれていないため，制御系の TF の影響が回復後にも残る，という RazorII の課題を解決する．

(In-Order Commitment)

プログラム順に狭義のコミットをした命令までのすべての命令が，プログラム順にデータ・アレイに反映されることを保証する．

また，論理的にだけではなく，そのような P/M ステートを物理的に確実に実現することを保証する．

実現方法は後述する．

(Imprecise Cancellation)

Out-of-Order スーパスカラ・プロセッサの場合，TF の影響よりも TF 検出情報の方が先にコミット・ステージに到達することがある．これは，TF 通知ネットワークが単純なパイプラインで構成されるからである．

コミットの停止は，RazorII の手法のように TF を受けた命令を特定して行う必要はない．TF の発生が判明した時点で狭義のコミットをしていないすべての命令をキャンセルする．

Out-of-Order スーパスカラ・プロセッサの場合，フロントエンドで TF を起こした命令を命令ウィンドウに格納して…というように precise にするのは極めて困難であり，imprecise なキャンセルによって設計の自由度が大きく上がる．

4.2.3 コミット方式

従来のコミット方式では，LSQ の内部にストア・バッファが存在している（store ポインタ と commit ポインタ との間 ）が，この方式だとポインタの TF に脆弱であるという課題がある．

そのため，我々はストア・バッファを LSQ から分離するコミット方式を提案する．（**図 4.2**）

ROB/LSQ で同期した命令は，ストア命令も含めてただちに ROB/LSQ から送り出される．（ただし，実際に書き込みが完了するまでは，ROB/LSQ のエントリは残しておく）こうすることで，ROB/LSQ から命令を送り出すところが狭義のコミットになる．

ここにスタビライズ・ステージをおき，上流の TF の影響を絶対に下流に流さないようにすることで，Faulty ドメインの TF による P/M ステートの誤った更新を防ぐことができる．

狭義のコミットをした命令はすべて送り出しているため，それらを確実にデータ・アレイに反映させれば P/M ステートは逐次的な更新を正しく実現する．そのため，仮に LSQ 内でポインタに TF が起こった場合は，ROB/LSQ 内のエントリはすべて捨てて無効化してかまわない．従来の課題はこうして解決する．

この方式では，ROB と LSQ との動作はほとんど変わらず，LSQ の store ポインタは必要なくなる．両モジュールとも，commit ポインタ 〜 head ポインタのエントリには，コミットして ROB/LSQ から（足並みをそろえて）追い出したが，まだ LRF/L1D$への書き込みが完了していない命令，が存在する．

4.3 In-Order Commitment の 実現

前節のコミット方式によって，狭義のコミット自体は正しく行われることが保証される．

ここで，In-Order Commitment を実現する条件は，**狭義のコミットをした命令が，狭義のコミットをした順に（いつかは必ず）確実に LRF/L1D$ のデータ・アレイに書き込まれること**ということができる．

提案方式においては，

1. LSQ と分離したストア・バッファが Fault-Free であること

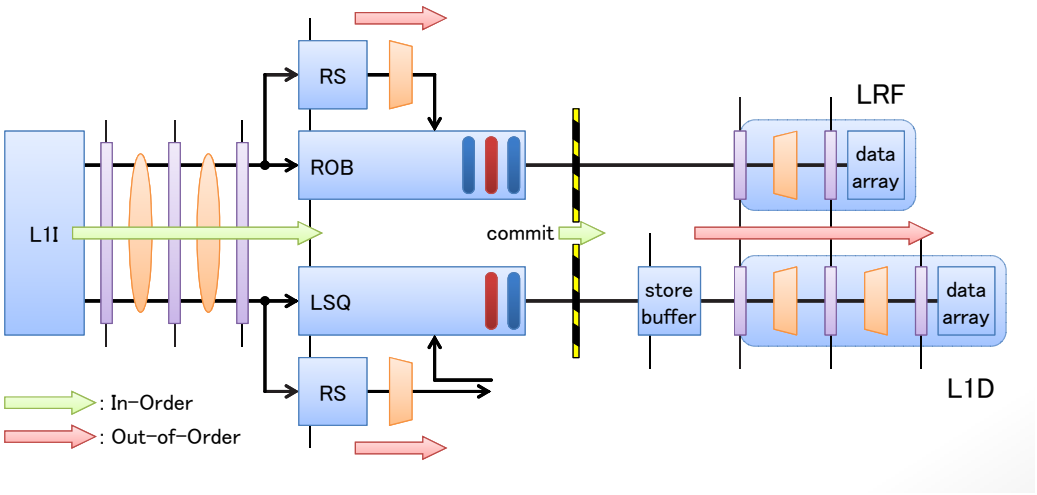

図 4.2: 提案モデル

2. LRF/L1D$への書き込み失敗がリカバリできること

を満たせば，上記の条件を達成することがきる．

4.3.1 Fault-Free なストア・バッファ

ストア・バッファが Fault-Free になるように，比較的単純なシフト・レジスタでストア・バッファを構成することを考える．ただし，ストア・バッファでサイクル・スチールの待ちが発生するため，ストア・バッファ内に存在する命令は多かったり少なかったりする．

ここで，ストア・バッファ内の命令数をみて，シフト・レジスタの段数が変わるようにするデザインと，段数の変わらないごくごく単純なシフト・レジスタで組むデザインとが考えられる．

前者はたとえば，**図 4.4** のようなデザインが考えられる．この図では，最長 6 段のシフト・レジスタになっており，中にいる命令数が少なければセレクタを使って 4 段や 2 段に変更することができる．セレクタはローカルに近くのレジスタや命令の Valid 情報をみて，どちらかのパスを選ぶ．

固定長のシフト・レジスタで組むと，より単純になるが，確実に最長の段数を経なければストア・バッファから出てこれないという点で，性能が落ちる可能性がある．

後の 5 章の予備評価でも述べるが，ストア・バッファのエントリ数は（2〜4 程度に）少なくても性能にほとんど影響がない．また，そのため固定長の単純なつくりでも問題ない．

そのため小さくつくることができ，Fault-Free にすることが可能であるといえる．我々は，2〜4 エントリの単純なシフト・レジスタとしてデザインすることを考えている．

4.3.2 書き込み失敗からのリカバリ

LRF の場合

LRF への書き込み失敗は，事後的に判明する．
ここからのリカバリ方法として，我々は**図 4.3** のようなデザインを考える．
具体的には以下のようにする．

1. LRF へアクセスする際に，その入力をコピーして，コピーをそのまま流し続ける並列のパイプラインに流す．
2. 書き込み失敗が判明したら，その時点で狭義のコミットを停止する．
3. 並列のパイプラインにその時点で存在するコピーをすべて，順番に LRF へ入力する．
4. コピーがすべて正しく書き込めたら終了，失敗があれば同じことを繰り返す．

狭義のコミットが停止していない場合，並列のパイプラインには，常に，狭義のコミットをしたばかりの命令列がプログラム順に入っている．
さらに，単純なパイプラインなので Fault-Free でつくることができ，常に正しい値を持つ．
そのため，上記の方法で，狭義のコミットをしたすべての命令を正しくプログラム順に確実に LRF に反映することができる．

L1D$の場合

L1D$の書き込み失敗の場合も LRF の場合とほとんど同じである．(図 4.4)
ストア・バッファがある分だけ，少し異なり，以下のようになる．

1. L1D$へアクセスする際に，その入力をコピーして，コピーをそのまま流し続ける並列のパイプラインに流す．
2. 書き込み失敗が判明したら，その時点で狭義のコミット，および，ストア・バッファから L1D$へのストアの送出，を停止する．
3. 並列のパイプラインにその時点で存在するコピーをすべて，順番に L1D$へ入力する．

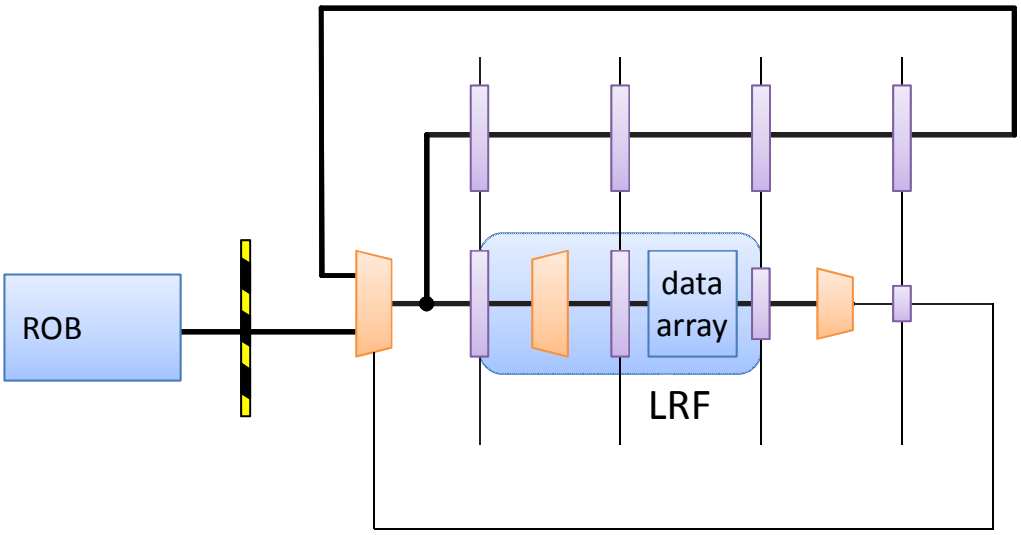

図 4.3: LRF への書き込み失敗からのリカバリ

4. コピーがすべて正しく書き込めたら，その後にストア・バッファにあるストアをすべて L1D$へ書き込んで終了，失敗があれば同じことを繰り返す.

4.4 議論

今回のように、故障耐性や信頼性を議論するときには，問題を網羅的・包括的に扱えることが大事である.

すなわち,「ここだけ頑張ればいい」，さらに「絶対にここだけ」，と言えるとよい.

この観点から，今回の TF 検出/回復について考える.

- 狭義のコミットより前の Fauty ドメインで発生する TF は，狭義のコミットによって無効化，さらに初期化によって TF の影響をすべて除去できるため，包括的に扱うことができる.
- ゆえに，P/M ステートを破壊するとすれば，狭義のコミットより後ろの Fault-Free ドメインだけ（絶対にここだけ）であり，個々に Fault-Free であることをたしかめた（そこだけ頑張ってつくる）.

よって，提案の方式は Out-of-Order スーパスカラ・プロセッサ上の TF について，包括的に扱っているということができる．[2]

[2]もちろん，従来の LSQ のモデルで，ROB/LSQ の制御系には絶対に TF が起こらないよう

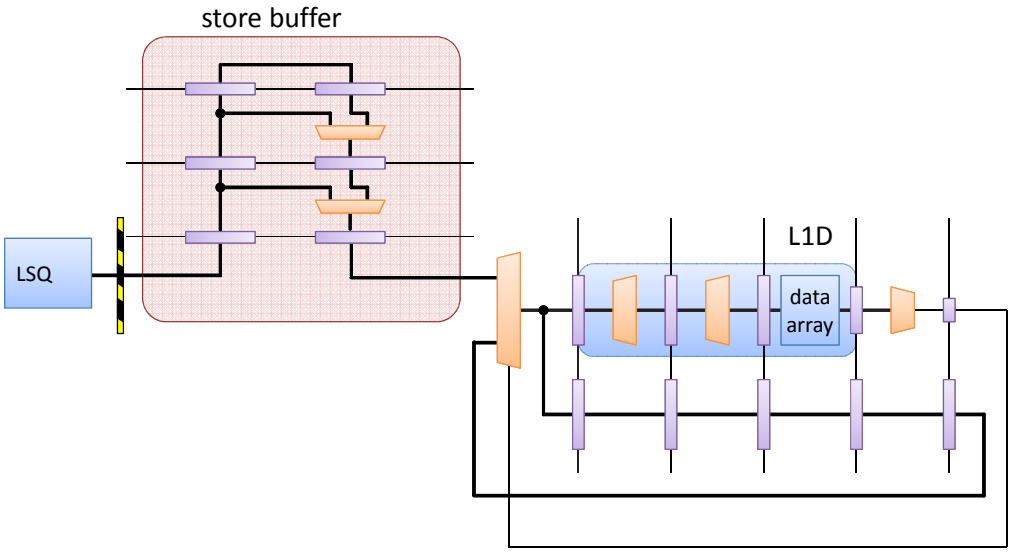

図 4.4: L1D$への書き込み失敗からのリカバリ

に頑張る，ということも可能である．しかし，3 章でみたように，具体的に複雑なことが分かっている Out-of-Order なモジュールに対して，TF が起こらないように頑張る，というのは現実的ではない．

第5章　予備評価

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・バッファを分離すると，ストア・バッファを経由する分だけストアする際のレイテンシが伸びる．そこで，そのレイテンシの増加による性能変化を調べるため予備評価を行った．

Fault-Freeな構成にするため，ストア・バッファは比較的単純な回路となるシフト・レジスタでデザインする．その際のモデルとして，① 可変長のシフト・レジスタ（図4.4のようなイメージ）で，ストア・バッファ内の有効なエントリ数を見ながら最短のサイクル数でストア・バッファを抜けていくモデルと，② 単純なシフト・レジスタで，ストア・バッファに入ってから出ていくまで最短でもエントリ数分のサイクルが必要になるモデルを考える．

評価はプロセッサ・シミュレータ**鬼斬弐**[9]に対し，以下のモデルを実装した．評価したプロセッサのパラメータは**表 5.1**の通りである．ベンチマークはSPECCPU 2006[10]を用いた．

- **Baseline**　従来のストア・バッファモデル．LSQ内部にある．
- **NoFIX**　ストア・バッファを分離し，最短のサイクル数でストアが抜けていくモデル．
- **FIX**　ストア・バッファを分離し，ストアが抜けていくのにエントリ数分のサイクルが必要なモデル．

なお，ストア・バッファのエントリ・サイズは，従来のモデルでは32（LSQとunifiedなため），提案手法ではパラメタ化し，2,4,8,16, のそれぞれで評価を行った．またNoFIXのモデルにおいては，ストア・バッファを抜けるのに最低でも2サイクルかかるものとした．

表 5.1: プロセッサの構成

パラメータ	値
ISA	Alpha 21164A
logical thread	4 way
fetch width	4 inst.
execution unit	int : 2, fp : 2, mem : 2.
instruction window	int : 32, fp : 16, mem : 16
register file	int : 256, fp : 256
load/store queue	Unified, 32 entries
load/store ports	1-read + 1-read/write, cycle strealing
branch prediction	8KB g-share
miss penalty	10 cycle
BTB	2K entry, 4-way
L1C	32KB, 4-way, 64B/line, 3 cycle
L2C	4MB, 8-way, 64B/line, 15 cycle
main memory	200 cycle

5.1 評価結果

図 5.1 に，Baseline に対する提案手法の相対 IPC を示す．また，**図 5.2** にストア命令のストア・バッファの平均滞留サイクル数を，**図 5.3** では NoFIX のモデルでストア・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．

これらの評価結果に関しては，SPECCPU 2006 の中で特徴的な 4 つのベンチマークを抽出し，全 29 種類のベンチマークの平均値と共に示している．

図 5.1 は，ストア・バッファを分離することによる性能低下がほとんどないことを示している．このグラフでは，性能低下が平均 0.7 %，最も悪い soplex でも 6.0 %に留まっていることがわかる．

ストア・バッファの分離によるレイテンシの増加で危惧されるのは，レイテンシの増加分だけ LSQ のエントリの解放が遅れる可能性であり，それによる資源不足でパイプライン上流がストールすることである．

しかし，図 5.3 にあるように，ロード命令のサイクルあたりの L1D$ポート利用数は平均でも 0.3，最悪の h264ref でも 0.6 と低い値となっている．この結果が示すように，L1D$のポート をロード命令が独占している（サイクルあたり 2 ポート利用する）確率は極めて低く，サイクル・スチールであっても，ほ

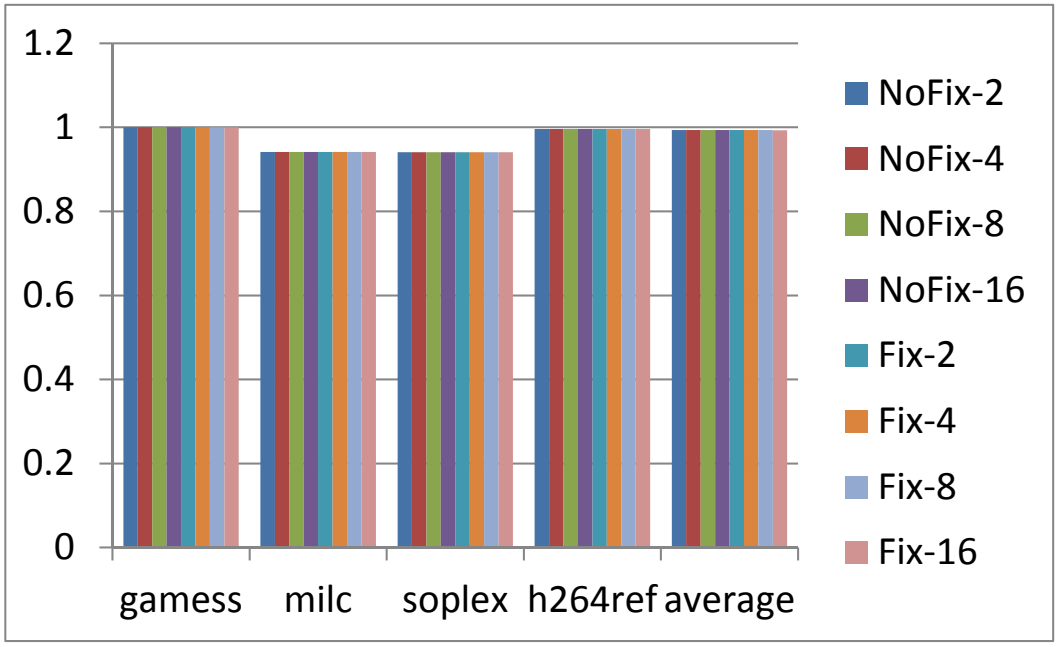

図 5.1: ベースラインに対する相対 IPC

ぼ毎サイクル，ストアがポートを利用することができるため，LSQ の資源が足りなくなるほどに多くの命令がストア・バッファにとどまることはない．

図 5.2 では，提案手法のモデルが FIX であっても NoFIX であっても，ほとんどの場合においてストア・バッファを抜けるのに必要な最低サイクル数しかかからないことを示している．FIX モデルで 2 エントリのモデルを例にとると，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付けている．

以上の結果から，TF 耐性を持たせるためにストア・バッファを LSQ から分離する提案手法は，分離することによる性能低下がほとんどないため現実的であるといえる．

さらに，サイクル･スチールであっても，ストア命令はほぼ毎サイクル L1D$のポートを利用することができるため，ストア・バッファのエントリ数は小さくてよい．

具体的には 2 エントリ程度あれば良く，抜けるのに必要なサイクル数が固定の単純なシフト・レジスタでストア・バッファを組むことによって，より Fault-Free なストア・バッファを実現することができる．

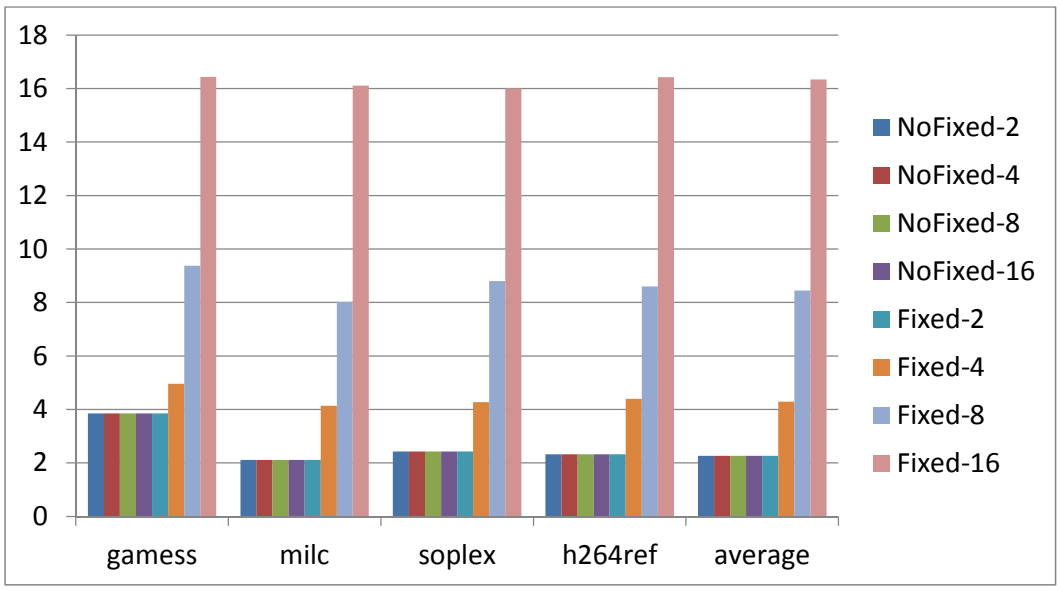

図 5.2: ストア・バッファの平均滞留サイクル

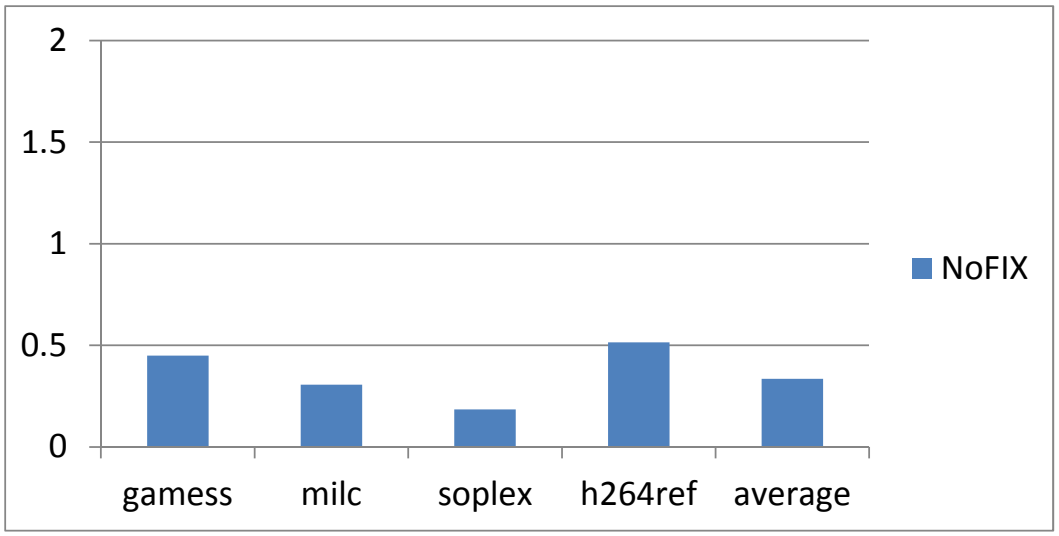

図 5.3: ロード命令のサイクルあたり平均ポート利用数

第6章　まとめと今後の課題

本稿では，微細化によるばらつき増大の問題への対策の 1 つである，TF と DVFS とを協調させる手法を念頭に，複雑な Out-of-Order スーパスカラ・プロセッサであっても TF から正しく回復できる手法を提案した．

既存の手法では 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，コミット・モジュールの LSQ で起こる制御系の TF に対して脆弱であり，正しい P/M ステートが破壊される可能性がある．また，一般的な制御系の TF から正しく回復することができない．

一方，我々の提案手法では，LSQ からストア・バッファを分離することで，LSQ 内の TF による P/M ステートの更新を防ぐ．ここで，ストア・バッファを分離することによる性能低下は，シミュレーションの結果，ほとんどないことを確認した．また，提案手法では，制御系の TF も初期化によって正しく回復させることができる．

コミット後の Fault-Free ドメインにおいては，書き込みの失敗があっても正しくリカバリできる手法について提案した．分離したストア・バッファは小容量の単純なシフト・レジスタで組み，Fault-Free にデザインするが，それでも性能が落ちないことをシミュレーションで確認した．

今後の課題としては，ROB + LRF 方式ではない Out-of-Order スーパスカラ・プロセッサに，提案手法を組み合わせることである．本研究室では現在，NORCS[11] など様々な技術を取り入れたプロセッサの設計を行っているが，具体的には，これに提案手法を実装するための検討を現在進めている．

参考文献

[1] 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. Vol. 92, No. 6, 2009.

[2] 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. 京都大学博士論文, 2003.

[3] 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)*, April 2006.

[4] S. Mukhopadhyay, H. Mahmoodi, and K. Roy. 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.

[5] Arindam Mallik, Jack Cosgrove, Robert P. Dick, Gokhan Memik, Peter Dinda. Picsel: measuring user-perceived performance to control dynamic frequency scaling. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pp. 70–79, 2008.

[6] D. Blaauw, S. Kalaiselvan, K. Lai, Wei-Hsiang Ma, S. Pant, C. Tokunaga, S. Das, and D Bull. Razor II: In Situ Error Detection and Correction for PVT and SER Tolerance. In *Int'l Symp. on Solid-State Circuits Conference (ISSCC)*, 2008.

[7] 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 speculation. In *Int'l Symp. on Microarchitecture (MICRO)*, pp. 7–18, 2003.

[8] Hidetsugu Irie, Ken Sugimoto, Masahiro Goshima, Suichi Sakai. Preventing timing errors on register writes: Mechanisms of detections and recoveries. *ACM SIGARCH Computer Architecture News(ACM CAN)*, Vol. 35, No. SIG5, pp. 25–31, 2007.

[9] 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.

[10] The Standard Performance Evaluation Corporation. *SPEC CPU2006 suite http://www.spec.org/cpu2006/*.

[11] Ryota Shioya, Kazuo Horio, Masahiro Goshima, and Shuichi Sakai. Register cache system not for latency reduction purpose. In *Int'l Symp. on Microarchitecture (MICRO-43)*, pp. 301–312, 2010.

研究業績

主著論文

1. 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
   有馬 慧
   卒業論文，東京大学 工学部 (Feb. 2010)

2. 過渡故障耐性を持つ Out-of-Order スーパスカラ・プロセッサの評価
   有馬 慧, 岡田 崇志, 堀尾 一生, 喜多 貴信, 塩谷 亮太, 五島 正裕, 坂井 修一
   情報処理学会 第 72 回全国大会 (2010)

3. 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
   有馬 慧, 岡田 崇志, 喜多 貴信, 塩谷 亮太, 五島 正裕, 坂井 修一
   電子情報通信学会研究報告 CPSY2010-5, pp. 21-26 (2010)

4. 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, 岡田 崇志, 塩谷 亮太, 五島 正裕, 坂井 修一
   情報処理学会研究報告 2010-ARC-190, No.10 (2010)

5. 耐過渡故障耐性を持つ Out-of-Order スーパスカラ・プロセッサ
   有馬 慧, 岡田 崇志, 塩谷 亮太, 五島 正裕, 坂井 修一
   電子情報通信学会研究報告 CPSY2011-5, pp. 23-28 (2011)

6. タイミング・フォールト耐性を持つ Out-of-Order プロセッサ
   有馬 慧, 倉田 成己, 塩谷 亮太, 五島 正裕, 坂井 修一
   先進的計算基盤システムシンポジウム SACSIS 2012
   （投稿中）

共著論文

1. 動的タイムボローイングを可能にするクロッキング方式
   吉田 宗史, 有馬 慧, 岡田 崇志, 塩谷 亮太, 五島 正裕, 坂井 修一
   情報処理学会 第 73 回全国大会, pp.1-91‐1-92 (2011)

2. 動的タイムボローイングを可能にするクロッキング方式の予備実験
   吉田 宗史, 有馬 慧, 倉田 成己, 塩谷 亮太, 五島 正裕, 坂井 修一
   電子情報通信学会研究報告 CPSY2011-7, pp. 13-18 (2011)

受賞

1. 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
   情報処理学会創立 50 周年記念第 72 回全国大会, 情報処理学会推奨卒業論文認定 (2010)

謝辞

非常に多くの方々から多大なご指導、ご協力、励ましを頂き、本論文を完成させることができました。この場を借りて、感謝の意を表したいと思います。

指導教官である五島正裕准教授には学部4年からの3年間に渡って多くのご指導，ご助言を頂きました．坂井修一教授からも，大変多くのご指導を頂きました．ここに深く感謝の意を表します.

名古屋大学の塩谷亮太助教には，東京大学在学時よりミーティングなどでいつも指導を行って頂いたのを始めとして，様々な点でお世話になりました.

八木原晴水さん，長谷部環さん，研究室における設備の導入や各種事務手続きなど，研究室で過ごすための様々なご支援を頂きました.

研究室同期である倉田成己氏には，実装・評価において大変お世話になりました．都井紘氏には，異なった視点からのご助言を多く頂きました.

また，ここでは紹介しきれなかった研究室のメンバーの皆様にも様々な形でお世話になりました．本当にありがとうございます.