修士論文

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

Timing-Fault-Tolerant Superscalar Processor

指導教員　坂井修一 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

48-076417 杉本 健

概要

プロセッサの動作周波数は，クリティカルな回路遅延に，静的に定められたタイミング・マージンを加えて決定される．しかし，近年では，製造ばらつきや動作時の温度ばらつきが大きくなり，悲観的なマージンを設計時に確保することがコストに見合わなくなってきている．今後のプロセッサでは，回路の応答がクロック同期とずれてしまう，動的なタイミング・フォルトが深刻な問題となると予想される．

この問題に対し，タイミング・フォルトを動的に検出/回復に関する手法を提案されている．これに関する既存の研究はデータパス系のタイミング・フォルトのみに着目しており，制御系で発生するタイミング・フォルトが考慮されていない．しかし，out-of-order スーパスカラ・プロセッサは，wakeup/select のような，これ以上パイプライン化できない制御系の，遅延の長いパスを多く含む．そのため，制御系がタイミング・フォルトを起こす確率は無視できないものとなっている．それに対し，本稿では，制御系を含めたいかなる場所でタイミング・フォルトが発生しても正しく回復ができる手法を提案する．この手法では，フォルトをコミット・ステージまで通知し，コミット・ステージでフォルトが検出された場合，レジスタ・ファイルと PC によって定義されるアーキテクチャ・ステートを更新せずに，プロセッサをリセットする．リセットの結果，制御系を含むすべてのレジスタが初期状態＋正しいアーキテクチャ・ステートとなるので，制御系を含めた，すべてのタイミング・フォルトから回復することができる．また，提案手法を適用した out-of-order スーパスカラ・プロセッサを FPGA に実装し，正しく動作することを確認した．

目 次

第1章　はじめに **1**

第2章　タイミング・フォルト **2**

2.1 タイミング・フォルトとは 2
2.2 設計/製造とタイミング・フォルト 2
2.3 微細化とバラつき 3
2.4 実際に近い遅延に基づく設計/製造手法 4
2.5 タイミング・フォルト検出/回復 4
2.6 DVFS 制御 5
2.7 既存の動的タイミング・フォルト検出技術 7
2.7.1 RAZOR 7
2.7.2 カナリア・フリップフロップ 8
2.8 タイミング・フォルトからの回復技術 9
2.8.1 回路レベルの回復技術 9
2.8.2 アーキテクチャ・レベルの回復技術 10
2.8.3 制御系のフォルトからの回復の必要性 12
2.8.4 制御系のフォルトからの回復の困難 13

第3章　提案手法 **16**

3.1 フォルト通知ネットワーク 16
3.1.1 回路レベルのフォルト通知ネットワーク 18
3.1.2 パイプライン化 19
3.2 リセット 20
3.2.1 リセットによる回復 20
3.2.2 リセットのレベル 21
3.3 議論 22

第4章　実装 **24**

4.1 実装の意味 24
4.2 ボード 25

4.3 FPGA 及び開発環境 26
4.4 命令セット及びビット幅 26
4.5 全体構成 26
4.6 パイプライン 27
4.7 提案手法の実装 27

第 5 章 評価 29
5.1 提案手法の有効性の確認 29
5.2 フォルト通知ネットワーク 30
5.2.1 クリティカル・パス 31
5.2.2 ハードウェア量 32

第 6 章 関連研究 34

第 7 章 おわりに 35

参考文献 36

発表文献 38

図目次

2.1 タイミング・フォルトの例 3
2.2 DVFS 技術の概要 6
2.3 RAZOR 8
2.4 カナリア・フリップフロップ 9
2.5 制御系とデータパス系の面積 12
2.6 制御系とデータパス系の面積の割合 13
2.7 遅延の分布 14
2.8 遅延の分布のデータパス系と制御系との割合 15

3.1 SPARC64 V のフォルト通知ネットワーク 17
3.2 パイプライン化されたフォルト通知ネットワーク 18
3.3 リセット 20

4.1 動作周波数・電源電圧可変 FPGA ボード 25
4.2 実装した out-of-order スーパスカラ・プロセッサの全体図 . . . 26

5.1 実験装置 30

表目次

5.1 パイプラインの各段における FNN 31

第1章　はじめに

今日，コンピュータ・システムは社会の隅々にまで浸透しており，その中核となるマイクロプロセッサには非常に高い信頼性が求められている．

マイクロプロセッサの信頼性を脅かす要因のうち，今後深刻となると予想されるものの一つに，動的な**タイミング・フォルト**がある．動的なタイミング・フォルトは，回路遅延の動的な変化によって，信号のタイミングに齟齬が生じ，設計者の想定外の動作が生じる過渡故障である．

従来, タイミング・フォルトは，設計，製造時の問題であって，出荷後の発生は, 熱暴走のようなケースを除き, ほとんど問題とならなかった．しかし，次章で詳しく述べるように，今後は，LSI の製造プロセス微細化によって，設計，製造時の対処のみでは，著しく非効率になると予想されている．

そこで，有効と考えられるのが，タイミング・フォルトを動的に検出，回復するハードウェア手法である．

そのような動的なタイミング・フォルト対策技術として，**RAZOR**[2, 8]，**カナリア FF**[12, 13]，**書き込み保証バッファ**[14, 3] などが既に提案されている．しかし，これらの技術は検出についての議論が主である．回復についてはデータパス系のみに着目したものばかりで，制御系にタイミング・フォルトが発生した場合の議論がなされていない．しかし，out-of-order スーパスカラ・プロセッサのパスには，wakeup/select といった，これ以上パイプライン化することのできない，制御系のクリティカル・パスが存在する．そのため，制御系で発生するタイミング・フォルトを無視することはできない．

そこで，本論文では，out-of-order スーパスカラ・プロセッサにおいて，制御系を含めた，あらゆる FF において発生したタイミング・フォルトを動的に検出/回復する手法を提案する．

本論文は以下，次のように構成される．まず，2 章で，タイミング・フォルトについて述べる．タイミング・フォルトの既存の対策手法とその問題点についても 2 章で述べる．3 章で，提案手法について説明する．4 章で，FPGA に実装した out-of-order スーパスカラ・プロセッサについて述べる．5 章で評価結果を述べる．6 章で関連研究について述べ，最後に 7 章でまとめと今後の課題について述べる．

第2章　タイミング・フォルト

2.1　タイミング・フォルトとは

タイミング・フォルトとは，回路の遅延の増減により, 設計時の想定外の挙動を起こす過渡故障である. 同期回路における典型的なタイミング・フォルトは，遅延の増加によってロジックの出力がクロック期間内に安定せず，前のサイクルの出力がサンプリングされる事である．図 2.1 に，そのような例を示す. この例では，bit1 と bit4 を生成するロジックの遅延が何らかの原因で増加し，前のサイクルの値が取り込まれている.

2.2　設計/製造とタイミング・フォルト

従来の設計/製造は，タイミング・フォルトを起こさないために，以下のように進められる：

1. 回路遅延が設計に対して変動する要因は，主に，製造ばらつき (**P**rocess), 電源電圧 (**V**oltage)，温度 (**T**emperature)，の 3 つである．設計時には，まず，それぞれについて許容範囲を設定する．例えば，電源電圧は 1.10V ± 5%, 温度は 0°C〜85°C，そして，製造ばらつきについては,「ロジックは 3σ, メモリは 6σ 以内」といった具合である.
2. その上で，すべての動作条件 —— 特に最悪条件において，正しく動作するように設計する.
3. それでも，製造ばらつきが許容範囲外であるようなチップが製造されることは原理的に避けられない．そのようなチップは出荷検査によって取り除かれる.

したがって，出荷後には，電源電圧と温度が設定された許容範囲内に収まっている限り，タイミング・フォルトは（ほとんど）発生しないことになる.

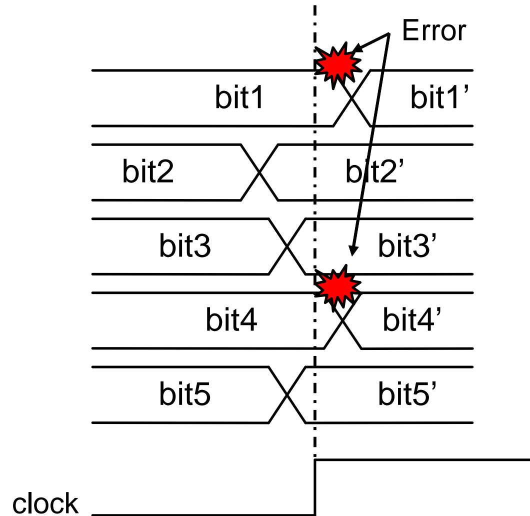

図 2.1: タイミング・フォルトの例

2.3 微細化とバラつき

しかし近年では，LSI の微細化に伴い，同じチップ内でも回路の性能が大きくバラつくことが問題となりつつある [11].

チップ内バラつきとの原因としては，具体的には以下が挙げられる：

1. 製造時のバラつき
   (a) 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る各加工寸法のバラつき
   (b) 不純物密度分布のバラつき
2. 動作時の消費電流に起因するバラつき
   (a) IR ドロップによって生じる電源電圧のバラつき
   (b) 温度分布のバラつき

これらの要因により，トランジスタの閾値電圧や ON/OFF 抵抗値，配線の抵抗値がバラつき，回路の遅延と消費電流量がバラつくことになる．また，その結果，動作時の消費電流量のバラつきがさらに増大する負のフィードバックが存在する．

これらは，加工寸法が原子のサイズに近いたことによるものであるので，悪化を完全に抑えることは原理的に難しい．

2.4　実際に近い遅延に基づく設計/製造手法

バラつきが増大していくと，前述した最悪値に基づく設計，製造手法は悲観的になりすぎることになる．そのため，設計/製造段階で，より実際に近い遅延に基づく手法が提案されている．

設計時の手法としては，**SSTA** (Statistical Static Timing Analysis)[7] が導入されつつある．パスを構成するトランジスタ，配線の遅延がすべて最悪であるような確率は高くはない．SSTA では，バラ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もりが可能になる．

また，レイアウト後に，チップ内の温度分布や電源電圧分布を精密にシミュレートする手法なども提案されている．

さらに，製造後に，実際の遅延を測定して，クロック・スキューを調整することなども提案されている．この方法では，歩留まりの向上が見込めるほか，見積もりではない，その個体が持つ実際の遅延に基づいて，最高動作周波数，最低電圧での動作が可能になる．

2.5　タイミング・フォルト検出/回復

これら設計時の手法に加えて, 動作時にタイミング・フォルトを検出し，回復する手法が提案されている [8]．この手法は，**DVFS** (Dynamic Voltage and Frequency Scaling) 技術と併用することによって，大きな効果を発揮する．

DVFS とは，電源電圧および動作周波数を動的に変化させることによって，省電力を達成する技術である．

DVFS とタイミング・フォルト検出・回復技術と協調するためには，具体的には以下のようにすればよい：

1. 電源電圧を一定とし，動作周波数を徐々に上げる（あるいは，動作周波数を一定とし，電源電圧を徐々に下げる）．
2. タイミング・フォルトが発生したら，動作周波数（あるいは，電源電圧）を元に戻し，回復を行う．
3. 以降，これを繰り返す．

このようにすることによって，以下のような，2 段階の効果が得られる：

1. 前述した出荷時調整と同様，見積もりではない，その個体の実際の遅延に基づいて，最高動作周波数，最低電圧での動作が可能になる．

2. 実際には，チップ内のクリティカル・パスのすべてが毎サイクル活性化されるわけではない．たとえば加算器のキャリー生成回路では，実際にキャリーが長い距離を伝播しなければ，非常に短い時間で演算を終えることができる[1].

   したがって，上述のような手法を用いれば，チップ全体のクリティカル・パスではなく，実際に活性化されたクリティカル・パス —— いわば，動的なクリティカル・パスによって決まる最高動作周波数，最低電圧での動作が可能になるのである．クリティカル・パスが活性化されるかどうかはロジックの入力 (**I**nput) に依存するので，前述した PVT のバラつきに加えて，入力のバラつきにまで対応できると言ってもよい.

このように，タイミング・フォルトを検出/回復する技術は，前述した実際に近い遅延に基づく設計/製造技術の延長線上にあるととらえることができる.

2.6 DVFS 制御

前節で述べた方法は，タイミング・フォルトが発生した場合，電源電圧と動作周波数のどちらを制御するのか，という点で曖昧である．この制御方法については，議論の余地が大きい．本節では，現在利用されている DVFS 技術における，周波数と電源電圧の制御方法について述べ，その後にタイミング・フォルト検出/回復に利用できる，制御方法の案を示す.

現在利用されている DVFS 技術

現在の，DVFS の仕組みを図 2.2 に示す．まず，プロセッサに適用可能な周波数と電源電圧は 1 対の組となっており，周波数 (もしくは電源電圧) を選択すれば，電源電圧 (もしくは周波数）も決まる．周波数（もしくは電源電圧）の決定は，ソフトウェアによって行われ，プロセッサのベンダーから提供されるドライバを介して，どの周波数と電源電圧の組を用いるのかを，プロセッサに伝える．プロセッサには，動的に周波数と電源電圧を変更する仕組みが備わっており，その仕組みを用いてソフトウェアの要請に応じて各々を変更する．こ

[1]なおこの議論は，非同期設計と比較するといっそう興味深い．標準的な 2 線 2 相式回路では，クロックではなく，回路の実際の遅延に基づいて動作する．したがって，上述の第 1 段階の効果は自然に得られることになる．しかしこの方法では，第 2 段階の効果を得ることはできない．先のキャリー生成回路の例では，2 線 2 相式回路では，キャリーが伝播しなかった場合でも,「しなかった」信号が出力まで伝搬する必要がある.

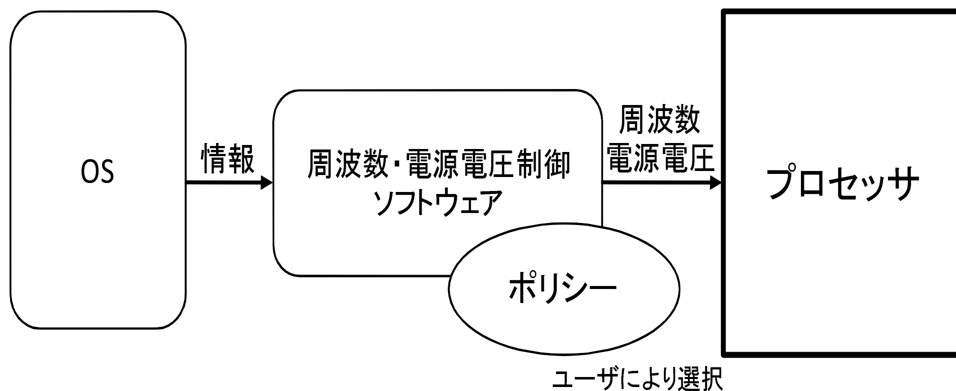

図 2.2: DVFS 技術の概要

の，プロセッサが動的に周波数と電源電圧を変更する仕組みは，Intel のプロセッサでは，Intel Speed Step と呼ばれている．

ソフトウェアは，OS より情報を取り出し，その情報から，周波数と電源電圧を決定するが，その決定の基準は，ポリシーと呼ばれ，実装されている範囲内でユーザが任意に選択可能である．例えば，Windows XP は，4 つのポリシーを持ち，AC 電源駆動の場合とバッテリー駆動の場合それぞれどのポリシーを用いるかの組み合わせを選択可能である．[6] ポリシーは，他にも数多くの手法が提案されている．[5]

電源電圧制御と周波数制御

DVFS では電源電圧や周波数を必要に応じて変化させるのであるが，これらの値を変更するために必要な時間は長い．変化させる幅などにも依存するが，いずれも数 us〜数十 us 程度必要である．これは，動作速度が 1GHz のプロセッサで数千〜数万 clock に相当する．

また，電源電圧を制御する場合，電源電圧をゆっくりと変化させるという条件下では，プロセッサを駆動しつつ電源電圧を変更することができる．それに対して，周波数を制御する場合，変化させている最中，プロセッサを停止しておかなければならない．これは，周波数を変化させる場合は，性能上数千〜数万 clock のペナルティを負うことを意味する．

周波数を変化させる場合に，プロセッサを停止しておかなければならない理由としては，PLL の再ロックに時間がかかることや，ジッタなどの影響により，プロセッサのクロックの同期がとれなくなり，タイミング・フォルト が発生してしまう，ということがあげられる．タイミング・フォルト検出/回復手法と

併用する場合，このときに発生するタイミング・フォルトも正しく検出/回復できる可能性もある．しかし，既存のタイミング・フォルト検出/回復手法は，提案手法含め，調べた限りでは，クロックが急激に変化する場合に対応できない．そのため，周波数を変化させるときにプロセッサを停止する必要性が現状ではあり，必要性をなくすためには，新たな手法が必要である．

タイミング・フォルト検出/回復における DVFS 制御

タイミング・フォルト検出/回復手法へ DVFS を適用させる上で，現在行われている方法と違う点は，現在は周波数と電源電圧がペアになっているのに対して，周波数と電源電圧をそれぞれ独立に制御できることである．言い換えれば，制御するパラメータが 1 つから 2 つに増えているという点である．

制御方法として，次のような，既存の方法を自然に拡張した方法が考えられる．すなわち，現在すでに提案されている手法を用いて，周波数を決定し，その周波数を用いて電源電圧を最小化するという方法である．たとえば，Windows XP に実装されている手法 [6] は，現在必要な周波数を CPU 使用率のから定め，電源電圧は周波数に付随して定める，という方法なので，上記の方法は自然に適用可能である．

2.7 既存の動的タイミング・フォルト検出技術

タイミング・フォルトの回復技術は，どのようにフォルトが検出されるかに依存する．そこで，本節では，回復技術より先に，既存のタイミング・フォルト検出技術について述べる．

タイミング・フォルト検出技術には，検出後に回復を行うことを前提とした，検出技術と，あらかじめタイミング・フォルトの発生を察知し，フォルトの発生を未然に防ぐ予報技術に分けられ，RAZOR とカナリア FF がそれぞれに相当する．以下では，RAZOR とカナリア FF それぞれについて説明する．

両者とも，同じ信号を二度サンプリングすることにより冗長化を行っているため, 別途ロジックを追加する必要がなく，非常に安価な方法である．

2.7.1 RAZOR

Austin らは，ロジック・レベルに近い技術として，**RAZOR**[2, 8] を提案，試作し，評価している．RAZOR では（基本的には）すべてのラッチに対して，やや位相の遅れたクロックで動作する**シャドウ・ラッチ**を付加する．この様子

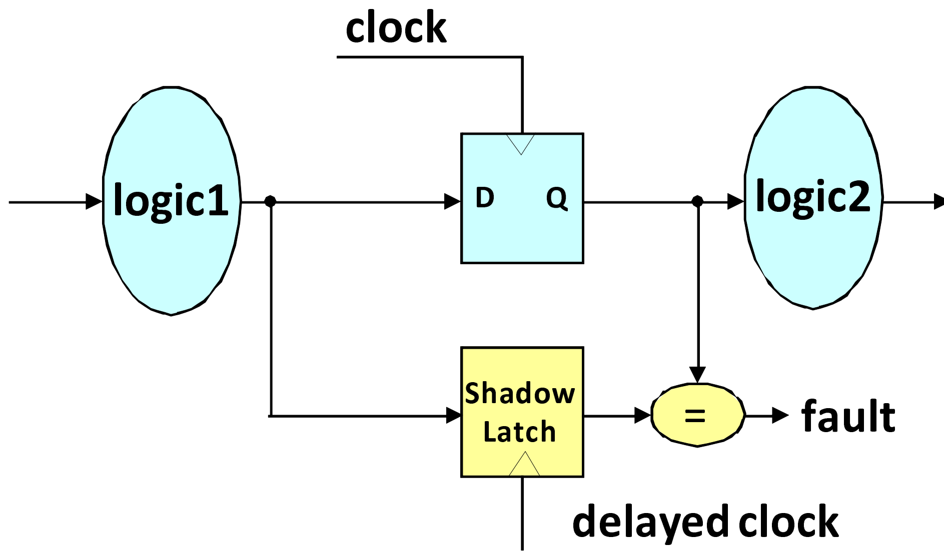

図 2.3: RAZOR

を図 2.3 に示す．ロジックの出力は，通常のクロックで動作するラッチと，シャドウ・ラッチの双方でそれぞれサンプリングされる．ロジックの遅延の増加のために通常のラッチで古い値がサンプリングされても，遅れた位相で動作するシャドウ・ラッチでは正しい値がサンプリングされる可能性が高い．そのため，双方の出力を比較することによりタイミング・フォルトを検出できる．

2.7.2　カナリア・フリップフロップ

佐藤らは，**カナリア・フリップフロップ**[12, 13] を提案，評価している．この手法では，すべてのラッチに対して遅延素子を挿入したカナリア FF 用意する．この様子を図 2.4 に示す．カナリア FF は，メイン FF よりもタイミング制約が厳しい．そのため，徐々にタイミング制約を厳しくしてゆくと，カナリア FF の方が先にタイミング・フォルトに遭遇する．従って，メイン FF とカナリア FF の値を比較することにより，タイミング・フォルトの予報を行うことができる．

カナリア FF はメイン FF にタイミング・フォルトが起こらないので，回復を行う必要性がない，という点で優れている．

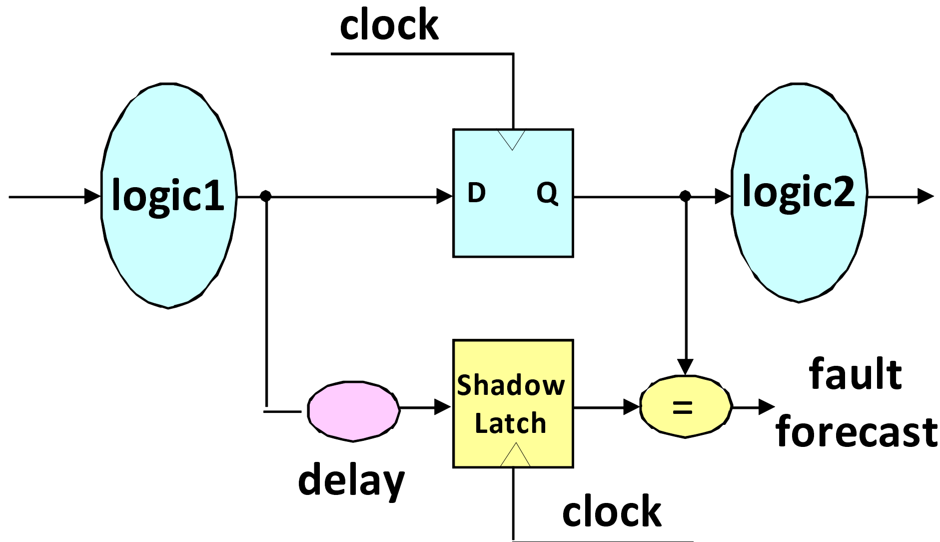

図 2.4: カナリア・フリップフロップ

2.8　タイミング・フォルトからの回復技術

前章で述べたRAZORなどの技術によってタイミング・フォルトが検出された場合には，そこから回復を行うことになる．本章では，そのような回復技術について述べる．

タイミング・フォルトからの回復技術には，RAZORなどを自然に応用したもののほか，対象がプロセッサである場合には，その性質を利用ものがある．そこで，前者を**回路レベル**，後者を**アーキテクチャ・レベル**の回復技術ということにし，以下それぞれについて説明する．

2.8.1　回路レベルの回復技術

前述したように，RAZORがタイミング・フォルトを検出した時には，そのメイン・ラッチの正しい値は対応するシャドウ・ラッチに格納されている．したがって，このラッチに限って言えは，シャドウ・ラッチの値をメイン・ラッチに戻せばタイミング・フォルト発生前の状態を回復することができる．

しかし，回路全体をタイミング・フォルト発生以前の状態に回復するためには，タイミング・フォルトが検出されたサイクル内に，回路内のすべてのラッチの更新を停止する必要がある．さもないと，タイミング・フォルトが発生したラッチの値はそのサイクルの値に戻る一方で，その他のラッチの値は次のサ

イクルの値に更新されてしまうからである．回路の状態は，一貫性を失うことになる．

回路内のあらゆるシャドウ・ラッチからのタイミング・フォルト発生の情報を収集し，全てのラッチへとブロードキャストする**フォルト通知ネットワーク**が必要である．このネットワークは，ハードウェア量も無視できないが，遅延の大きさが特に問題となる．通常の回路では，この遅延は元の回路の数サイクル分程度にはなるであろう．

この遅延が元の回路の n サイクル分 ($n > 1$) になったとすると，以下のようになる：

- タイミング・フォルトが検出されたサイクル内にすべてのラッチの更新を停止するということは，このフォルト通知ネットワークがクリティカル・パスになる．動作周波数は $1/n$（以下）に低下する．
- すべてのラッチの値を n サイクル分のバックアップをとれば，動作周波数は元のまま，n サイクルをかけて停止することができる．しかし，回路内のラッチの総量が n 倍となる．

いずれの場合も，現実的な解法とは言い難い．

2.8.2　アーキテクチャ・レベルの回復技術

プロセッサは通常，レジスタ・ファイル，PC，および，小数の制御レジスタによって定義される**アーキテクチャ・ステート**を持つ．原理的には，アーキテクチャ・ステートだけをフォルトから保護すればよく，回路中のすべてのラッチを保護する必要はない．

さらに，プロセッサは通常，並列に実行される各命令が起こした例外に対して，逐次的なアーキテクチャ・ステートを回復する機能を持つ．タイミング・フォルトを例外の一種ととらえれば，この機能をほぼそのまま利用することができる．

SPARC64 V プロセッサ[10] には，実際そのようなアーキテクチャ・レベルの技術が実装されている．SPARC64 V では，データパス上のすべてのステージにパリティが付加されており，それによってデータパス上に発生したフォルトを検出する．

回復は，以下のように行われる：

1. フォルトが検出されると，直ちに命令のコミットメントが停止される．これによって，誤ったデータによってアーキテクチャ・ステートが更新されるのを防ぐ．

2. 次いで，そのデータを実行結果とする命令（とその前後の命令）をキャンセル/再実行する．

ただし SPARC64 V の方式は，以下の点で十分とは言えない：

1. パリティは，タイミング・フォルト検出には不十分．
2. フォルト検出後，直ちにコミットメントを停止する．
3. 制御系のフォルトを考慮していない．

以下，この 3 点について述べる．

(1) パリティは，タイミング・フォルト検出には不十分

2 章で述べたように，タイミング･フォルトはマルチビットの誤りになりやすい．パリティや ECC では，マルチビットの誤りに対応できない．なお SPARC64 V が主に対象としているのはソフト・エラーであり，タイミング・フォルトはそもそも対象としていないので，このこと自体は SPARC64 V の問題ではない．

(2) フォルト検出後，直ちにコミットメントを停止する

フォルト通知ネットワークは，データ・パス上の各部からフォルト発生の情報を収集し，コミットメント部に伝えるものになる．回路レベルの場合に比べれば格段に小さいとは言え，それでもこのネットワークがクリティカル・パスとなる可能性は残る．

(3) 制御系のフォルトを考慮していない

フォルトは，必ずしもデータ・パス上にのみ発生するものではない．特に out-of-order スーパスカラ・プロセッサの場合，ユニットによっては制御系の占める面積割合やパスの割合が非常に大きく，その部分でのフォルト発生確率は，データパス上のそれよりもむしろ高くなる可能性がある．

SPARC64 V では，制御系には検出回路が挿入されておらず，フォルトの検出がそもそもできない．さらに，たとえ検出回路が挿入されていたとしても，制御系に生じたフォルトからの回復は，データパス系に生じたフォルトからの回復に比べて，格段に難しい．

制御系のフォルトの検出/回復の必要性について次の 2.8.3 節で，制御系に生じたフォルトの回復の難しさについて 2.8.4 節でさらに詳しく述べる．

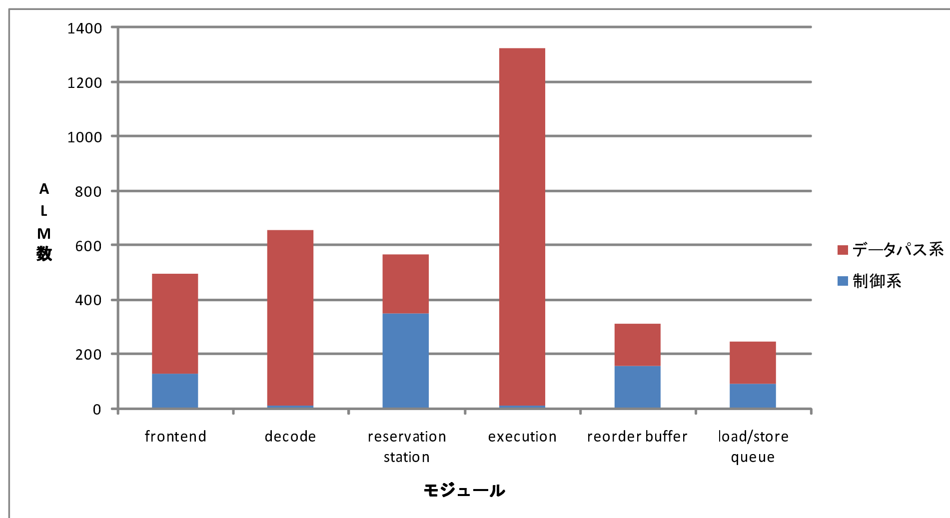

図 2.5: 制御系とデータパス系の面積

2.8.3　制御系のフォルトからの回復の必要性

詳しくは 4 章で述べるが，研究を行う上で，FPGA 上に out-of-order スーパスカラ・プロセッサを実装を行った．

このプロセッサの，それぞれのモジュールにおいて，制御系とデータパス系それぞれの面積の大きさを ALM 数であらわしたグラフを 2.5 に，制御とデータパスの面積の割合であらわしたグラフを図 2.6 に示す．リザベーション・ステーション，リオーダ・バッファ，ロード/ストア・キュー，の 3 つの項目では，SRAM で構成されるそれら自身 (リザベーション・ステーションの項目ならリザベーション・ステーション自身) の面積は含まれておらず，それらを構成するために必要な周りの回路のために必要な ALM の数を示している．また，命令キャッシュの制御回路は frontend の項目に，データキャッシュの制御回路は load/store queue の項目に，バイパス・ネットワークは execution の項目にそれぞれ含めてある．なお，ALM とは，今回使用した FPGA である，Altera 社の Stratix II で，ルック・アップ・テーブルと FF を構成する，FPGA の最小構成単位である．

この図より，リザベーション・ステーション，リオーダ・バッファ，ロード/ストア・キューといった，命令の実行順序などを制御するためのユニットでは，制御系の割合が 40-60% と大きいが，バイパスネットワークを含めた実行ユニット周りは面積が大きく，そのほとんどがデータパス系であるため，制御系の面積は全体の約 20% にとどまっている．

　タイミング・フォルトの発生は，ソフト・エラーと違って，回路面積のみ

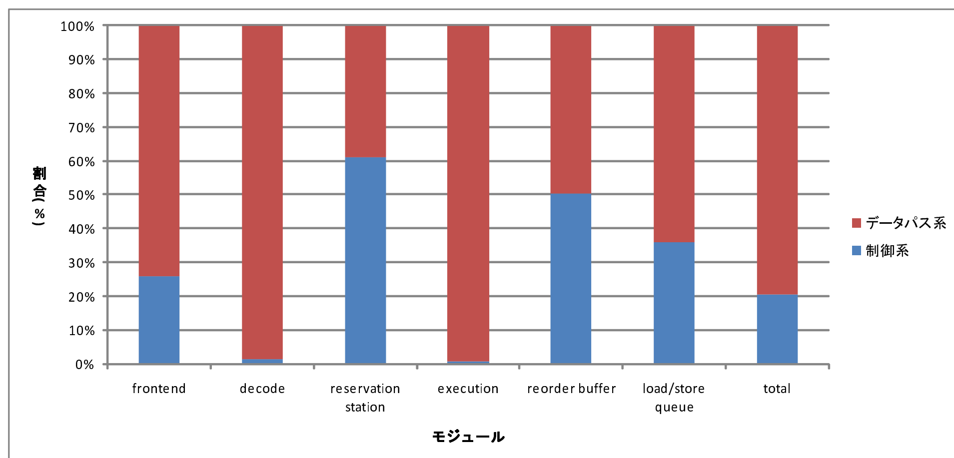

図 2.6: 制御系とデータパス系の面積の割合

ではなく，パスごとの遅延の分布にも依存する．図 2.7 に，実装したプロセッサのパスの長さの分布のグラフを示す．この図は，プロセッサ中に含まれるすべての FF に対して，それぞれの FF のクリティカルパスの遅延を横軸に，それぞれの遅延のパスの数を縦軸にとったものである．データパス系と制御系で分けてずに示した．また，データパス系と制御系との相互のデータのやり取りのパスは，すべて制御系として分類している．この結果を，遅延ごとにデータパス系と制御系との割合であらわしたグラフを，図 2.8 に示す．なお，これらの遅延の計算は，今回用いた利用した Altera 社の FPGA の開発環境である，Quartus II に含まれる TimeQuest Timing Analyzer によって行った．

これらの図よりわかるように，回路中に含まれるパスのうち，制御系のパスは全体の 4 割程度と多く，また，クリティカル・パスにも制御系のパスが 4 割程度含まる．これより，out-of-order スーパスカラ・プロセッサでは，データパス系のみならず制御系に対するタイミング・フォルト対策も必須であると言える．

また，最も長いクリティカル・パスは，wakeup/select ステージのパスであり，遅延は 13.805ns であった．このことからも，制御系に対するタイミング・フォルト対策の必要性が伺える．

2.8.4 制御系のフォルトからの回復の困難

制御系にフォルトが生じていないなら，データ・パス上にフォルトが生じても，プロセッサは「正気」である．たとえば 64bit のデータのどのビットがビッ

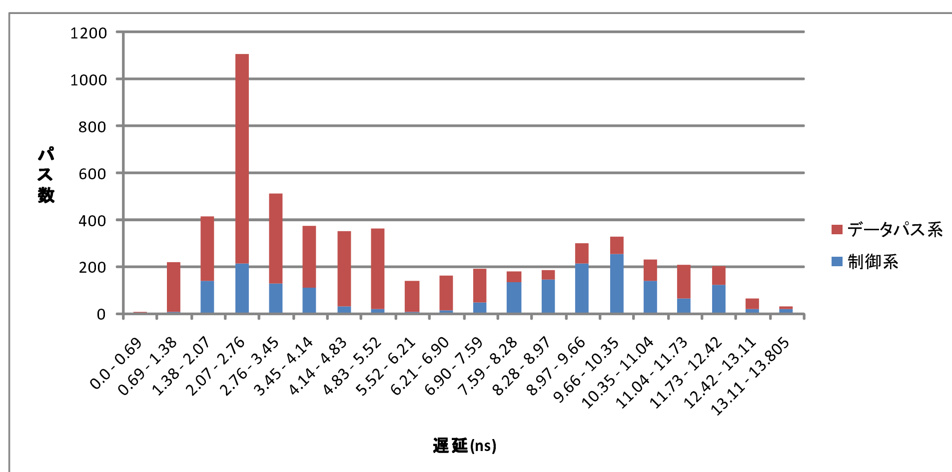

図 2.7: 遅延の分布

ト誤りを起こしたとしても，設計者が「その値になるとは想定していなかった」などということはない．

したがって，そこからの回復は容易である．たとえばSPARC64 Vのように，命令を再実行することで回復可能である．

それに対して，制御系にフォルトが発生した場合には，プロセッサは，もはや「正気」ではない，すなわち，設計時にはまったく想定されていなかった未知の状態に遷移することになる．そこからは，SPARC64 Vのような方法では回復することはできない．

分かりやすい例としては，FIFOのリード/ライト・ポインタとカウンタなどが挙げられる．Out-of-orderスーパスカラ・プロセッサでは，このようなFIFOは，さまざまなバッファに用いられている．フォルトによってこれらのポインタとカウンタの値に齟齬が生じた場合，アンダランやオーバランなどが発生することになる．

たとえ命令を再実行したところで，このような状態から回復することはできない．そもそも，どの命令を再実行すればよいかすら判然としないし，命令の再実行自体を正しく行える保証もない．

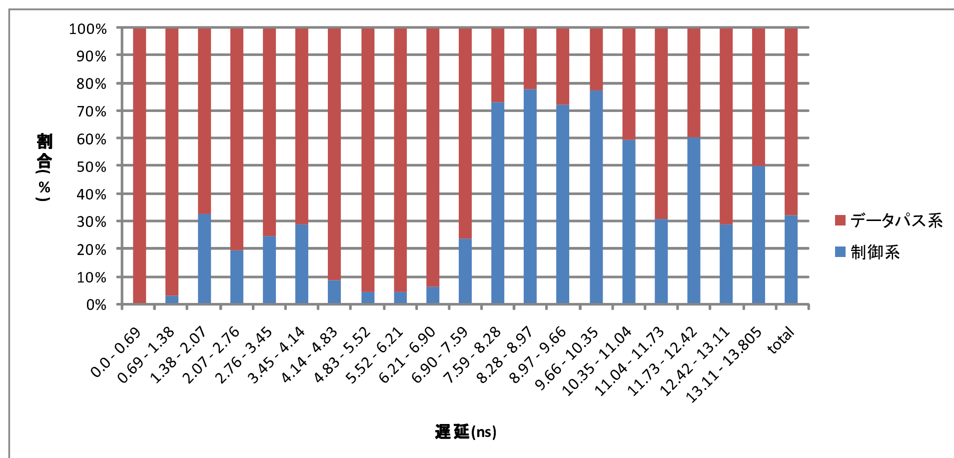

図 2.8: 遅延の分布のデータパス系と制御系との割合

第3章　提案手法

SPARC64 V の問題としては，以下の 3 点があった：

1. パリティは，タイミング・フォルト検出には不十分.
2. フォルト検出後，直ちにコミットメントを停止する.
3. 制御系のフォルトを考慮していない.

提案手法では，以下の 3 点を用いて，これらの問題に対処する：

1. RAZOR もしくはカナリア FF 等を利用
2. 回路レベルの，パイプライン化されたフォルト通知ネットワーク
3. リセットによる回復

次節からは，2 点目と 3 点目について詳しく述べる.

3.1　フォルト通知ネットワーク

SPARC64 V のフォルト通知ネットワークを図 3.1 に，提案手法のフォルト通知ネットワークを図 3.2 示す．なお，図 3.2 の左上 4 つの FF は RAZOR やカナリア FF などのタイミング・フォルト耐性を持つ FF を表している.

詳しくは後に述べるが，提案手法では，SPARC64 V と同様に，フォルトが検出されると，命令コミットを停止することによってアーキテクチャ・ステートが更新されるのを防ぐ．言い換えれば，アーキテクチャ・ステート以外にフォルトを含んだ情報が伝播されることは，正しく回復が行われる限り，問題とならない．この条件下でフォルト通知ネットワークに対する必要な条件は，次の 4 つである.

1. すべてのフォルトの情報が正しくコミット・ステージに伝わる

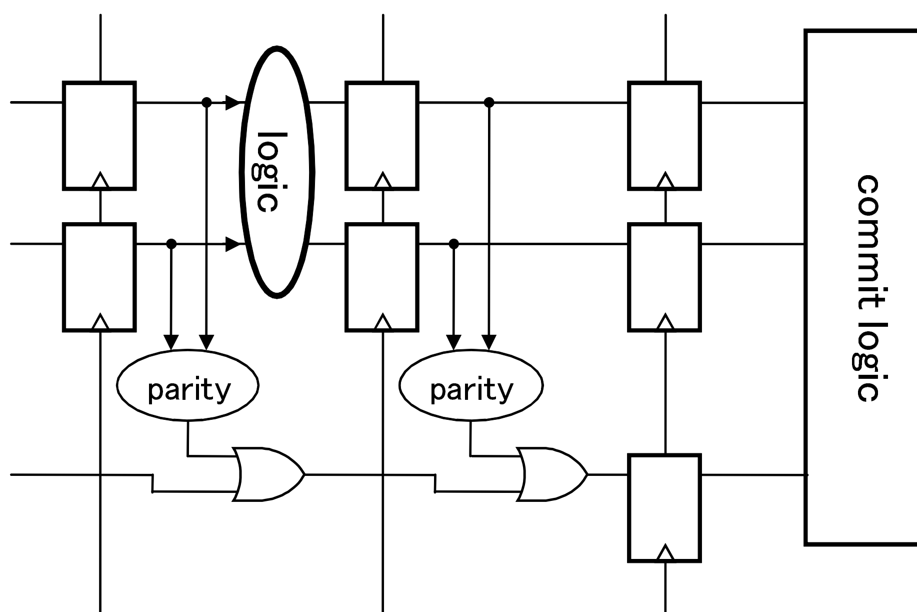

図 3.1: SPARC64 V のフォルト通知ネットワーク

2. フォルトを含むデータよりも先に，フォルトの情報がコミット・ステージに伝わる

3. 制御系に対して適用可能

4. クリティカル・パスにならない

1 つ目と 2 つ目の条件は，上記の条件で，フォルト通知ネットワークが最低限満たすべき条件である．3 つ目の条件を満たすことにより，提案手法では制御系を含めたフォルトからの回復を可能にする．4 つ目の条件は，フォルト通知ネットワークが性能へ影響を与えないために必要な条件である．SPARC64 V のフォルト通知ネットワークは，3 つ目の条件を満たさず，4 つ目の条件も満たさない可能性が残っているのであった．

提案手法のフォルト通知ネットワークは次の 2 つの特徴を備えることによって，これらの条件を満たす．

1. 回路レベルである

2. パイプライン化されている

以下，2 つの特徴について説明する．

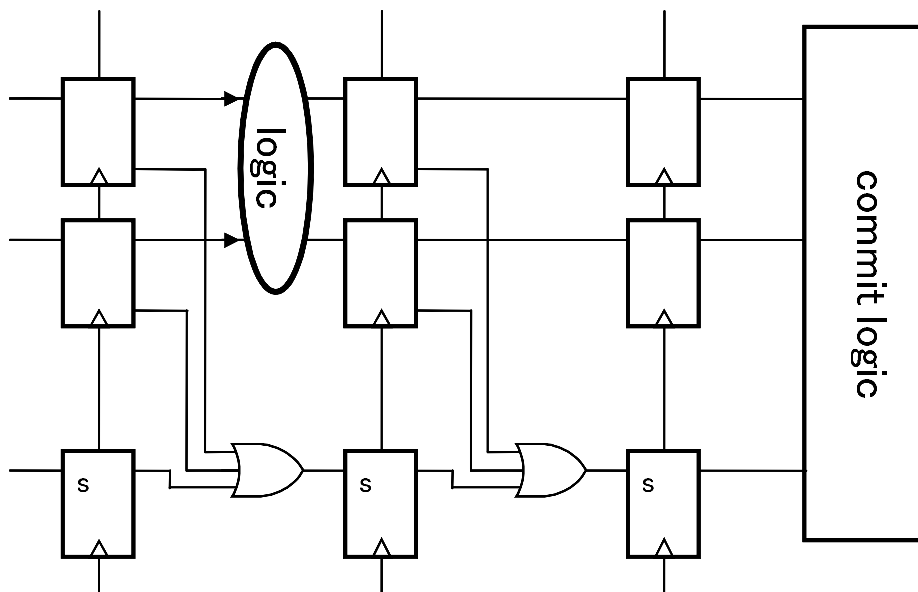

図 3.2: パイプライン化されたフォルト通知ネットワーク

なお，SPARC64 V，提案手法ともに，フォルト通知ネットワークによってフォルトがコミット・ステージまで，通知されるのが，フォルトを起こした命令の到着より早い事があるが，これは問題ない．コミット・ステージでフォルトが検出された場合，SPARC64 V の場合は命令再フェッチ，提案手法では後に述べるようにリセットによって，命令の再実行を行う．もし，insn0,insn1 の順番で命令が実行され，insn1 がフォルトを起こした場合に，コミット・ステージへのフォルトの通知が早く，コミット・ステージで，insn0 がフォルトを起こしたと判断されたとしても，命令が insn0 から再実行されるだけである．フォルトの発生確率が十分低ければ，性能与える影響も無視できる程度となる．

3.1.1　回路レベルのフォルト通知ネットワーク

提案手法では，すべての FF を RAZOR やカナリア FF に置き換えることによって，フォルト耐性を持たせ，フォルトをすべて or した結果をコミット・ステージに通知するように構成されている．これは，フォルト通知ネットワークは回路レベルであり，図 3.2 のように，データパス系 や制御系の差異にかかわらずに構成することができる事を意味する．これにより，制御系に対しても適用可能という条件が満たされる．

一方，SPARC64 V の場合では，図 3.1 のように，アーキテクチャ・レベルの操作である，parity を取るという操作を行っている．parity を取ることができ

るのはデータパス系に対してのみであるので，制御系にも適用可能という条件は満たされない．

3.1.2 パイプライン化

図 3.2 に示すように，提案手法のフォルト通知ネットワークはパイプライン化され，フォルトは 1 clock につき 1 段ずつ，コミット・ステージへ向けて通知される．

パイプライン化することによって，

1. or の入力数の総数を減らす
2. 1 clock あたりの配線遅延が短くなる

という，2 つの効果が得られる．

パイプライン化により，各段の or の入力数は数百程度になる．これを 4 入力の or で構成すると，5 段以内で構成できる．スーパスカラ・プロセッサのクリティカル・パスは，10-15 段程度あるので，1 clock あたりの配線が短く抑えられることも伏せて考えれば，フォルト通知ネットワークがクリティカル・パスの遅延はプロセッサのクリティカル・パスの遅延の半分以下になる．よって，フォルト通知ネットワークがプロセッサのクリティカル・パスとなる確率は，十分低い．FPGA に実装したプロセッサでの，フォルト通知ネットワークの遅延の測定結果は，評価の章で述べる．

パイプライン化による問題として，SPARC64 V では，コミット・ステージにすぐにフォルトが届くのに対して，提案手法では，フォルト通知ネットワークのパイプライン化によって，コミット・ステージへフォルトが到着するまでにより多くの clock 数がかかるようになる，ということがあげられる．しかし，パイプライン化してもフォルトを含む結果がコミット・ステージへ，フォルトよりも早く到着することがないように設計できる為，問題とならない．このことは以下のように考えれば，簡単に理解できる．例えば，コミット・ステージ到達まで最短であと n clock 必要なステージで，フォルトが発生したとする．このとき，提案手法のフォルト通知ネットワークでは，フォルトは n clock 後にコミット・ステージに到着するようにできる．言い換えれば，n clock 後にコミット・ステージに情報を伝える FF に対して，フォルトの情報を書き込むように設計する事が，容易にできる．よって，遅くとも，フォルトを含む結果と同時にフォルトはコミット・ステージステージに到着させる事が可能である．

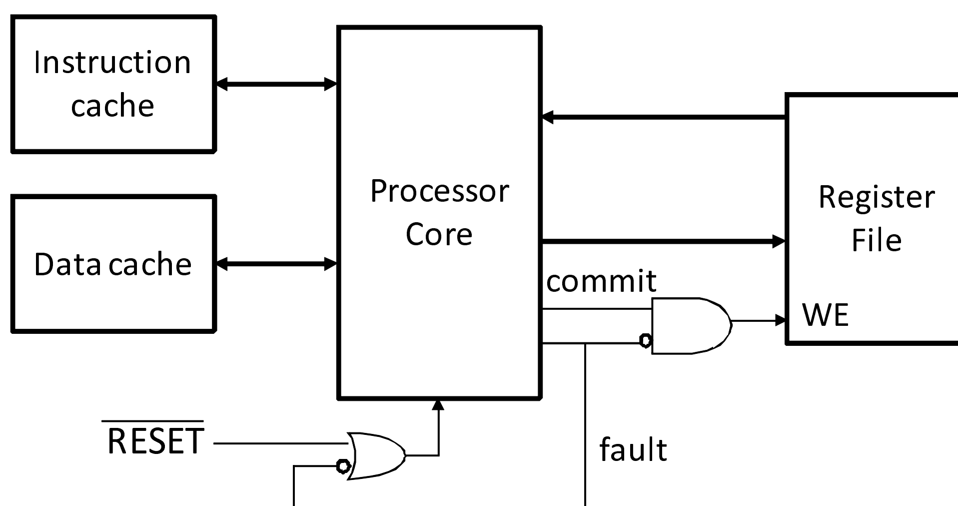

図 3.3: リセット

3.2 リセット

3.2.1 リセットによる回復

提案手法では，SPARC64 V と同様に，フォルトが検出されると，命令コミットを停止することによってアーキテクチャ・ステートが更新されるのを防ぐ．SPARC64 V ではその後，そのデータを実行結果とする命令（とその前後の命令）をキャンセル/再実行する事によって回復を図るが，この方法では制御系のフォルトに対処することができないのであった．

そこで，提案手法では，フォルトが検出された後に，プロセッサをリセットすることによって，制御系を含めたフォルトからの回復を可能にする．その様子を図 3.3 に示す．

リセットを行うことにより，制御系を含めた全てのレジスタが初期状態となる．つまり，リセットを行った後は，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，一貫した状態に回復する事ができる．

このようなことができるためには，アーキテクチャ・ステート，すなわちレジスタ・ファイルと PC にタイミング・フォルトが発生しないという条件が必要である．この条件は，レジスタ・ファイル及び PC の更新，十分に時間をかけて行っても，性能等への影響はないため，満たすことができる．PC はレジスタ・ファイルの 1 エントリと考えても差し支えない為，以下では，レジスタ・ファイルについて，書き込みをタイミング・フォルトが発生しないだけの十分な時間をかけて行える理由を説明する．

レジスタ・ファイルの値が利用されるのは，タイミング・フォルト発生時と分岐予測ミス等のミスによりパイプラインフラッシュが発生したときのみである．タイミング・フォルト発生時は，プロセッサコアをリセットするための時間 (〜数百サイクル) が必要であり，この時間は，レジスタ・ファイルの書き込みがタイミング・フォルトを起こさないだけの十分な時間がある．パイプラインフラッシュが発生した場合も同じように，コミット・ステージで書き込まれた値が次にレジスタ・ファイルから読みだされるまでには時間がある．最低でも，パイプラインフラッシュに必要なクロック数と，フェッチに必要なクロック数を合わせただけのクロック数をかけて，レジスタ・ファイルへの書き込みを行っても問題ない．このクロック数は，レジスタ・ファイルへの書き込みがタイミング・フォルトを起こさないようにするのに，十分である．

3.2.2 リセットのレベル

このように，プロセッサをリセットすることによって，out-of-order スーパスカラ・プロセッサの回復が可能となる．本章では，このリセットについて，さらに詳しく述べる．

リセットは，どの範囲について行うか，というレベルがある．提案手法の中で，リセットしなければならない範囲は小さい．これは，以下の 3 種類に分類できる．

- 主に FIFO で構成される，リザベーション・ステーションやリオーダ・バッファ等の SRAM で構成されるテーブルのエントリが有効であることを示すビット
- データの値が有効であることを示すビット

  FIFO への，ライト・イネーブル線など
- その他の制御系のレジスタ

  カウンタや FIFO のリード/ライトポインタ，ステートマシンのステートなど

逆にいえば，これらさえ正しくリセットされれば，out-of-order スーパスカラ・プロセッサを正しく回復できる．これらに含まれない，データパスの FF

が保持するデータは，リセットする必要はない．データパス中の FF が保持するデータは，上記の 3 種類のいずれかが有効となった時，同時に別の値が書き込まれるからである．言い換えれば，データパス中の FF が保持する値は，常に上記の値とセットになっていると言える．

現状，プロセッサには，power on reset や，warm reset が実装されているが，これらのリセットは，プロセッサのすべてをリセットするため，リセットに非常に長い時間を要する．なので，タイミング・フォルトからの回復に，これらのリセットをそのまま用いると，大きく性能が低下する．逆に，上記にあげた範囲のみをリセットするための配線を引くことは，回路面積が増大するため，良い手法と言えない．

そこで，最低限上記の部分をタイミング・フォルトが発生したときにもリセットが行われる様に，既存のリセット回路を変更するという方法が考えられる．上記 3 種類の部分の近くにある FF は，一緒にリセットをかけてしまうほうが効率が良いのであれば，そうしてしまうように回路を組めば良い．そのようにすれば，面積も小さく，リセットの範囲を抑えることで，リセットに必要なクロック数も抑えることができる．

また，BTB，PHT のような予測テーブルのように，リセットをしない方が良いユニットがあることにも注意したい．例えば，BTB や PHT の値が間違っていたとしても，これらの値は分岐予測に利用されるだけなので，分岐予測ミスが発生するだけである．つまり，実行結果に影響は及ばない．逆に，リセットをかけてしまうと，予測の的中率の低下を招く．さらに，BTB や PHT といったそれなりに容量のあるテーブルは，リセットをするのに時間がかかる．これらのことより，PHT や BTB といった予測テーブルは，リセットせずに，そのまま放置しておけばよいとわかる．

具体的にどのような配線を行えば良いのか，その場合，リセットに必要な遅延はどのくらいの時間となるのか，といったことは，今後考えてゆく必要がある．

3.3 議論

最後に，本節では，制御系を含めたプロセッサのあらゆる部分で発生したタイミング・フォルトからの回復方法の可能性について考える．

前節で述べたように，制御系にフォルトが発生した場合には，プロセッサは，もはや「正気」ではない，すなわち，設計時にはまったく想定されていなかった未知の状態に遷移する．そこから既知の状態に遷移させるためには，全てのレジスタの既知の状態のチェックポイントを取っておく，もしくはアーキテク

チャ・ステート＋初期状態に戻す，の２つしかないであろう．チェックポイントを取る手法は，前節で述べた回路レベルの回復技術と同等であり，現実的でない．その為，アーキテクチャ・ステート＋その他のレジスタの初期状態に戻す，という方法が，未知の状態から既知の状態へと遷移させる，ほとんど唯一の解であると考えられる．

第4章　実装

4.1　実装の意味

現在のプロセッサ研究では，シミュレーションによって性能評価を行うことが一般的である．IPC の評価には，SimpleScalar などが，回路遅延の評価には，CACTI，hspice などが，消費電力の評価のためには WATTCH などが，それぞれ用いられる．

しかし，シミュレーションは，今回の研究の目的である，動的タイミング・フォルトに関する技術の最も重要な部分を評価するためには，役に立たない．このような技術を評価するためには，回復機構を含む，プロセッサのいかなる部分にフォルトが発生しても対処可能であることを示す必要がある．しかし，シミュレーションによってそれを示すことは不可能である．

そこで，FPGA 上に out-of-order スーパスカラ・プロセッサを実装し，さらにそのプロセッサに，提案手法を実装した．また，提案手法の有効性を確かめる為に，動作周波数・電源電圧を任意に変化させる事が可能なボードを用意した．このボード上の FPGA に out-of-order スーパスカラ・プロセッサを実装し，動作周波数・電源電圧を変化させることによってわざとタイミング・フォルトを発生させて，提案手法の妥当性を確かめることができる．

今回，このように実装することによって提案手法の妥当性の確認したわけであるが，次のことに留意しておくことが必要である．すなわち，タイミング・フォルト対策技術の妥当性は，動作することの確認が取れた回路に対してのみ言うことができる．例えば，A という回路にタイミング・フォルト対策技術を実装して，動作確認が取れたからといって，その技術が必ずしも B という別の回路で正しく動作すことを，保証できない．つまり，今回実装した out-of-order スーパスカラ・プロセッサで提案手法が正しく動作したとしても，他の構成の out-of-order スーパスカラ・プロセッサでは正しく動作することを保証できない．しかし，ある一つの構成の out-of-order スーパスカラ・プロセッサで提案手法の動作証明をすることは，他の回路に適用可能か検証する際のモデルケースとして有用である．また，実装したプロセッサでの提案手法の動作が確認できれば，少なくとも今回実装した out-of-order スーパスカラ・プロセッサが備え

図 4.1: 動作周波数・電源電圧可変 FPGA ボード

る機能ユニットにおいては，提案手法は有効であると期待することができる．

以下，本章では，out-of-order スーパスカラ・プロセッサの FPGA への実装に関する事について述べる．

4.2 ボード

先に述べたように，搭載された FPGA の動作周波数・電源電圧を任意に変化させることができるボードを用意した．写真を図 4.1 に示す．

動作周波数･電源電圧はパソコンより手動で変更することが可能である．FPGA として，Altera 社の Stratix II が使用されている．

動作周波数制御は，Analog Devices 社の AD9851 によって，電源電圧制御は，MAXIM 社の MAX1855 によって行われている．パソコンとボードとのインターフェースは USB となっている．

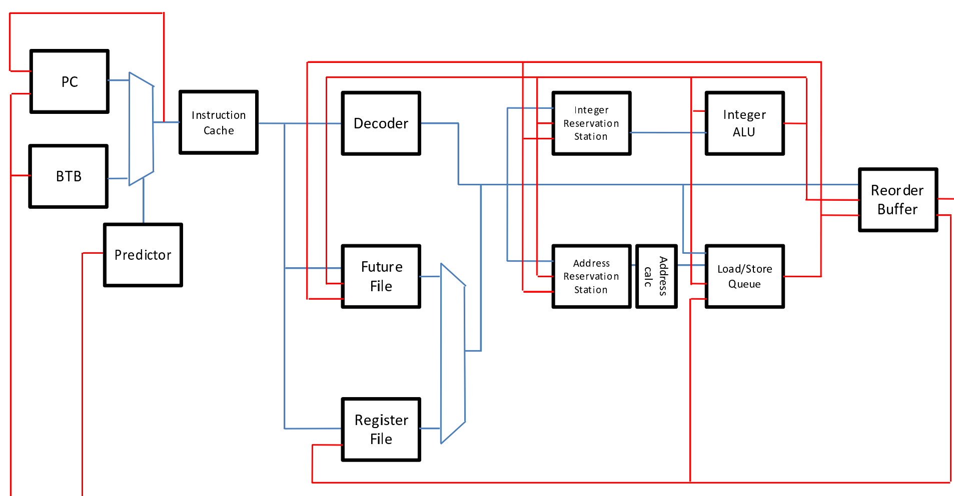

図 4.2: 実装した out-of-order スーパスカラ・プロセッサの全体図

4.3　FPGA 及び開発環境

FPGA としては Altera 社の Stratix II を採用した．そして，開発環境としても同じく Altera 社の Quartus II を用いた．記述は，Verilog HDL により行った．

4.4　命令セット及びビット幅

命令セットとして，Alpha 21264[4] を採用した．また，ビット幅は 32 とした．

実装した命令は，整数演算（足し算，引き算のみ），ロード/ストア，条件分岐/無条件分岐，比較である．掛け算と割り算，浮動小数点演算等は実装していない．

4.5　全体構成

Out-of-order スーパスカラ・プロセッサの実装方法には大きく分けて，リオーダ・バッファとリザベーション・ステーションを併用する方法とレジスタ・リネーミングによる方法がある．

レジスタ・リネーミングによる方法では，RMT(Register Map Table) が必要である．RMT はチェックポイントを取る必要があるが，これは通常，特殊な構造の SRAM を用いて実装される．しかし，当然だが FPGA にそのような SRAM は搭載されていない．また，RMT は 4-way の out-of-order スーパスカラ・プロ

セッサでは通常 12-read/4-write ポートであり，ポート数が非常に多い．FPGA には通常，1-read/1-write ポートの SRAM しか搭載されておらず，RMT を FPGA 上に実装するのは難しい．

そこで今回は，リオーダ・バッファとリザベーション・ステーションを併用する方式を採用した．ただし，これでは，本節冒頭で述べたように，レジスタ・リネーミングによる方式における，提案手法の正しさを示す事はできない，という点に注意が必要である．

このような構成で実装した out-of-order スーパスカラ・プロセッサの全体構成を図 4.2 に示す．

メモリのポート数を減らすために， フェッチ幅は 2 とした．そして，gshare により実装された，分岐予測器を搭載した．

図 4.2 にある，Integer Reservation Station のエントリ数は 8，Address Reservation Station のエントリ数は 4，Load/Store Queue のエントリ数は 8，Reorder Buffer のエントリ数は 16，レジスタファイルおよびフューチャ・ファイルの本数は 32 とした．また，命令キャッシュ及びデータキャッシュの容量は 0.5Kbyte，BTB の容量は 0.25Kbyte，PHT の容量は 128byte とした．

4.6 パイプライン

fetch，register-read(decode)，dispatch，schedule，issue，execute，write-back，commit の全 8 段から成るパイプライン化を施した．

fetch はポート数の少ないメモリより値を読み出すのに対して，register-read や dispatch，issue などでは多ポートのメモリの読み書きを行う．FPGA では，多ポートのメモリの実装は厳しいので，通常のプロセッサと比べて，fetch にかかる時間は，register-read，dispatch，issue などにかかる時間と比べて相対的に短くなっている．

4.7 提案手法の実装

前節で述べた out-of-order スーパスカラ・プロセッサに対して，提案手法の実装を行った．タイミング・フォルト検出手法としてカナリア FF を用いた．カナリア FF の実装には遅延が必要であるが，それは，インバータを偶数段挿入することにより実現した．

すべての FF をカナリア FF に変更し，フォルトはフォルト通知ネットワークを通じてコミット・ステージ，すなわちリオーダ・バッファに通知されるよう

に実装した．フォルトが発生した場合，コミット・ステージすなわちリオーダ・バッファにその情報が通知され，アーキテクチャ・ステート (レジスタ・ファイルと PC)，キャッシュ，BTB，PHT を除くすべての場所がリセットされる．リセット完了後，プロセッサの実行は再開される．

キャッシュのタイミング・フォルト対策は必要であるが，現在は実装していない．今回実装したプロセッサでは，キャッシュの読み書きはクリティカル・パスではなかったので，手法の妥当性を確かめる実験で，影響が現れることはない考えられる．

なお，実装したソースコードは，7000 行程度となった．

第5章　評価

5.1　提案手法の有効性の確認

実装したプロセッサが正しくタイミング・フォルトを検出/回復を行えるか，その有効性の確認を行った．

そのために，図5.1に示すような実験装置を組んだ．そして，電源電圧，動作周波数，温度の3つのパラメータをそれぞれ独立に変化させ，正しくタイミング・フォルトを検出/回復できるか確認を行った．

図5.1にあるように，電源電圧及び動作周波数は，パソコンより制御した．また，温度はFPGAにドライヤーをあてることによって変化させた．ボード上のFPGAにタイミング・フォルトを持つout-of-orderスーパスカラ・プロセッサが書き込まれている．そして，プロセッサの命令メモリにはあらかじめプログラムが書き込まれており，電源電圧及び周波数を適切に設定することによって，プロセッサはプログラムの実行を開始するようにした．

周波数及び電源電圧に対する耐性の確認

まず，この装置において，パソコンより周波数及び電源電圧を動的に変更した場合に，以下ようになることを確認した．

- 電源電圧を一定とし，周波数を徐々に上げる，もしくは周波数を一定とし，電源電圧を徐々に下げると，タイミング・フォルトが発生し，そのタイミング・フォルトをプロセッサが正しく検出する．
- タイミング・フォルトを検出したら，周波数，もしくは電源電圧を元の状態に戻す．そうすると，プロセッサは正しくプログラムの実行を再開する．

これより周波数及び電源電圧によって遅延が変化した場合，タイミング・フォルトを正しく検出/回復できることが確認された．

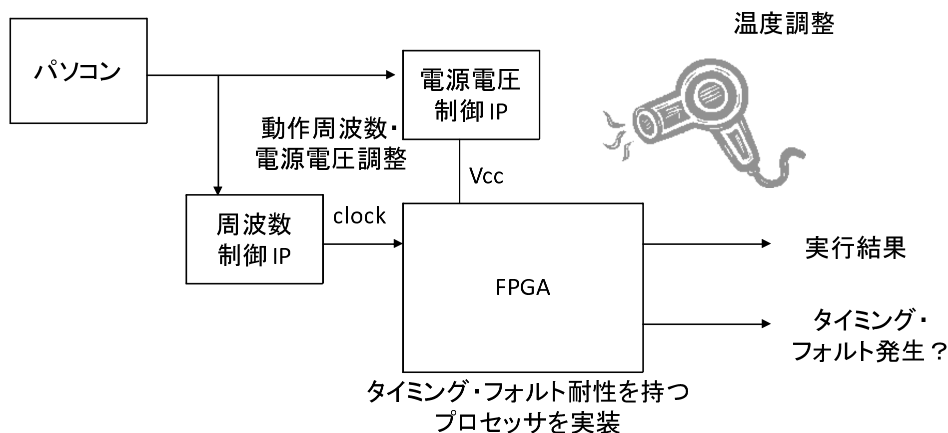

図 5.1: 実験装置

温度に対する耐性の確認

次に，スーパスカラ・プロセッサが搭載されている FPGA にドライヤーをあてることによって，温度を変化させた場合に，以下のようになることを確認した．

- FPGA をドライヤーで熱風を吹きかけることによって熱すると，タイミング・フォルトが発生し，タイミング・フォルトを正しく検出する．
- タイミング・フォルトが検出されたら，今度はドライヤーで冷風を吹きかけることによって FPGA を冷やす．そうすると，プロセッサは正しく実行を再開する．

これより，温度が変化することによって遅延が変化した場合にも，タイミング・フォルトを正しく検出/回復できることが確認された．

また，電源電圧・動作周波数の組み合わせによっては，その組み合わせを変更しないままで，タイミング・フォルトの検出・回復が繰り返されるという現象が確認された．原因は，熱などの外部環境の影響によるものと考えられる．

5.2 フォルト通知ネットワーク

本節では，フォルト通知ネットワークについての評価を行う．

表 5.1: パイプラインの各段における FNN

ステージ	入力数	4 入力 or の必要な個数	遅延 (実装上)
Fetch	76	25	4.538
Decode	68	23	7.645
Dispatch	168	56	8.294
Schedule(integer)	5	2	4.767
Issue(integer)	194	65	6.499
Execution(integer)	321	107	7.394
Writeback(integer)	75	25	8.231
Schedule(address)	2	1	7.435
Issue(address)	70	23	7.481
Execution(address)	105	35	4.254
Load/StoreQueue write	38	13	6.859
Writeback(load/store)	138	41	4.524
Commit	225	75	6.494

既に述べたように，フォルト通知ネットワークは，パイプライン各段において，多入力の or によって構成される．各段において，何入力の or が必要であったか，各段を 4 入力の or で構成した場合，それが計算上いくつ必要であるのか，そして FPGA における遅延がどの程度であったのかを，表 5.1 に示す．FPGA における遅延の項目は，各段のクリティカル・パスを Quartus II に付属する TimeQuest Timing Analyzer を用いて算出した結果を記した．

5.2.1　クリティカル・パス

表 5.1 を見てもわかるとおり， or の入力数の最大は，実行ユニットにおける 321 入力であった．これは 4 入力の or で構成したとすると，5 段で構成することができる．よってフォルト通知ネットワークのクリティカルパスは，4 入力 or 5 段＋配線遅延となる．既に述べたように，昨今のプロセッサにおけるクリティカルパスはゲート数 10 - 15 段程度分あるので，フォルト通知ネットワークがクリティカルパスとなる可能性は十分低いといえる．

次に，FPGA 上に実装したフォルト通知ネットワークにおける遅延と，そのプロセッサのクリティカル・パスの比較した結果を述べる．2.8.3 節で述べたよ

うに，実装したプロセッサのクリティカルパスはwakeup/selectステージのパスであり，遅延は13.805nsであった．そして，フォルト通知ネットワークのクリティカル・パスは表5.1を見ればわかるとおり，dispatchステージの8.294nsであった．フォルト通知ネットワークはクリティカル・パスの遅延は，プロセッサのクリティカル・パスの60%程度であり，クリティカル・パスとなる確率は低いといえる．

5.2.2　ハードウェア量

フォルト通知ネットワークは，他入力のorとFFのみによって構成される．一般的に，orの入力数の最大値は4であるので，フォルト通知ネットワークを構成するために必要なorを，すべて4入力orで構成した場合，それがいくつ必要であったかを計算した．結果は，表5.1に示した．合計で，561個必要であった．フォルト通知ネットワークのパイプラインレジスタは，ステージ間にひとつ必要なので，合計13個必要となる．よって，フォルト通知ネットワークを実装するために必要なハードウェア量は，600ゲート相当もないとわかる．

昨今のプロセッサが数千万ゲート規模であることを考えれば，フォルト通知ネットワークのハードウェア量は0.01%にも満たず，十分小さいといえる．

また，フォルト通知ネットワークが実装されている場合とフォルト通知ネットワークが実装されていない場合それぞれの場合のプロセッサをQuartus IIにてコンパイルを行い，必要なFPGA上のハードウェア量について比較を行った．フォルト通知ネットワークを実装するにあたって，すべてのFFをカナリアFFに置換した．よって，比較した両者の違いは，FFがカナリアFFかそうでないかと，フォルト通知ネットワークがあるかないか，の2点である．カナリアFFに，遅延は挿入しなかった．FFを1つカナリアFFに置換すると，FFひとつ分のコストが余計かかっていたこととなる．置換したFFの数は全部で1964個であった．

測定の結果，フォルト通知ネットワークが実装されている場合に必要なハードウェア量は15670ALMであり，実装されていない場合は，14122ALMであった．フォルト通知ネットワーク実装に必要であったALM数は，他のモジュールとの最適化の兼ね合いもあるが，大体$15670 - 14122 = 1548$ALMであることがわかる．

ALMは，今回使用したFPGAであるStratix IIの，ルックアップテーブルとフリップフロップを構成する最小単位であり，AUT1つで，ルックアップテーブル2つとフリップフロップ2つが構成可能である．

理論上必要なALMの個数を計算すると，ALMひとつ用いて4入力のorは2

つ構成可能なので，フォルト通知ネットワークの or の部分を構成するために必要な ALM は大体 $561/2 = 281$ 個，フォルト通知ネットワークのパイプラインレジスタのために必要な FF は，$13/2 = 7$ 個，FF をカナリア FF に置換するために必要な ALM は大体 $1964/2 = 982$ 個となり，合計で，$281+7+982 = 1270$ 個となる．

実際に合成した場合の方が，理論上の値よりも，ALM$1548 - 1270 = 278$ 個分大きい．これは恐らく，性能上をより上げるために，コンパイラが ALM に含まれる 2 つの FF の片方のみ使用するようにマッピングを行っているなどの事などが原因であろう．いずれにせよ，FPGA に実装した場合も，フォルト通知ネットワークの大きさは理論上の大きさと同じかそれよりも若干大きい程度ということが言える．理論上のフォルト通知ネットワークのハードウェア量が大変少ないので，FPGA での実装上においても，フォルト通知ネットワークのハードウェア量が少ないと言える．

第6章　関連研究

RAZORとカナリアFFはともに，回路レベルでFFのタイミング･フォルトを検出する技術であった．本章では，アーキテクチャレベルのフォルト検出/回復手法として，DIVAを，また，本文でも若干ふれたが，我々の提案するSRAMの書き込みのタイミング･フォルトを検出する手法として書き込み保証バッファを取り上げ，それぞれの特徴を述べる．

Austinらが提案した**DIVA**アーキテクチャ[1, 9]では，プロセッサの実行コア全体に対する検証用ロジックとしてチェッカー・コアを用意し，それぞれ同じ命令を実行することで，プロセッサに生じる様々なフォルトを検出する．チェッカー・コアは，ちょうど値予測を行うプロセッサにおける検証ユニットのように，実行コアの実行結果を検証するだけでよい．チェッカー・コアが正しい限り，メイン・コアに発生するフォルトはすべてチェッカー・コアで検出/回復可能である．そして，チェッカー・コアはメイン・コアよりはるかに小規模なので，フォルトが発生する確率はほとんどないと言える．しかし，チェッカー・コアが必要なことによるハードウェア量の増加や，デザインの手間が増えるなどの問題がある．

また，我々は，レジスタ・ファイルへの書き込み時に発生するタイミング・フォルトを検出する手法に，**書き込み保証・バッファ**という手法を提案している．この手法では，レジスタ・ファイルへ値を書き込むときに，それと並行して, 書き込み保証・バッファにも書き込みを行う. その後, 書き込んだ値をレジスタから再び読み出し, 書き込み保証・バッファに保持されている値と一致比較して, 書き込みを検証する．検証の済んだ値はWABから削除される．値が食い違っていた場合に，タイミング・フォルトが発生したと考える．ポイントは，書き込み保証・バッファはレジスタ・ファイルに比べて小容量でよいということである．遅延はエントリ数に依存する為，書き込み保証・バッファはレジスタ・ファイルよりもタイミング･フォルト耐性が高い．よって，レジスタ・ファイルに書き込まれたデータがフォルトを起こしていたとしても，書き込み保証・バッファに書き込まれたデータはフォルトを起こしていないことが期待できる．3.2.2節でも述べたように，キャッシュに対しては書き込み保証・バッファの利用を現在考えている．

第7章　おわりに

本稿では，動的なタイミング・フォルトがout-of-orderスーパスカラ・プロセッサの制御系を含めたあらゆる箇所で発生しても，正しく検出/回復できる手法を提案した．そして，提案手法を，プロセッサ上に実装し，そのプロセッサに対してわざとタイミング・フォルトを発生させても正しくフォルトを検出/回復できることを確認した．また，提案手法を実装するためには，フォルト通知ネットワークを実装することになるが，そのコストが非常に安いことも確認した．

今後の課題としてはまず，リセットのさらなる評価があげられる．リセットの遅延はどの程度なのか，ハードウェア量の増加はどの程度になるのか，プロセッサの他の機構に影響を及ぼさないのか，などが評価項目として考えられる．

次に，DVFSと協調した場合の評価があげられる．フォルトの発生確率が，電圧や周波数に応じてどのように変化するのかといったことや，削減電力はどの程度となるのか，といったことを評価環境を組んで，測る必要がある．また，その結果をもとに，より良いDVFSのポリシーを考えることも必要あると考えられる．

参考文献

[1] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. *Int.Symp on Microarchitecture*, pp. 196–207, 1999.

[2] D. Ernst, N.S. 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.symp.on Microarchitectre*, 2003.

[3] 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.

[4] R. E. Kessler. THE ALPHA 21264 MICROPROCESSOR. *IEEE Micro*, Vol. 19, No. 2, pp. 24–36, Mar. 1999.

[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] Microsoft Corporation. 2003. *Windows Native Processor Performance Control. Windows Platform Design Notes http://www.microsoft.com/whdc/system/pnppwr/powermgmt/ProcPerfCtrl.mspx*, May 2003.

[7] 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.

[8] Shidhartha Das, David Roberts, Seokwoo Lee, Sanjay Pant, David Blaauw, Todd Austin, Krisztian Flautner, Trevor Mudge. A Self-Tuning DVS Proces-

sor using Delay-Error Detection and Correction. *IEEE Journal of Solid-State Circuits (JSSC)*, April 2006.

[9] C. Weaver and T. Austin. A Fault Tolerant Approach to Microprocessor Design. In *Int.conf.on Dependable Systems and Networks*, pp. 411–420, 2001.

[10] 安藤 壽茂, 吉田 裕司, 井上 愛一郎, 杉山 五美, 浅川 岳夫, 森田 国樹, 牟田 俊之, 元車田 強, 岡田 誠之, 山下 英男, 薩川 禎彦, 紺本 明彦, 山下 良一, 杉山 広行. 高信頼設計 SPARC64 V マイクロプロセサ. *Technical report of IEICE. DSP*, Vol. 103, No. 380, pp. 35–40, 2003.

[11] 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.

[12] 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.

[13] 佐藤寿倫, 国武勇次. ばらつき耐性を持つカナリア ff を利用したデザインマージン削減による省電力化. 情報処理学会論文誌, pp. 2029–2042, 2008.

[14] 入江 英嗣, 杉本 健, 五島 正裕, 坂井 修一. レジスタファイルの書き込み時タイミングエラーの検出・回復手法. 先進的計算基盤システムシンポジウム SACSIS, pp. 235–244, 2007.

発表文献

主著論文

1. Out-of-Order スーパスカラプロセッサの FPGA への実装
   杉本 健，入江 英嗣，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, pp.196-197 (2007). (ポスター)

2. タイミング・エラー耐性を持つスーパスカラ・プロセッサ
   杉本 健, 入江 英嗣, 五島 正裕, 坂井 修一
   電子情報通信学会研究報告 CPSY2008-14, pp.19-24 (2008).

共著論文

1. Preventing Timing Errors on Register Writes: Mechanisms of Detections and Recoveries
   Hidetsugu Irie, Ken Sugimoto, Masahiro Goshima, Suichi Sakai
   ACM SIGARCH Computer Architecture News(ACM CAN),Vol. 35, No. SIG5, pp.25-31 (2007).

2. Preventing Timing Errors on Register Writes:Mechanisms of Detections and Recoveries
   Hidetsugu Irie, Ken Sugimoto, Masahiro Goshima and Shuichi Sakai
   2nd Int'l Workshop on Advanced Low Power Systems (ALPS), pp.31-38 (2007).

3. レジスタファイルの書き込み時タイミングエラーの検出・回復手法
   入江 英嗣，杉本 健，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, pp.235-244 (2007).

4. 動的タイミングエラー検出のための「書き込み保証バッファ」の評価
   入江 英嗣，杉本 健，五島 正裕，坂井 修一
   電子情報通信学会研究報告 ICD2007-29, pp.77-78 (2007).

5. パッシブ WAB の改良による低コストなレジスタ書き込みエラー検出手法
   入江 英嗣, 杉本 健, 塩谷 亮太, 渡辺 憲一, 五島 正裕, 坂井 修一
   電子情報通信学会研究報告 CPSY2008-3, pp.13-18 (2008).

謝辞

本研究を進めるにあたり，多くの方々から多大なご指導，ご協力をいただき大変お世話になりました．

坂井修一教授には，相談会等において研究内容に関する助言をいただきました．大変感謝しております．

五島正裕准教授には，本研究の進行から論文の添削，そして研究者としての心構えに至るまで，あらゆる面においてご指導いただきました．本当にありがとうございました．

入江秀嗣博士には，学部 4 年と修士 1 年に，直接指導していただいたのを始めとして，様々な点でお世話になりました．

事務補佐員の八木原晴水さん，月村美和さん，伊世知代さん，長谷部環さんには研究を行う上での事務等で，お世話になりました．

その他にも，坂井研究室の皆様には研究や論文執筆，研究室での生活のサポートなど様々な面でご協力いただき，大変お世話になりました．