博 士 論 文

高効率なメモリ順序違反検出機構に関する研究

指導教員

坂井 修一 教授

倉田 成己

概要

マルチコア・プロセッサが広く普及している昨今において，out-of-order スーパスカラ・プロセッサの面積効率・エネルギー効率が重要なものになっている．コアの面積・エネルギー効率の向上によって，チップに搭載するプロセッサ・コアの数を増やすことが可能となり，チップの最大性能を高めることができるからである．

Out-of-order スーパスカラ・プロセッサの構成要素の中では，ロード・ストア・キュー (LSQ) が最も高コストなものの 1 つとなっている．LSQ は，ロード/ストア命令の依存関係による先行制約を守りつつ，それらを投機的に実行する役割を果たす．その他の命令とは異なり，ロード/ストア命令の投機実行においては，ロード/ストア命令間の先行制約を満たすため，依存元のストア命令の発見，あるいは，メモリ・アクセス順序違反の検出など，動的なターゲット・アドレスの比較が必須である．この比較は，従来，CAM を用いて LSQ を構成することによって行われてきた．しかし CAM は，その構造上，回路面積と消費エネルギーが非常に大きい．

一方で，メモリの下位階層との速度差を埋めるため，in-flight なロード/ストア命令の数を増加させることは極めて重要であり，LSQ のエントリ数は増加の一途をたどっている．また，同時実行可能なロード/ストア命令の数を増やすことは，LSQ を構成する CAM のポート数の増加につながる．CAM の面積は，ポート数の 2 乗に比例して増加するため，プロセッサの規模の拡大に伴って LSQ の面積は加速度的に増大している．

そのため，フィルタを用いて LSQ の CAM の電力を削減する手法や，CAM を用いない手法が提案されてきた．これらは，RAM で構成されたフィルタを用いて，順序違反/フォワーディング・ミス検出を行うことに特徴がある．フィルタは，ターゲット・アドレスをキーとするハッシュ・テーブルであり，ハッシュ値の衝突による偽陽性を不可避的に伴う．しかし，このわずかな偽陽性を許容することによって，CAM を排除することが可能になるのである．

本論文では，このフィルタとして，Bloom Filter (BF) を用いる手法を提案する．

BF は，複数のハッシュ関数を用いることによって極めて低い偽陽性率を達成することに本質的な特長がある．過去には BF を用いたとする手法はいくつか提案されているが，これらはいずれも BF の本質的な特長である複数のハッシュ関数を用いておらず，サイズの割に高い偽陽性率に苦しんでいる．

また，BF を用いるにあたって必要な工夫である以下の点についても提案する．

1. ロード/ストア命令が投機ミス等によりフラッシュされた時，そのままではフィルタに不整合が生じるため，そうした場合にもフィルタの一貫性を保つ手法を提案する．
2. BF 自体の回路面積と消費エネルギーを削減するため，パラレル・カウンティング・ブルーム・フィルタ (PCBF) を採用するほか，カウンタ機能付き機能メモリ (functional memory) を用いて PCBF を構成する手法を提案する．特に後者は重要であり，同時実行可能なロード/ストア命令が多いハイエンドのプロセッサ・コアにおいて，RAM を用いて PCBF を構成するよりも大幅に有利となる．
3. 上述の PCBF を用いる手法では，偽陽性が発生したとき以外にも，カウンタがフルになったとき，ハッシュが衝突したときにペナルティが生じる．また，フィルタを用いた手法には，ロード/ストア命令のアクセス・サイズに起因してペナルティが増大する課題がある．本論文では，これらのペナルティを削減する PCBF の構成法についても提案する．

提案手法としてこれらの技術を組み合わせることが有効であることを確認するため，シミュレーションによる IPC（クロックサイクルあたりの実行命令数）の計測と，ツールによる回路面積および消費エネルギーの評価を行った．その結果，従来の CAM を用いた手法と比較して，平均 98.6%の IPC を維持しながら，回路面積を 20.3%，消費エネルギーを 24.4%まで削減できることが示された．

目 次

第 1 章　序論 **1**

1.1　ロード/ストア・キューの規模の増大 1

1.2　本論文の貢献 3

1.3　本論文の構成 5

第 2 章　ブルーム・フィルタ **7**

2.1　ブルーム・フィルタの基本 7

2.2　ブルーム・フィルタの陽性率 9

2.3　パラレル・ブルーム・フィルタ 10

2.4　カウンティング・ブルーム・フィルタ 13

第 3 章　順序違反/フォワーディング・ミス検出とロード/ストア・キュー **15**

3.1　Out-of-Order 実行の基礎 15

3.2　ロード/ストア命令間の曖昧な依存関係 17

3.3　順序違反/フォワーディング・ミス検出 18

3.3.1　メモリ依存予測 19

3.3.2　順序違反/フォワーディング・ミス検出 20

3.4　順序違反/フォワーディング・ミス検出とロード/ストア・キューの CAM 24

第 4 章　フィルタを用いた検出手法 **27**

4.1　フィルタを用いた手法の分類 27

4.2　各手法におけるフィルタの基本構成と基本的なアクセス手順 29

4.2.1　提案手法 29

4.2.2　Store Vulnerability Window 33

4.2.3　Delayed Memory Dependence Checking 38

4.2.4　Single Hash Filter を用いた手法 42

4.3　ロード/ストア命令のアクセス・サイズ 42

4.4 各手法の比較 43

第 5 章 提案手法の詳細 45

5.1 フィルタの基本構成とアクセス手順 46
5.2 フラッシュされたロード命令の遅延消去 46
5.2.1 命令のフラッシュに伴うフィルタの一貫性の欠如 46
5.2.2 予測ミスした命令の遅延消去 48
5.2.3 アーキテクチャ・ステートの保護 49
5.2.4 遅延消去による資源圧迫とその軽減 51
5.3 カウンタ機能付き機能メモリを用いた PCBF の構成手法 53
5.3.1 PCBF を RAM で構成することの問題点 54
5.3.2 カウンタ機能付きメモリを用いた PCBF 55
5.3.3 カウンタ機能付きメモリを用いた PCBF の動作 57
5.4 PCBF の利用に伴うペナルティの削減手法 59
5.4.1 衝突時の動作 60
5.4.2 陽性時の動作 61
5.4.3 オーバーフローへの対処 62
5.5 ロード/ストア命令のサイズに起因したペナルティの削減手法 . . . 63
5.5.1 MCBF 63
5.5.2 SCBF 64
5.5.3 StBF 64

第 6 章 性能評価 67

6.1 評価環境 67
6.2 提案手法の構成の検討 68
6.2.1 フィルタの容量に対する k および c の効果 68
6.2.2 サイズを考慮した PCBF の効果 71
6.2.3 IPC 低下の内訳 74
6.2.4 CAM を用いた LSQ と Store Set を組み合わせたときの偽陽性 75
6.3 既存手法との IPC の比較 78
6.3.1 評価モデル 78
6.3.2 評価結果 79

6.4 提案手法の回路面積と消費エネルギーの評価 81
6.4.1 評価モデル 81
6.4.2 評価結果 82

第 7 章 関連研究 **87**
7.1 LSQ を縮小する手法 87
7.2 メモリ依存予測と投機的フォワーディング 88
7.3 ロード再実行 89
7.4 その他のフィルタ手法 90

第 8 章 結論 **93**
8.1 本論文のまとめ 93
8.2 今後の課題 94

謝辞 **97**

参考文献 **97**

著者発表論文 **105**

図 目 次

1.1 POWER8 プロセッサ・コアのチップ写真 [3] 3

2.1 ブルーム・フィルタの例 8
2.2 BF のエントリ数と 陽性率の関係 10
2.3 パラレル・ブルーム・フィルタの例 11
2.4 Parallel および True BF のエントリ数と positive 率 12

3.1 Out-of-Order コアの基本的なパイプライン構成 16
3.2 Store Set 予測器 20
3.3 パイプラインの表記 21
3.4 メモリ・アクセス順序違反の例 22
3.5 フォワーディング・ミスの例 23

4.1 提案手法の順序違反検出 30
4.2 提案手法のフォワーディング・ミス検出 32
4.3 SVW の順序違反検出 35
4.4 SVW のフォワーディング・ミス検出 37
4.5 DMDC の順序違反検出 40

5.1 ロード命令の実行後にフラッシュされた場合のパイプライン 47
5.2 提案手法:予測ミスした命令を遅延消去する場合のパイプライン . . 48
5.3 命令のフラッシュが発生した際の動作:既存 50
5.4 命令のフラッシュが発生した際の動作:遅延消去 51
5.5 予測ミスした命令を遅延消去する手法の相対 IPC 52
5.6 1 ポート RAM セル・アレイ（左）と 2 ポート RAM セル・アレイ（右）の模式図 54
5.7 カウンタ機能付き機能メモリを用いた PCBF の構成 56

5.8 カウンタ機能付き機能メモリの例：3 ビットの場合 57
5.9 $k = 4$，$m' = 128$ のときの同時アクセス数 n と衝突率 P_{colAll} の関係 . 61
5.10 提案フィルタの構成 64

6.1 k の効果 偽陽性率（上）と相対 IPC（下） 70
6.2 c の効果 相対 IPC 71
6.3 MCBF の構成 相対 IPC 72
6.4 SCBF の構成 相対 IPC 73
6.5 オーバーフロー時のペナルティ 75
6.6 PCBF によるペナルティの内訳（上）と相対 IPC（下） 76
6.7 CAM を用いた手法におけるフォワーディング時の偽陽性（上）と，
Store Set による学習後の動作（下） 77
6.8 提案手法におけるフォワーディング時の動作 78
6.9 表 6.3の評価モデル 偽陽性率（上）と相対 IPC（下） 80
6.10 表 6.4の各モデルと L1D キャッシュの相対回路面積 83
6.11 表 6.4の各モデルの平均相対消費エネルギー 84

表 目 次

4.1 各手法の比較 28
4.2 各手法の比較と問題点 43

6.1 プロセッサの構成 69
6.2 PCBF のパラメータ 69
6.3 評価モデル 79
6.4 回路面積と消費エネルギーの評価モデル 82
6.5 ロード/ストアの動作と LSQ，フィルタに対する操作の関係 85

第1章

序論

1.1 ロード/ストア・キューの規模の増大

ハイエンドの out-of-order スーパスカラ・プロセッサ・コアの規模の拡大は，ゆっくりとだが確実に続いている．Intel Haswell プロセッサや IBM POWER8 プロセッサのように，ついに命令発行幅が 8〜10 のコアを持つプロセッサが市販されるようになった [1, 2].

チップ上に複数のプロセッサ・コアを搭載したマルチコア・プロセッサが広く普及している昨今においては，このコアの面積効率・エネルギー効率がより重要になる．コアの面積効率・エネルギー効率とは，回路面積，消費エネルギーあたりの実行速度のことである．コアの面積・エネルギー効率を向上させることができれば，チップに搭載するコアの数を増やすことで，チップの最大性能を高めることが可能となる．

Out-of-order コアは，out-of-order 実行のために数多くの RAM/CAM を持ち，ほとんど「多ポートの RAM/CAM のかたまり」と言ってよい．w-way の out-of-order コアでは，各 RAM/CAM のポート数と容量はそれぞれ w に比例する．また，RAM/CAM の回路面積はポート数の 2 乗と容量の積に，エネルギーはポート数と容量の積に比例する．したがって，w-way プロセッサの制御部の回路面積と消費エネルギーは，それぞれ，w^3 と w^2 に比例することになる．一方，演算器の面積・エネルギーは w にしか比例しないから，w を増やすと面積効率・エネルギー効率は急激に悪化してしまう．よく「Out-of-order コアは肥大化する」と言われるが，これはこのようなことを指している．

こうしたout-of-orderコアの制御部の中でも，ロード/ストア・キューは最も高コストなものの1つとなっている．

ロード/ストア・キューとCAM

最近のout-of-orderコアにおいては，**ロード/ストア・キュー** (**LSQ**) は，単なるロード/ストア命令のバッファではなく，ロード/ストア命令の依存による先行制約を守りつつ，それらを投機的に実行する役割を果たす．

その他の命令とは異なり，ロード/ストア命令は「曖昧 (ambiguous)」である．すなわち，ロード/ストア命令間の依存関係は，ロード/ストア命令のターゲット・アドレスを計算した後でしか判明しない．ロード/ストア命令の投機実行においては，ロード/ストア命令間の先行制約を満たすため，依存元のストア命令の発見，あるいは，メモリ・アクセス順序違反の検出など，動的なターゲット・アドレスの比較が必須となる．

このターゲット・アドレスの比較は，従来，**CAM**を用いてLSQを構成することによって行われてきた．しかしCAMは，その構造上，回路面積と消費エネルギーが非常に大きい．

ロード/ストア・キュー規模の増加

その一方で，最近のハイエンドのout-of-orderコアにおいては，メモリの下位階層との速度差を埋めるため，in-flightなロード/ストア命令の数を増加させることは極めて重要であり，LSQのエントリ数は増加の一途をたどっている．

また，同時実行可能なロード/ストア命令の数を増やすことは，LSQを構成するCAMのポート数の増加につながる．前述のように，CAMの面積はポート数の2乗に比例して増加するため，コアの規模の拡大に伴ってLSQの面積は加速度的に増大することになる．

こうした理由により，最近のハイエンド・コアでは，LSQは最も高コストな構成要素の1つとなっている．図1.1に，POWER8プロセッサのコアのチップ写真を示す[3]．同図に示されているように，LSU（ロード/ストア・ユニット）がコアの1/4程度を占めている．LSUには，L1Dキャッシュ・アレイが含まれるが，それがLSUに占める割合は高くなく，残りの部分，すなわち，LSQの面積と消費エネルギーの削減が極めて重要であることが分かる．

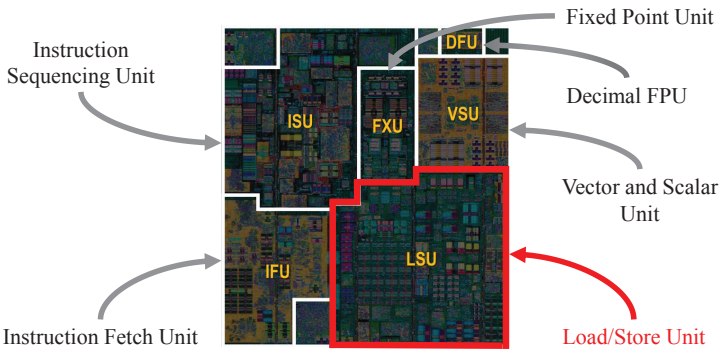

図 1.1: POWER8 プロセッサ・コアのチップ写真 [3]

フィルタを用いた順序違反/フォワーディング・ミス検出

こうした理由により，LSQ の CAM の電力を削減する手法や，CAM を用いないで LSQ を構成する手法が提案されている [4, 5, 6, 7]．これらの手法では，RAM によって構成された**フィルタ**を用いて順序違反/フォワーディング・ミス検出を行うことに特徴がある．

このフィルタは，ターゲット・アドレスをキーとするハッシュ・テーブルであり，ハッシュ値の衝突による偽陽性を不可避的に伴う．しかし，このわずかな偽陽性を許容することによって，CAM を排除することが可能になるのである．

本論文では，このフィルタとして，**ブルーム・フィルタ**を用いる手法を提案する．

1.2 本論文の貢献

本論文の貢献は，以下のとおりである．

1. フィルタを用いた手法の分類

フィルタを用いた手法には，特にフィルタに対するアクセス手順に関して，様々なバリエーションがあり [4, 5, 6, 7]，お互いに比較することが難しい．

本論文ではまず， 1. フィルタの基本構成，および，2. フィルタへの基本的なアクセス手順 の 2 点を基に，フィルタを用いた手法を分類する．その結果，本論文でとりあげる既存手法と提案手法を含む，多くの手法を体系的に説明することができる．

この結果，順序違反検出に関しては，Sethumadhavan らの手法 [4] と提案手法のものがより問題が少ないことが示される．またフォワーディング・ミス検出に関しては，順序違反検出の自然な延長により，問題の少ないアクセス手順を導くことができる．

2. ブルーム・フィルタの採用

本論文では，順序違反/フォワーディング・ミス検出のためのフィルタとして**ブルーム・フィルタ** (**BF**) を用いる手法を提案する [8, 9]．BF [10] は，複数のハッシュ関数を用いることによって極めて低い偽陽性率を達成することに本質的な特長がある．過去には BF を用いたとする手法はいくつか提案されている [4, 5, 7, 6] が，これらはいずれも BF の本質的な特長である複数のハッシュ関数を用いておらず，容量の割に高い偽陽性率に苦しんでいる．筆者の知る限り，既存研究で複数のハッシュ関数に言及したものは存在しない．

ただし，BF のように，命令を登録/削除するタイプのフィルタを採用するためには，ロード命令のフラッシュによるフィルタの不整合の問題を解決する必要がある．ロード命令は，実行時に BF に登録され，コミット時に BF から削除される．したがって，分岐予測ミスなどによって，ロード命令が実行後にコミットされずにフラッシュされた場合，登録されたロード命令が削除されないまま残ってしまうことになる．Sethumadhavan らの手法 [4] では，提案と同じタイプのフィルタを採用しているにも関わらず，この問題を解決していない．本論文では，フラッシュされたロード命令を BF からの削除のために残しておく手法を提案する．

3. ブルーム・フィルタ自体の面積・エネルギーの削減

さらに本論文では，BF 自体の回路面積・消費エネルギーを削減するため，**パラレル・カウンティング・ブルーム・フィルタ** (**PCBF**) [11] と呼ばれるタイプの BF を採用するほか，カウンタ機能付き**機能メモリ** (functional memory) を用いて PCBF を構成する手法を提案する．特に後者は，以下に述べるように，同時実行可能なロード/ストア命令が多いハイエンドのプロセッサ・コアにおいて重要となる．

通常は，各エントリにカウンタの値を格納する RAM を用いて PCBF を構成し，RAM の外部に置かれたアダーを用いてカウント・アップ/ダウンを行う．そのため，カウント・アップ/ダウンは read-modify-write となり，リードのためとライトのために，同時実行可能なロード/ストア命令の数の 2 倍のポート数が必要になる．

前述のように，RAM の面積はポート数の 2 乗に比例するため，これによる面積増加は深刻である．

提案手法で用いるカウンタ機能付き機能メモリとは，各エントリがカウント・アップ/ダウンのためのロジックを内蔵する RAM であり，カウンタのアレイと考えてよい．この機能メモリを用いれば，カウント・アップ/ダウンは各エントリの内部で行われ，アップ/ダウンに際してはカウンタの値をリード/ライトする必要がない．更にいくつかの工夫により，PCBF の面積を，同時実行可能なロード/ストア命令の数に比例しない，定数オーダとすることができる．

4. PCBF 採用に起因するペナルティの削減手法

上述した PCBF を用いる場合，偽陽性が発生した時以外にも，カウンタがフルとなった時，ハッシュの衝突が発生した時にもペナルティが生じる．それぞれのペナルティを削減する手法について提案する．

フィルタを用いた手法には，ロード/ストア命令のアクセス・サイズに起因してペナルティが増大する課題がある．本論文ではまた，このサイズに起因するペナルティを削減する PCBF の構成法についても提案する．

5. 評価

提案手法は，これらの技術の組み合わせにより，IPC を維持しながら，回路面積と消費エネルギーを大幅に削減することができる．

評価では，シミュレーションによる IPC の計測と，ツールによる回路面積と消費エネルギーの計測を行う．その結果，CAM を用いた方式に対して，平均 98.6% の IPC を維持しながら，回路面積を 20.3%，消費エネルギーを 24.4% まで削減することが示される．

1.3 本論文の構成

本論文の構成は，以下のとおりである：

1 章 序論 本章.

2 章 ブルーム・フィルタ 背景知識として，提案で採用するブルーム・フィルタ (BF) について説明し，複数のハッシュ関数を用いることが本質的に重要で

あることを示す.

また，ハードウェア実装のためBFのポート数を削減するパラレルBFと，要素の削除を可能とするカウンティングBFを紹介する．提案で使用するパラレル・カウンティングBF (PCBF) は，パラレルBFとカウンティングBFを組み合わせたものである.

3章 順序違反/フォワーディング・ミス検出とロード/ストア・キュー まず，背景知識として，out-of-order実行について簡単にまとめる．その後，ロード/ストア命令を投機的に実行する手法として，メモリ依存予測と，予測に付随して必須となるメモリ・アクセス順序違反検出とフォワーディング・ミス検出について説明する．また，順序違反/フォワーディング・ミス検出を行う上でのLSQの役割について説明する.

4章 フィルタを用いた検出手法 前章で述べた順序違反/フォワーディング・ミス検出を，フィルタを用いて行う検出手法について論ずる．1. フィルタの基本構成，および，2. フィルタへの基本的なアクセス手順 の2つの点に関して，既存手法と提案手法を含む，フィルタを用いる手法全般を体系的に説明する．そして，主にこの2点における既存手法の問題点を指摘する.

この章では，まずこの2点に基づいて各手法を分類する．次いで，この分類を踏まえて，各手法の2点を具体的に説明し，同時にこの2点における既存手法の問題点を明らかにする.

5章 提案手法の詳細 前章で述べた2点以外の提案手法の詳細について説明する．すなわち，フィルタとしてBFを採用するうえで必要となったフラッシュされたロード命令を残す手法，PCBFの面積とエネルギーを削減するカウンタ機能付き機能メモリの構成，PCBF採用に起因するペナルティの削減手法について，詳しく説明する.

6章 性能評価 提案手法の実行速度と回路面積，消費エネルギーについての評価を，既存手法と比較しながら行う.

7章 関連研究 本研究に関連する研究として，LSQの回路面積や消費エネルギーを削減する手法についてまとめる.

8章 結論 本論文の内容についてまとめ，今後の展望を示す.

第2章

ブルーム・フィルタ

提案手法の第一の特長は，CAM ではなくフィルタを用いる順序違反/フォワーディング・ミス検出手法のフィルタとして，**ブルーム・フィルタ** (**BF**) [10] を用いることにある．本章では，背景知識として，この BF について説明する．

まず次節で，BF の基本について説明し，その後 2.2 節で，ハッシュ関数の数と偽陽性率との関係を解析し，複数のハッシュ関数を用いることが BF において本質的に重要であることを示す．2.3節と 2.4 節では，BF を発展させた，パラレル BF とカウンティング BF をそれぞれ紹介する．実際に提案で用いるパラレル・カウンティング BF は，パラレル BF とカウンティング BF の組み合わせである．

2.1 ブルーム・フィルタの基本

ブルーム・フィルタ (**BF**) とは，1970 年に Burton H. Bloom が考案した空間効率のよい確率的データ構造で，要素が要素の集合に含まれているかどうかを判定するために用いられる [10]．判定には，**偽陽性** (False Positive) があるが，偽陰性 (False Negative) はない．したがって，BF の応答が陽性である場合には，普通，偽陽性か真陽性かを判定するための**確認検査**が必要となる．

ブルーム・フィルタの動作

BF は，m エントリのビットの配列と，k 個のハッシュ関数 $h_0, \ldots, h_{k-1}$ からなる．図 2.1の例を用いて，BF の動作を説明する．この例では，要素は八進で 000

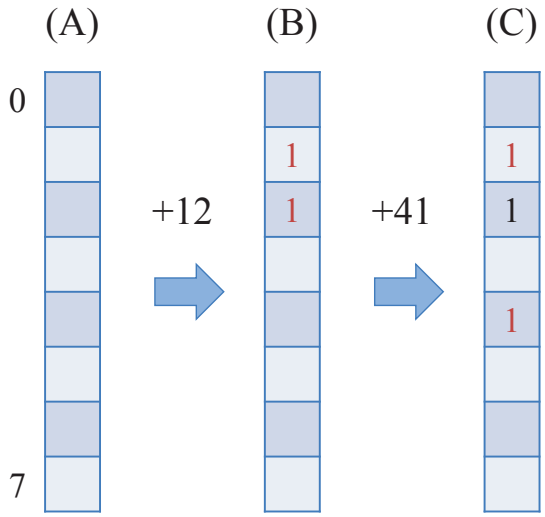

図 2.1: ブルーム・フィルタの例

から 077 までの 64 通りとする．BF のエントリ数は $m = 8$，ハッシュ関数は $h_0 =$ 要素の八の位，$h_1 =$ 要素の一の位 の 2 つとする．

(A)　初期状態では，BF の全てのビットが 0 である（図では 0 は省略）．

(B)　要素 012 を BF に追加するとき，この要素のハッシュ値 $h_0 = 1$，$h_1 = 2$ に対応するビットをセットする．

(C)　同様に，要素 041 を追加するときは，$h_0 = 4$，$h_1 = 1$ に対応するビットをセットする．
$h_1 = 1$ に対応するビットは既にセットされているが，特別な動作は行わない．実装上は，盲目的に 1 を上書きすればよい．

ここで，例えば (B) で追加された 012 を検索すると，$h_0 = 1$，$h_1 = 2$ に対応するビットがいずれもセットされているため，012 が追加されていたと判定できる．

ブルーム・フィルタの偽陽性

BF には，ハッシュ値の偶然の衝突によって，偽陽性が発生する．例えば，図 2.1 (C) の，012 と 041 が追加されている状態において，024 を検索すると，$h_0 = 2$，$h_1 = 4$ に対応するビットのいずれもセットされているため，実際には追加されていないにもかかわらず結果は陽性となる．

このような偽陽性が発生する確率は，配列のエントリ数を増加させるよりも，次節で述べるように，ハッシュの数を適切に設定することによって劇的に削減することができる．

2.2 ブルーム・フィルタの陽性率

BF の**陽性率** = 偽陽性率 + 真陽性率 は，ハッシュ値が一様に分布している場合，以下のように計算することができる．ある 1 つのハッシュ値によってあるエントリがセットされる確率は $1/m$ であるから，逆に，ある 1 つのハッシュ値によってエントリがセットされ**ない**確率は，$1-1/m$ である．したがって，n 個の要素を配列に追加したとき，合計 nk 個のハッシュ値によってあるエントリがセットされ**ない**確率は，$(1-1/m)^{nk}$ となる．よって，逆に，n 個の要素を配列に追加したとき，合計 nk 個のハッシュ値によってあるエントリがセットされ**る**確率は $1-(1-1/m)^{nk}$ となる．陽性率は，検索時に対象となる k 個のエントリが全てセットされている確率であるから，

$$P_{true} = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k} \tag{2.1}$$

となる．この式からでも，m を増加させるより，k をわずかに増加させることによって，P_{true} が劇的に減少することが分かるであろう．実用上は，P_{true} をある一定の値以下にすることを要求される場合が多い．その場合には，k をわずかに増加させることによって，必要なエントリ数 m を劇的に減少させることができる．図 2.2に，エントリ数 m に対する陽性率 P_{true} を示す．曲線は，$k = 1, 2, 3$ と，m に対して最適な k を選択した場合の，計 4 本ある．BF に追加されている要素数 n は，$n = 30$ である．ここで，例えば陽性率を 0.1%未満にしたい場合，$k = 1$ では約 $m \approx 30,000$ ものエントリが必要だが，$k = 2$ では約 $m \approx 2,000$，$k = 3$ では約 $m \approx 850$ となっており，k をわずかに増やすだけで必要エントリ数 m が劇的に小さくできることが分かる．

また，m，n が決まっているとき，偽陽性率を最小とする k は $k = \ln 2(m/n)$ で与えられ，この時の偽陽性率は，$P_{true} = (1/2)^k \approx 0.6185^{m/n}$ となる．すなわち，P_{true} を一定に保つためには $m \propto n$ なる m で十分である．このことはスケーラビリティの点で極めて重要である．例えば提案手法では，in-flight なロード命令数を

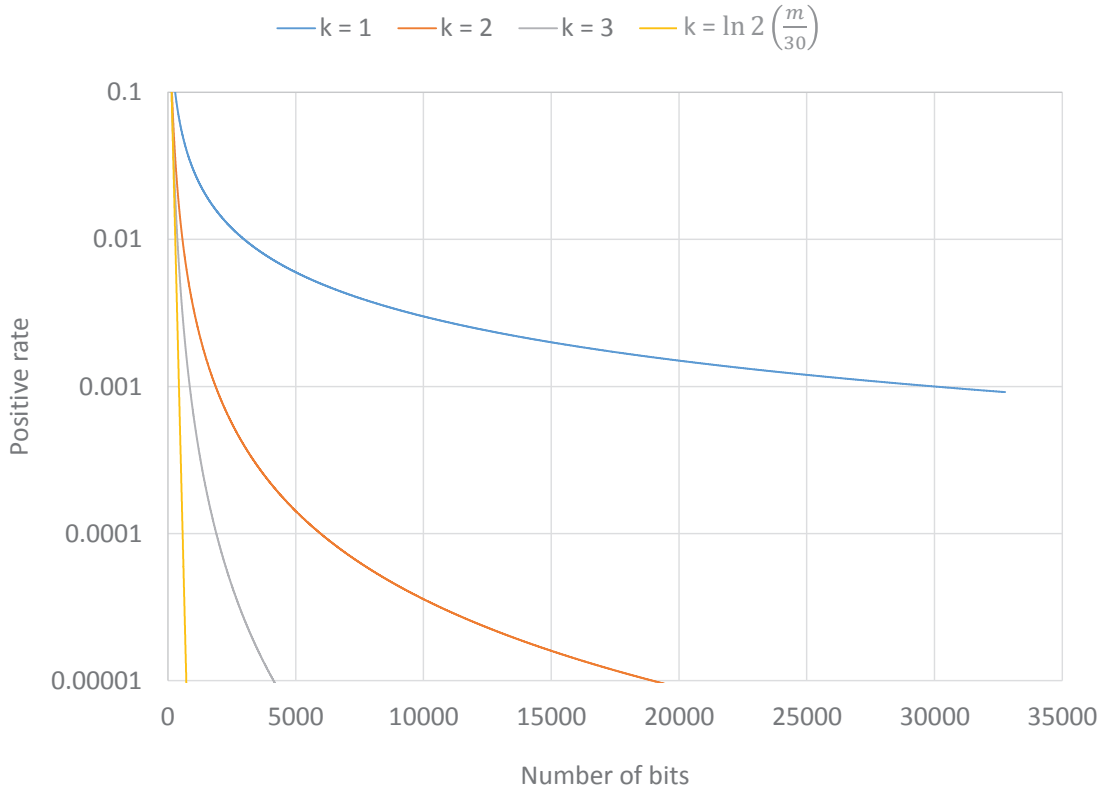

図 2.2: BF のエントリ数と 陽性率の関係

2 倍に増やした場合でも，フィルタのビット数も 2 倍に増やせば，同程度の偽陽性率を達成できることになる．

このように，ハッシュの数が $k \geq 2$ であることこそが，BF において本質的であると言える．しかし，BF を用いたと主張する研究はいくつかあるが，そのいずれにおいても $k \geq 2$ について言及されていない [4, 5, 6]．

2.3　パラレル・ブルーム・フィルタ

前述のように，BF においては $k \geq 2$ であることが本質的に重要である．ただしそのためには，1 回の追加・検索の度に，$k\,(\geq 2)$ 個のエントリにアクセスすることになる．ハードウェアにおいて，1 サイクルで追加・検索を行おうとすると，配列を構成する RAM のポート数を k 本とすることになろう．RAM の面積は，ポー

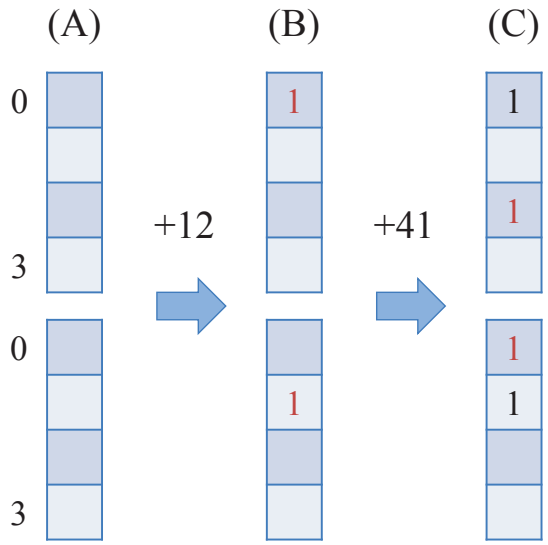

図 2.3: パラレル・ブルーム・フィルタの例

ト数の 2 乗に比例するため，k^2 に比例して増加することになる．

この問題は，**パラレル・ブルーム・フィルタ** (**PBF**) を用いることで解決できる [12]．PBF では，エントリ数 m の配列を，エントリ数 $m' = m/k$ の k 個の部分配列（サブ・アレイ）に分割し，各サブ・アレイは，それぞれ 1 個のハッシュ関数をインデクスとして読み書きすることにする（2.3参照）．すると，各サブ・アレイを構成する k 個の RAM のポート数はそれぞれ 1 で済む．

パラレル・ブルーム・フィルタの動作

図 2.3の例では，エントリ数 $8/2 = 4$ のサブ・アレイを 2 個用意し，それぞれのハッシュ関数は $h_0 =$ 要素の八の位/2，$h_1 =$ 要素の一の位/2 とする．要素 012 を追加する時には，上のサブ・アレイに対してはエントリ $h_0(1) = 0$ にアクセスし，下のサブ・アレイに対してはエントリ $h_1(2) = 1$ にアクセスする．

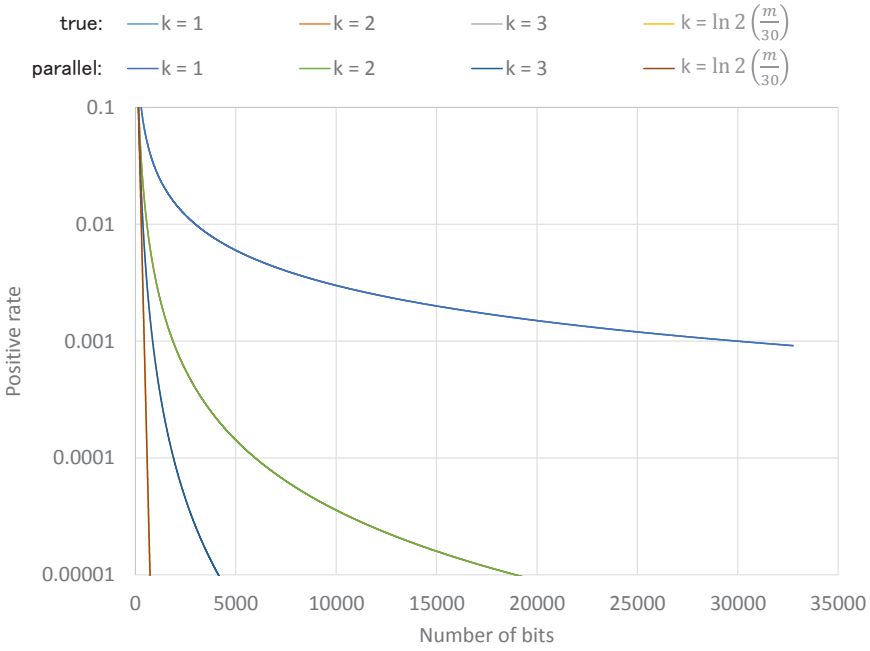

図 2.4: Parallel および True BF のエントリ数と positive 率

パラレル・ブルーム・フィルタの陽性率

パラレル・ブルーム・フィルタの陽性率 P_{para} は，パラレルではない BF とやや異なり，以下のようになる：

$$P_{para} = \left(1 - \left(1 - \frac{k}{m}\right)^{n}\right)^{k} \tag{2.2}$$

これは，2.2 節で述べた P_{true} (2.1) と比較すると大きいが，実用する領域においては，その差はごくわずかである．図 2.2と同じ条件下での P_{para} と P_{true} を，図 2.4に示す．P_{para} は P_{true} とほとんど重なっており，同等の効果が得られていることが分かる．

すなわち，PBF は，$k = 1$ の BF と同等の面積を保ちつつ，BF と同等に低い陽性率を得ることができるのである．

2.4 カウンティング・ブルーム・フィルタ

通常の BF は，要素の追加を行うのみで，削除することができない．BF の各エントリをビットからカウンタに拡張した**カウンティング・ブルーム・フィルタ** (**CBF**) を用いれば，削除が可能になる [13]．CBF では，要素に対応するカウンタを，追加時にインクリメントし，削除時にデクリメントする．また，要素の検索では，対応するすべてのカウンタの値が 1 以上であれば陽性とする．

通常の BF にはなかった問題として，CBF はカウンタのオーバーフローに対処する必要がある．カウンタのオーバーフローを放置すると，以降，正しく削除できなくなってしまうからである．

提案手法では，前節で述べた PBF と，本節で述べた CBF を組み合わせたパラレル・カウンティング BF 用いる．

第3章

順序違反/フォワーディング・ミス検出とロード/ストア・キュー

本章では out-of-order コアにおける順序違反/フォワーディング・ミス検出とロード/ストア・キューについて説明する．以下，まず，背景知識として，3.1 節と 3.2 節で，基本的な out-of-order 実行の基礎と，ロード/ストア命令間の曖昧な依存関係について説明する．3.3 節では，この曖昧性を除去する方法として，ロード/ストア命令の投機的実行を紹介し，そのために必須となる順序違反/フォワーディング・ミス検出について説明する．3.4 節では，順序違反/フォワーディング・ミス検出においてロード/ストア・キューの CAM の果たす役割について説明する．

3.1 Out-of-Order 実行の基礎

本節では，out-of-order スーパスカラ・プロセッサ・コアの基本的な動作原理について簡単にまとめたのち，特にロード/ストア命令がどのタイミングでメモリ（1 次データ・キャッシュ）にアクセスするかを明らかにする．

図 3.1に，out-of-order スーパスカラ・プロセッサの基本的な構成を示す．実際には，図に示す構成の他にいくつかのバリエーションが存在するが，メモリ・アクセス命令の動作に関しては大きく異なることはない．Out-of-order コアは，以下の 3 種類のパイプラインからなる：

- フロントエンド・パイプライン
- バックエンド・パイプライン

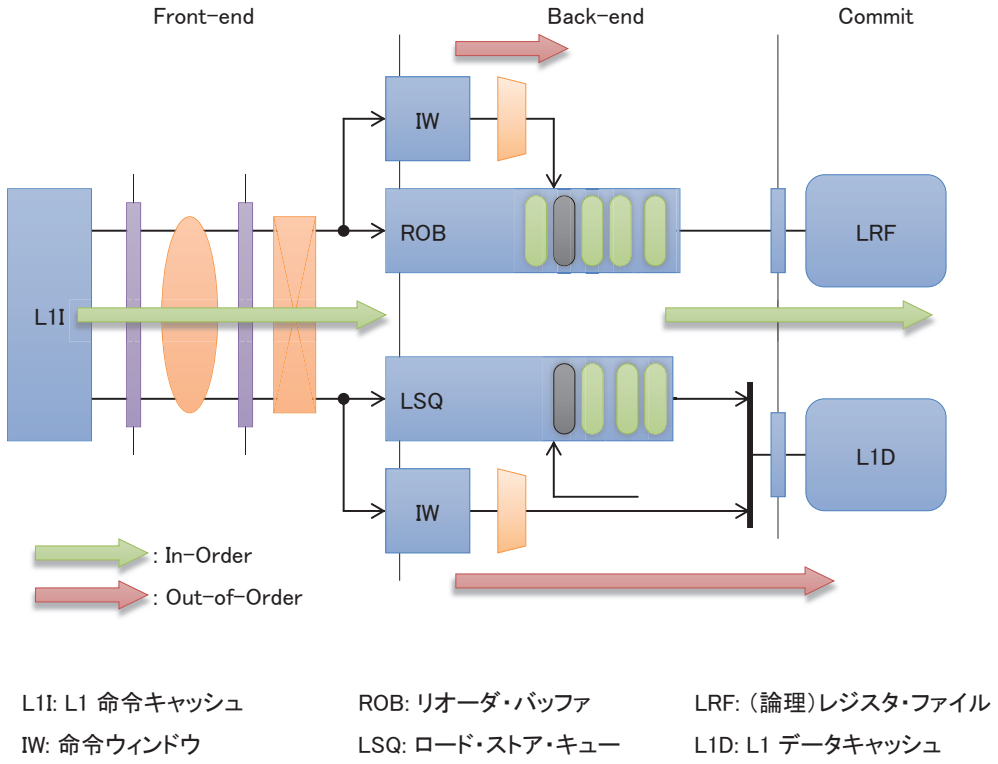

図 3.1: Out-of-Order コアの基本的なパイプライン構成（[14] より引用）

- コミット・パイプライン

これらのパイプラインは，命令を保持する以下のバッファによって分離されている：

- 命令ウィンドウ (IW)
- リオーダ・バッファ (**ROB**)
- ロード/ストア・キュー (**LSQ**)

Out-of-order コアでは，以下のように，命令キャッシュから in-order に供給された命令を命令ウィンドウに一旦保持し，命令の依存関係を保ちながら out-of-order に実行した後，再び in-order に状態を確定することで，プログラムの先行制約を守りつつ命令の実行順序の入れ替えを可能としている：

フロントエンド　実行される命令はまず命令キャッシュからフロントエンド・パイプラインに供給され，依存解析などが行われたあと，命令ウィンドウと ROB に送られる．また，ロード/ストア命令では LSQ に対応するエントリが確

保される．

ここで，ROB および LSQ は，out-of-order に実行された命令を再びプログラム・オーダ順に整列する役割をもつ．そのため，これらのバッファは FIFO の構造になっており，各エントリは命令列に対して in-order に割り当てられる．

バックエンド バックエンドでは，まず依存解析の結果を元に命令のスケジューリングが行われる．スケジューリングとは，命令ウィンドウ中の命令の中から依存関係を満たした命令を発見することである．スケジューリングによって選ばれた命令は命令ウィンドウから読み出され，演算器に送られて実際の計算を行う．計算された結果は，ROB におよび LSQ に格納される．

コミット コミットとは，**アーキテクチャ・ステート (AS)** の不可逆的な更新である．AS とは，**プログラム・カウンタ (PC)** や（**論理**）**レジスタ・ファイル (LRF)**,（キャッシュを含む）主記憶を指す．ROB には演算結果や例外情報などが，LSQ にはロード/ストア命令のターゲット・アドレスやストア・データなどがそれぞれ含まれており，実行結果を ROB/LSQ から in-order に読み出して行うことで，プログラム順に AS を更新する．

ロード/ストア命令の読み出し/書き込みタイミング

上記のように，メモリ（具体的には，1 次データ・キャッシュ）の更新はコミット時に in-order に不可逆的に行われる．そのため，ロード命令とストア命令では以下のようにメモリへのアクセス・タイミングが異なる．

ロード命令 ロード命令は実行時にアドレス計算を行い，キャッシュを読む．結果は LSQ/ROB に格納し，コミット時にレジスタ・ファイルへ書き込む．

ストア命令 ストア命令は実行時にはアドレス計算を行い，その結果を LSQ に書き込むが，キャッシュへの書き込みは行わない．コミット時に初めてキャッシュへの書き込み—AS の更新—を行う．

3.2 ロード/ストア命令間の曖昧な依存関係

本節では，データの依存とメモリの「曖昧性」について説明する．

命令がアクセスするレジスタの（論理）番号は，オペランドとして命令に埋め込まれている．つまり，レジスタを介した命令間の依存は**静的**であるため，実行前に依存を解決することができる．具体的には，命令セット上のレジスタ番号である論理レジスタ番号から，プロセッサ内部の物理レジスタ番号に書き換えるレジスタ・リネーミングをフロントエンドで行うことで，依存は解決される．

一方，ロード/ストア命令のターゲット・アドレスは，レジスタとは異なり，実行時に**動的**に決定される．特に，out-of-order コアにおいては，依存が存在するかどうかは，命令がコミットされるまで完全には分からない．このようなことを，ロード/ストア命令のメモリを介した依存の**曖昧性**という．この曖昧性のため，ロード/ストア命令を out-of-order に実行することは，その他の命令に比べて困難となる．

最も保守的には，ロード/ストア命令のみは in-order に実行する方法が採られる．しかしこの方法では，実際には依存が存在しない多くの命令をも in-order に実行することになり，IPC の向上の余地を見過ごすことになる．プログラムは一般に「ロード $\rightarrow$ 演算 $\rightarrow$ ストア」という命令列の断片の連続によって構成されている．もしロード命令がストア命令を追い越して実行することができれば，2 つ以上の断片を並列に実行することで，IPC が何倍も向上することがある．

このメモリの曖昧性に対処して，out-of-order にロード/ストア命令を実行することを**メモリ曖昧性除去**（メモリ非曖昧化，memory disambiguation）という．

ロード/ストア命令の投機実行は，このような技術の 1 つとして位置づけられる．ロード/ストア命令の投機実行では，予測に基づいて投機的にロード/ストア命令を実行し，メモリ・アクセス順序違反がないことを事後的に確認する．順序違反が検出された場合には，依存予測ミスであり，分岐予測ミスなどと同様の回復処理を行うことになる．全ロード命令のうちわずか 0〜7%しか順序違反を引き起こさないことが知られている [15].

次節では，こうしたロード/ストア命令の投機実行について詳しく説明する．

3.3　順序違反/フォワーディング・ミス検出

ロード/ストア命令を投機的に実行するには，以下の 2 段階の手順を踏む：

1. 実行の履歴に基づいて，ロード/ストア命令の依存関係を予測する
2. ロード/ストア命令を投機的に実行し，順序違反とフォワーディング・ミスを

検出する

以下，3.3.1 節で，メモリ依存予測についてまとめた後，3.3.2 節で，順序違反/フォワーディング・ミス検出について説明する．

3.3.1 メモリ依存予測

過去に同一のアドレスへアクセスし順序違反を起こしたロード/ストア命令のペアは，次回以降も同一のアドレスへアクセスする可能性が高い．そこで，順序違反を起こしたロード/ストア命令のペアの PC を学習し，次回以降はアドレスが一致すると仮定してスケジューリングを行う．

これを実現するための代表的なメモリ依存予測器として，**Store Set 予測器**がある [15]．6 章の評価では，メモリ依存予測器として，この Store Set 予測器を用いている．

Store Set 予測器

Store Set とは，あるロード命令に依存したことがあるストア命令の集合を指す．Store Set 予測器では，ロード命令が Store Set のすべてのストア命令に依存していると予測し，それらすべてのストア命令が実行された後でロード命令を実行する．

Store Set 予測器は，Store Set ごとに振られた **Store Set ID (SSID)** と呼ぶ ID を用いて予測を行う．図 3.2に，Store Set 予測器の概略を示す．Store Set 予測器は，主に以下の 2 つのテーブルを用いる：

Store Set ID Table (**SSIT**) は，SSID をエントリとして持ち，PC（の一部）をインデクスとしてアクセスする．

Last Fetched Store Table (**LFST**) は，最後にフェッチされたストア命令の動的な情報をもち，SSID をインデクスとしてアクセスする．

学習と，それに基づいた予測の手順は以下の通りである：

学 習 順序違反を検出したら，まず対となる先行ストアと後続ロードの PC がそれぞれ Store Set 予測器に伝えられる．それらの PC を用いて SSIT にアクセスし，それぞれに同一の SSID を書き込む．

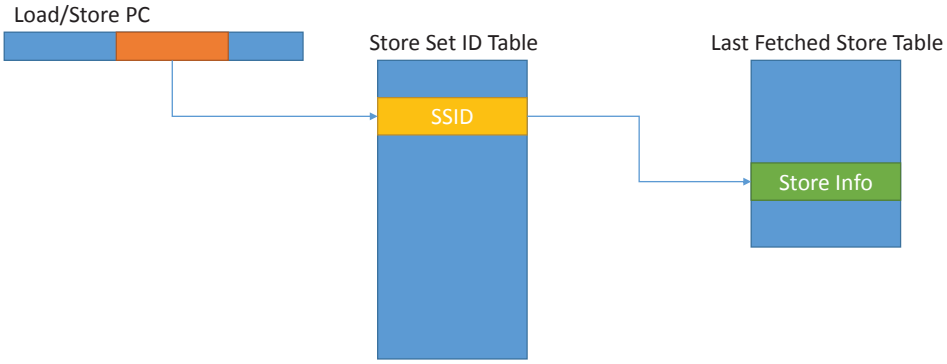

図 3.2: Store Set 予測器

予 測　ストア命令は，その PC を用いて SSIT にアクセスする．エントリに有効な SSID が格納されていれば，その値を用いて LFST にアクセスし，自身の動的な情報を書き込む．

ロード命令も，同様に PC を用いて SSIT にアクセスし，テーブル内の SSID を用いて LFST にアクセスする．LFST に存在するストア命令に依存しているとみなしてスケジューリングを行う．

3.3.2　順序違反/フォワーディング・ミス検出

ロード/ストア命令を投機的に実行するには，予測ミスの結果生じる順序違反とフォワーディング・ミスを検出する必要がある．本節では，順序違反/フォワーディング・ミス検出について説明する．

パイプライン図の表記法

パイプライン動作を説明する前に，本論文で用いるパイプライン図の表記法について説明する．順序違反/フォワーディング・ミス検出において意味があるのは，L1D アクセスを行うロード/ストア命令の実行（L1D アクセス）とコミット・ステー

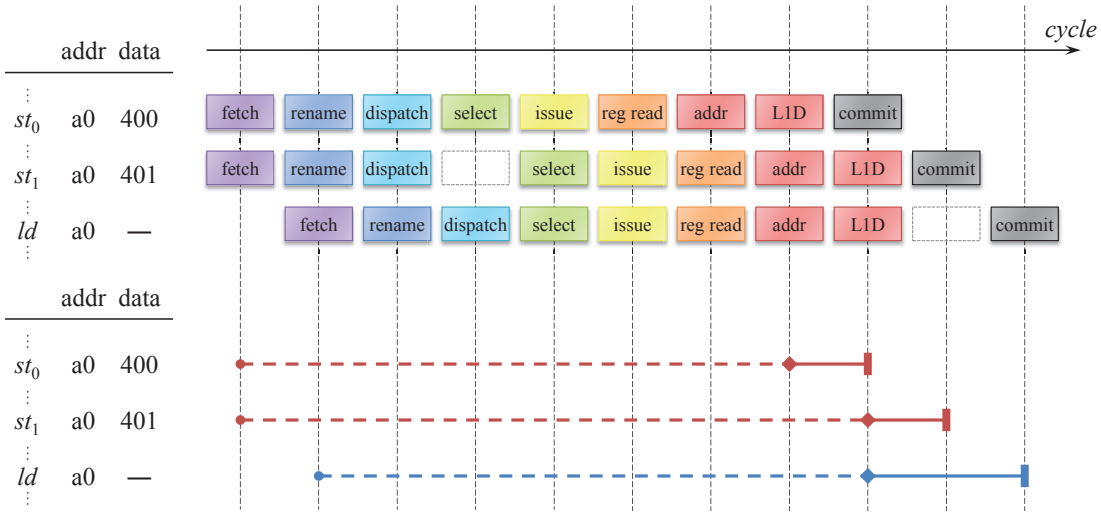

図 3.3: パイプラインの表記

ジの前後関係のみである．したがって，図 3.3 （下）に示すような，簡略化されたパイプライン図を用いることにする．

図 3.3（下）は，同図（上）に示す通常のパイプライン図に対応する．同図（上），（下）ともに，ストア 2 命令とロード 1 命令がフェッチ〜実行〜コミットされる様子を表している．同図（下）では，各命令の実行は菱形で，コミットは縦線で；また，フェッチ〜実行は破線，実行〜コミットは実線で，それぞれ表す．

メモリ・アクセス順序違反

図 3.4に，簡略化されたパイプライン図を用いてメモリ・アクセス順序違反の様子を示す．

同図では，ストア命令 st_0，st_1，ロード命令 ld が，その順序でフェッチされている．各命令のターゲット・アドレスはすべて $a0$ であり，st_0，st_1 のストア・データは，それぞれ，400，401 であるとする．プログラム・オーダ上，st_0 より st_1 の方が下流にあるから，ld は，st_0 のストア・データ 400 ではなく，st_1 のストア・データ 401 をロードしなければならない．

同図中，上が順序違反がない場合を示す．上部の右向き矢印は，(L1D の）アドレス $a0$ の値の変化を表す．st_1 のコミット後に ld が実行されており，ld は（L1D の）アドレス $a0$ から 401 をロードすることができる．一方，同図中下では，st_1 のコミットが ld の実行より遅れてしまったため，ld は st_0 のストア・データ 400 をロー

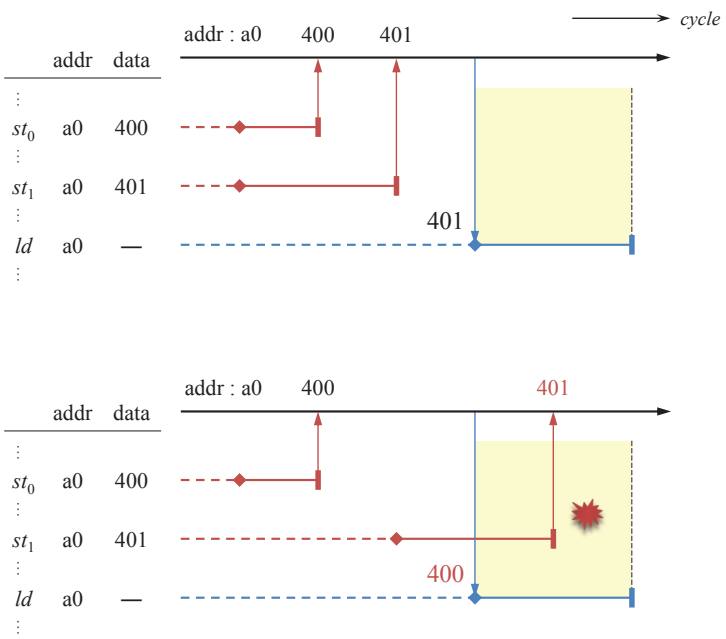

図 3.4: メモリ・アクセス順序違反の例：
順序違反がない場合（上）とある場合（下）

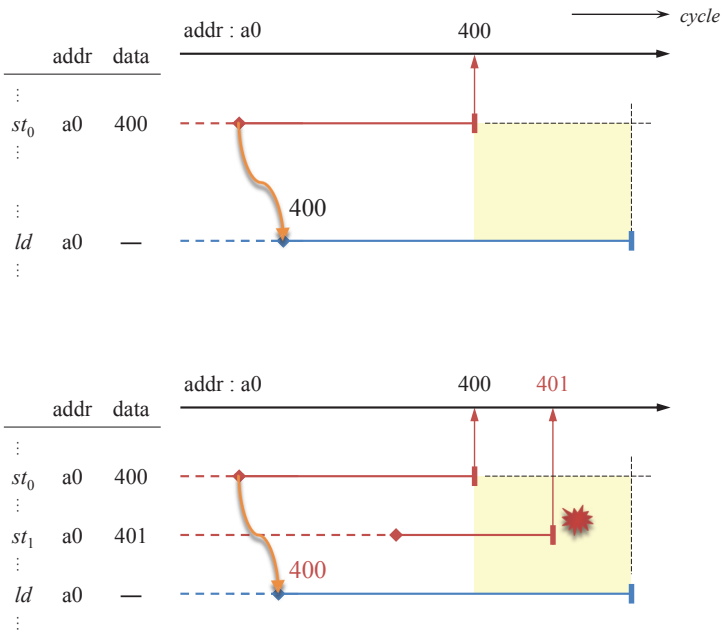

図 3.5: フォワーディング・ミスの例：
フォワーディング・ミスがない場合（上）とある場合（下）

ドすることになり，順序違反検出として検出しなければならない．

フォワーディング・ミス

図 3.5にフォワーディング・ミス検出の様子を示す．同図では，メモリ依存予測器からの指示に従い，st_0 のストア・データ 400 が ld に投機的にフォワーディングされている．ld は，コミット時に自身のターゲット・アドレス $a0$ をフォワーディング元の st_0 のそれと比較し，一致を確認する．しかしそれだけでは，フォワーディング・ミスがないと判断するには十分ではない．同図下では，st_0 より下流の st_1 のターゲット・アドレスもまた $a0$ となっており，ld は正しくは st_1 のストア・データ 401 をロードしなければならなかった．すなわち，フォワーディング・ミスである．

3.4　順序違反/フォワーディング・ミス検出と LQ/SQ の CAM

近年の out-of-order コアでは，LSQ を**ロード・キュー (LQ)** と**ストア・キュー (SQ)** に分離し，LQ にはロード命令のターゲット・アドレスを，SQ にはストア命令のターゲット・アドレスおよびストア・データを，それぞれ保持することが一般的になっている [2, 1]．本論文でも，LQ と SQ を分けて実装したモデルを想定する．

ロード/ストア命令の投機実行を行う場合には，LQ と SQ は，順序違反/フォワーディング・ミス検出において中心的な役割を果たす．CAM を用いる手法では，LSQ 自体を CAM によって実装し，CAM によって動的なターゲット・アドレスの比較を行う．

LSQ の CAM の果たす役割

CAM ベースの実装では，LQ/SQ に対して，以下のように優先順位付きの連想検索が行われる：

SQ　3.1 節で述べたように，L1D の更新は，不可逆的に行うため，通常ストア命令のコミット時に行われる．ストア・データは，ストア命令の実行〜コミットの間，SQ に置かれ，SQ から後続のロード命令への**フォワーディング**が行われる．ロード命令は実行時に SQ に対して連想検索をかけることになる．

LQ　Store Set などの依存予測器 [15, 16] を用いてロード/ストア命令を投機的に発行する場合，予測が誤りであるとメモリ・アクセス順序違反として検出される．順序違反検出は，ストア命令の実行時に LQ を連想検索し，当該ストア命令の後続のロード命令で，実行済みで，かつ，ターゲット・アドレスが一致するものがないかを探すことになる．

なお，順序違反検出の結果は，予測器の学習に用いられる．

LSQ の CAM の省略

提案手法を含め，本論文でとりあげる手法は，LQ/SQ の CAM の果たす順序違反/フォワーディング・ミス検出の役割を，フィルタによって置き換えるものである．

LQ/SQ の CAM は，同程度か，SQ の方が大きい．エントリ数は LQ の方大きいのが普通であるが，検索ポート数は SQ の方が等しいか大きいからである．LQ/SQ

の検索ポート数はストア/ロード命令の同時発行数で与えられるが，ストア命令の同時発行数はロード命令のそれに等しいかより小さいからである．

したがって，LSQ の面積と消費電力を問題にするのであれば，LQ と SQ の CAM を同時に省略することが望ましい．4 章で述べる手法では，ほぼすべてが LQ の CAM を省略している一方で，SQ の CAM を省略しているものは SVW と提案手法だけである．SQ の CAM を省略できない手法は，LSQ の問題のうち重要な部分が未解決であるということになる．

SQ の CAM を省略するためには，投機的フォワーディングを行うことになる [6, 17, 18, 19]．ただし，投機的フォワーディングを行えば，当然のことながら，フォワーディング・ミス検出を行う必要がある．

4 章で述べる SVW と本論文の提案手法では，LQ と SQ の CAM をほぼ同様の方法で解決している．このように，順序違反/フォワーディング・ミス検出は，LQ/SQ の CAM を省略するための手法であり，問題の規模と解決方法においても「双子」の問題であると言える．

第4章

フィルタを用いた検出手法

本章では，1. フィルタの基本構成，および，2. フィルタへの基本的なアクセス手順 の 2 点に関して，既存手法と提案手法を含む，フィルタを用いる手法を体系的に説明する．そして，この 2 点に関して，既存手法の問題点を明らかにする．

まず 4.1 節で，この 2 点に基づいて各手法を分類し，4.2.1 節で，この 2 点に関して各手法を具体的に説明する．

また，この 2 点とは別に，フィルタを用いた手法にはロード/ストア命令のアクセス・サイズに起因する課題がある．4.3 節で，この課題について触れる．

最後に，4.4 節で，これらの点に関して各手法の比較を行い，既存手法の課題を明らかにする．

4.1　フィルタを用いた手法の分類

フィルタを用いる手法の基本は，

1. フィルタ，すなわち，ロード/ストア命令のターゲット・アドレスのハッシュ値をキーとする RAM に対して，
2. ロード/ストア命令の実行/コミット時にリード/ライトを行うことで，

順序違反/フォワーディング・ミス検出を行うことにある．同一のターゲット・アドレスに対するロード/ストア命令は，フィルタの同一エントリへのリード/ライトを行うことになる．このエントリへのライトによって一方の命令がある特定の状態にあることを示し，他方の命令がそのエントリをリードし，値を検査することで検

表 4.1: 各手法の比較

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="2">Table</th><th colspan="2">Order Violation</th><th colspan="2">Forwarding Miss</th></tr>
<tr><th>value</th><th># hash func</th><th>write ↓ read</th><th>reset</th><th>write ↓ read</th><th>reset</th></tr>
<tr><td colspan="2">SVW</td><td rowspan="2">a. Sequence Number</td><td rowspan="4">1</td><td>i. st-cmt ↓ ld-cmt</td><td rowspan="2">—</td><td>i. same as OV</td><td>same as OV</td></tr>
<tr><td rowspan="2">DMDC</td><td>1st</td><td>ii. ld-exec ↓ st-cmt</td><td rowspan="3">N/A</td></tr>
<tr><td>2nd</td><td rowspan="3">b. Bit</td><td>i. st-cmt ↓ ld-cmt</td><td>Flash when safe</td></tr>
<tr><td colspan="2">Single Hash Filter</td><td>ii. ld-exec ↓ st-exec</td><td rowspan="2">ld-cmt</td></tr>
<tr><td colspan="2">Proposal</td><td>≧2</td><td>ii. ld-exec ↓ st-cmt</td><td>ii. st-cmt ↓ st-cmt</td><td>same as OV</td></tr>
</table>

出を行うのである．

表 4.1に，本章で紹介する各手法をまとめる．

各手法は，1. フィルタの基本構成 と，2. フィルタへの基本的なアクセス手順 の 2 点によって特徴づけられる．以下，それぞれについて説明する．

1. フィルタの基本構成

いずれの手法においても，フィルタを構成するテーブル (RAM) のキーにはロード/ストア命令のターゲット・アドレスのハッシュ値が用いられる．

一方，テーブルのバリューには，以下の 2 種類がある：

a. シーケンス・ナンバ　プログラム・オーダにしたがってロード/ストア命令にシーケンシャルに付された番号．

b. ビット　対応する命令が特定の状態にあることを示す 1 ビット．

b. ビット を用いる場合には，提案手法のように，複数のハッシュ関数を用いることが考えられる．一方，a. シーケンス・ナンバ を用いる場合には，複数のハッシュ関数を用いる方法は知られていない．

2. フィルタへの基本的なアクセス手順

フィルタへのライト → リードは，ロード/ストア命令の実行/コミットのいずれかのタイミングで行われる．提案手法を含め，既存の手法は，以下の 2 つに分類される：

i. st-cmt → ld-cmt 先行するストアのコミット時にライトし，後続のロードのコミット時にリードする．

ii. ld-exec → st-exec/cmt 後続のロードの実行時にライトし，先行するストアの実行，もしくは，コミット時にリードする．

なお，テーブルのバリューが b. ビット の場合には，ライトはビットのセットを意味する．また，リード/ライトに加えて，ビットのリセットを行う必要がある．

次節では，この分類に基づいて，各手法の動作を説明する．

4.2 各手法におけるフィルタの基本構成と基本的なアクセス手順

本節では，1. フィルタの基本構成 と 2. フィルタへの基本的なアクセス手順 の 2 点について，各手法を具体的に説明する．ここでは，読者の理解を助けるため，最初に 4.2.1 項で提案手法を，その後 4.2.2 項以降で既存手法について説明することとする．提案手法のアクセス手順が最も洗練されているので，まずそれによって順序違反/フォワーディング・ミス検出の基本を理解し，そのうえで，既存手法のより複雑なアクセス手順を理解するとよいであろう．また，既存手法を説明する際には，同時にそれらの問題点も指摘する．

4.2.1 提案手法

提案手法は，フィルタとして BF を採用する．前節の分類に従えば，BF は b. ビット にあたる．また，アクセス手順は，ii. ld-exec → st-cmt にあたる．

以下，提案手法の順序違反検出とフォワーディング・ミス検出についてそれぞれ説明する．

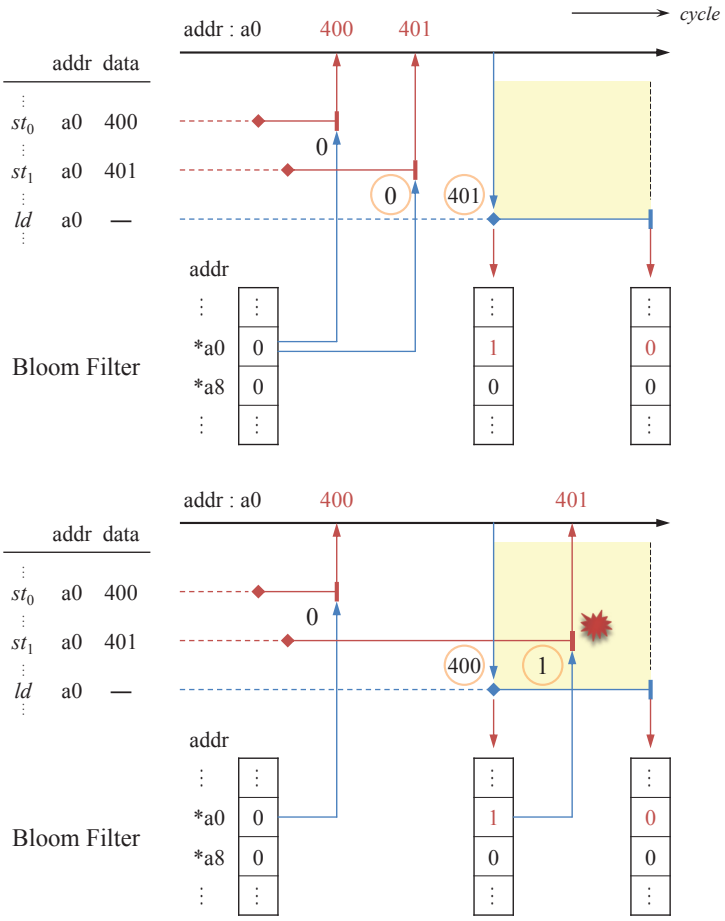

図 4.1: 提案手法の順序違反検出：
順序違反がない場合（上）とある場合（下）

順序違反検出

3.3 節で説明した簡略化されたパイプライン図を用いて，図 4.1に提案手法の順序違反検出の様子を示す．

同図は，図 3.4で説明した同じ状況である．つまり，ストア命令 st_0，st_1 と，ロード命令 ld のターゲット・アドレスがすべて $a0$ であり，ld は，st_0 のストア・データ 400 ではなく，st_1 のストア・データ 401 をロードしなければならない．

同図中，上が順序違反がない場合を示す．st_1 のコミット後に ld が実行されており，ld は $a0$ から 401 をロードしている．一方，同図中下では，st_1 のコミットが ld の実行より遅れてしまったため，ld は st_0 のストア・データ 400 をロードすることになり，順序違反検出として検出する．

提案手法は，前節の分類で言えば，b. ビット と，ii. ld-exec → st-cmt の組み合わせにあたる．ロード命令は，実行時にフィルタのビットをセットし，ストア命令は，コミット時にフィルタをリードしてビットを検査する．

この手法は，ロード命令が実行時にビットをセットすることで，実行以降（コミットまで）ターゲット・アドレス $a0$ に「監視領域」を設定すると考えると理解しやすい．この領域を同図中では網掛けで示した．この領域内で同一アドレスに対してコミットを行ったストア命令があれば，順序違反である．同図下では，実際に st_1 がこの領域内でコミットを行ったため，フィルタから 1 をリードし，順序違反として検出されることになる．

ロード命令は，コミット時に自らがセットしたビットをリセットする．その結果，監視領域はロード命令のコミット時までとなるが，それで十分である．コミットは in-order に行われるから，ld がコミット時にビットをリセットした後に，先行ストアがコミットを行うことはないからである．

フォワーディング・ミス検出

図 4.2に提案手法のフォワーディング・ミス検出の様子を示す．同図上では，予測器からの指示に従い，st_0 の正しいストア・データ 400 が ld に投機的にフォワーディングされている．一方同図下では，st_1 のストア・データ 401 をロードしなければならなかったにもかかわらず，st_0 のストア・データ 401 がフォワーディングされておりフォワーディング・ミスが発生している．

前述の順序違反検出の手法をわずかに変更することによって，この状況を検出することができる．すなわち，提案手法では，フォワーディングを行った場合には，

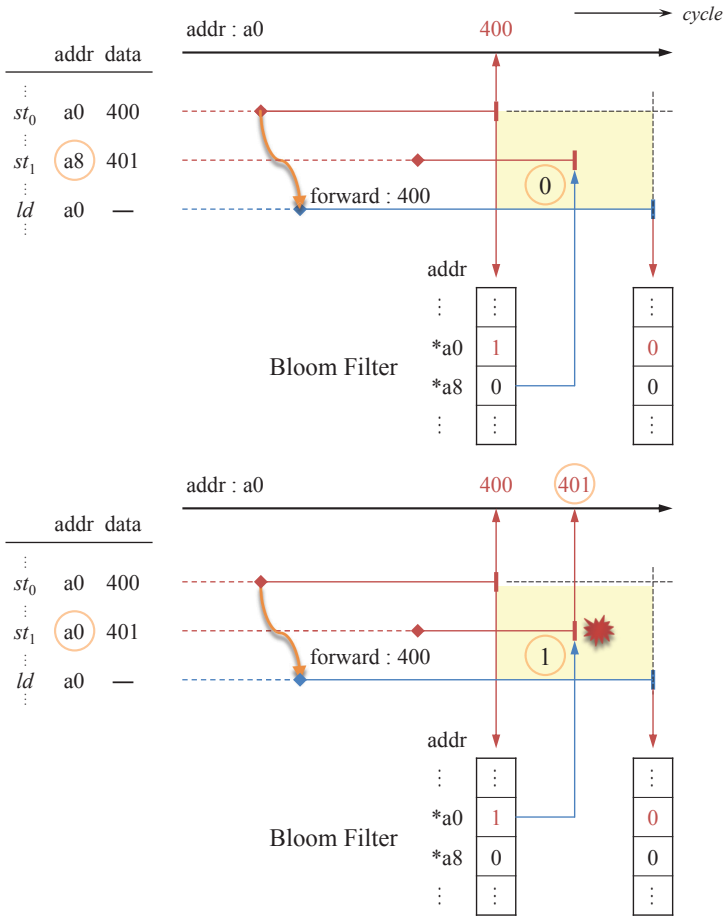

図 4.2: 提案手法のフォワーディング・ミス検出：
フォワーディング・ミスがない場合（上）とある場合（下）

ロード命令に代わって，フォワーディング元のストア命令がフィルタのセットを行うのである．その結果，監視領域は，図中網掛けした部分に変わる．順序違反検出の場合と同様に，この領域内で同一アドレスに対してコミットを行ったSTがあれば，フォワーディング・ミスである．同図下では，実際に st_1 がフィルタから1をリードし，フォワーディング・ミスとして検出されることになる．

なお，順序違反検出の場合と同様の理由により，監視領域はロード命令のコミット時までよい．

フィルタと確認検査

ここで用いられているフィルタはハッシュ・フィルタであり，ハッシュ値の偶然の**衝突**による偽陽性 (false positive, **FP**) が起こり得る．前出の例では，たとえ st_1 と ld のターゲット・アドレスが異なっていても，そのハッシュ値が一致した場合には，順序違反，フォワーディング・ミスとして検出されることになる．

そのため，これらの手法は**フィルタ**としての役割を果たすことになる．すなわち，低コストだが偽陽性が生じ得るフィルタによって，高コストだが偽陽性のない**確認検査**の実行頻度を減らすのである．提案手法の場合，確認検査はLQのシーケンシャル・サーチによって行うことを想定しており，これには数十サイクルもの時間がかかる．

したがってこれらのフィルタの性能は，一次的にはフィルタの容量に対する偽陽性率の低さによって評価されることになる．

4.2.2 Store Vulnerability Window

Store Vulnerability Window (**SVW**) は，元々は，ロード再実行と呼ばれる手法において，ロード再実行の頻度を削減するために提案された手法である [6]．**ロード再実行**とは，実行時に加えてコミット時にもロード命令を再実行し，両者のロード・データを比較することで順序違反を検出する手法である．ロード再実行を確認検査と捉えれば，SVWの手法はフィルタの役割を果たしている．

4.1節で述べた分類に従うと，SVWは，a. シーケンス・ナンバ と i. st-cmt → ld-cmt の組み合わせにあたる．

Store Sequence Number (SSN)

シーケンス・ナンバとしては，Store Sequence Number (**SSN**) と呼ぶ，ストア命令のみにプログラム・オーダ順にシーケンシャルに割り当てられた番号を用いる．

SSN をバリューとするテーブル本体は，Store Sequence Bloom Filter (**SSBF**) と呼ばれる．なお，Bloom Filter と名付けられてはいるが，ビットではなくシーケンス・ナンバをバリューとする場合，複数のハッシュ関数を用いる方法は知られておらず，文献 [6] でも複数のハッシュ関数に関しての言及はない．また，2.4 節で述べたカウンティング・ブルーム・フィルタとも異なることに注意されたい．

また，最後にコミットしたストア命令の SSN を保持するレジスタを用意し，ロード命令は実行時にこの値をリードする．これを $\mathrm{SSN}_{\mathrm{cmt}}$ と呼ぶ[1]．同一アドレスに対するストア命令とロード命令に関して，以下のことが言える；ロード命令の $\mathrm{SSN}_{\mathrm{cmt}}$ 以下の SSN を持つストア命令は，ロード命令実行時にはコミットしているのだから，このロード命令はそのストア命令のストア・データをロードしたことが保証される．逆に，$\mathrm{SSN}_{\mathrm{cmt}}$ より大きい SSN を持つストア命令があった場合には，順序違反である．

順序違反検出

4.1 節で述べた分類に従うと，SVW のテーブルへのアクセスは，i. st-cmt → ld-cmt となる．図 4.3の例では，ストア命令 st_{12} は，コミット時に，SSBF の $a0$ に対応するエントリと $\mathrm{SSN}_{\mathrm{cmt}}$ の 2 カ所に自身の SSN である 12 をライトする．st_{13} は，同じく 2 カ所に 13 をライトする．一方，ロード命令 ld は，実行時に $\mathrm{SSN}_{\mathrm{cmt}}$ をリードしておき，コミット時には SSBF の同じく $a0$ に対応するエントリをリードして，両者を比較することで検出を行う．

図 4.3（下）は，順序違反がある場合の動作である．ld は，実行時に $\mathrm{SSN}_{\mathrm{cmt}}$ として 12 をリードする．その後，st_{13} が SSBF の対応するエントリにコミット時に 13 をライトした後，ld がコミット時にリードすると 13 で，$\mathrm{SSN}_{\mathrm{cmt}}$12 より大きいため，順序違反が発生したことが分かる．

i. st-cmt → ld-cmt は，前節で述べた ii. ld-exec → st-cmt と逆の関係にあるため，ロード命令ではなくストア命令が「監視領域」を設定すると考えると分かりやすいかもしれない．図 4.3では，st_{13} が設定する領域を網掛けで示した．この領域内で ld が実行を行うと，13 より小さい $\mathrm{SSN}_{\mathrm{cmt}}$ をリードすることになり，順序違反

[1]元論文では，$\mathrm{SSN}_{\mathrm{NVUL}}$.

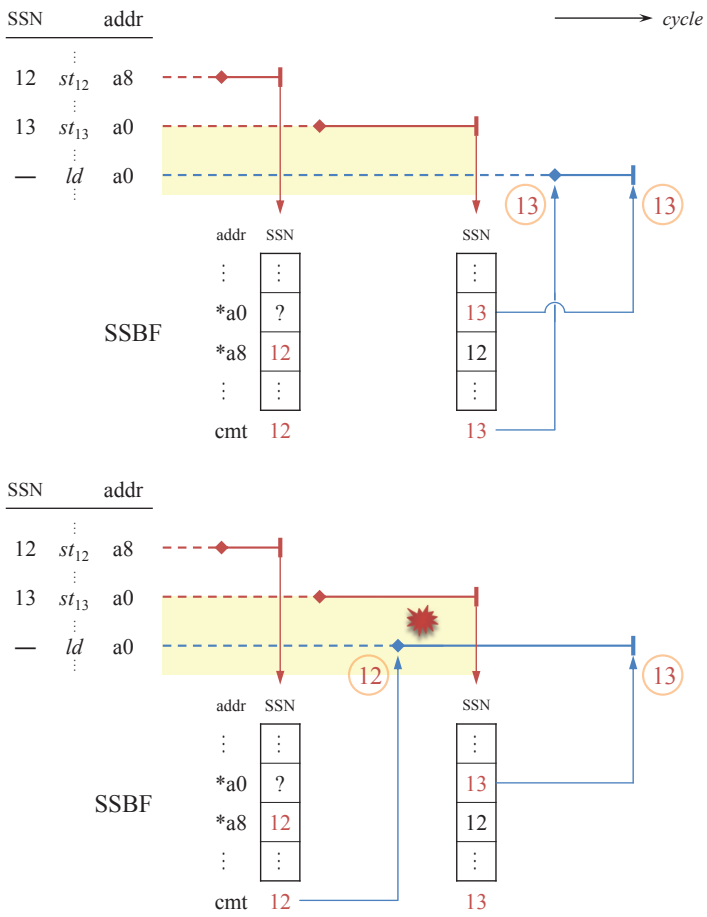

図 4.3: SVW の順序違反検出：
順序違反がない場合（上）とある場合（下）

検出となる．ただし，提案手法の場合と異なり，この領域の設定は事後的である，すわなち，ld の実行時には領域はまだ設定されておらず，この時点では順序違反検出であるかどうか分からない．そのため，ld の実行時には，コミット時に事後的に判定するための情報 —— $\mathrm{SSN}_{\mathrm{cmt}}$ を得ておくのである．提案手法に比べて SVW は一手間多いが，それはこのためである考えられる．

フォワーディング・ミス検出

SVW では，提案手法と同様，順序違反検出をわずかに変更することでフォワーディング・ミス検出も実現されている．SVW では，フォワーディング先のロード命令は，$\mathrm{SSN}_{\mathrm{cmt}}$ の代わりにフォワーディング元のストア命令の SSN を用いてコミット時の比較を行うことでフォワーディング・ミス検出を実現する．フォワーディングが行われる時は，ストア命令からロード命令にストア・データと同時に SSN も送られる．フォワーディング先のロード命令は，$\mathrm{SSN}_{\mathrm{cmt}}$ の代わりに送られて来た SSN を用いてコミット時の比較を行い，これらの値が一致していなければフォワーディング・ミスと判断できる．

図 4.4（下）では，ロード命令 ld がストア命令 st_{12} から値をフォワーディングされているが，実際に依存していたのは st_{13} である．このことは，送られて来た st_{12} の SSN 12 と，ld のコミット時に SSBF から読み出した st_{13} の SSN 13 を比較し，異なっているのでフォワーディング・ミスと判断できる．

a. シーケンス・ナンバ の問題点

テーブルのバリューとしてシーケンス・ナンバを用いる場合には，複数のハッシュ関数を用いる方法は知られていない．そのため，6 章における評価で詳しく述べるが，SVW は容量の割に高い偽陽性率を持つ．

i. st-cmt → ld-cmt の問題点

テーブルへのアクセスが i. st-cmt → ld-cmt の場合には，最後にロード命令のコミット時にテーブルをリードして，順序違反/フォワーディング・ミス検出を行うことになる．つまり，ロードのコミットの時に順序違反/フォワーディング・ミスが検出された時には，正しい依存元のストア命令が既にコミットしてしまっているのである．正しいストア・データはその時点で L1D に残っているため，例えばロード再実行によって得ることができ，その結果を持って確認検査とすることができる．

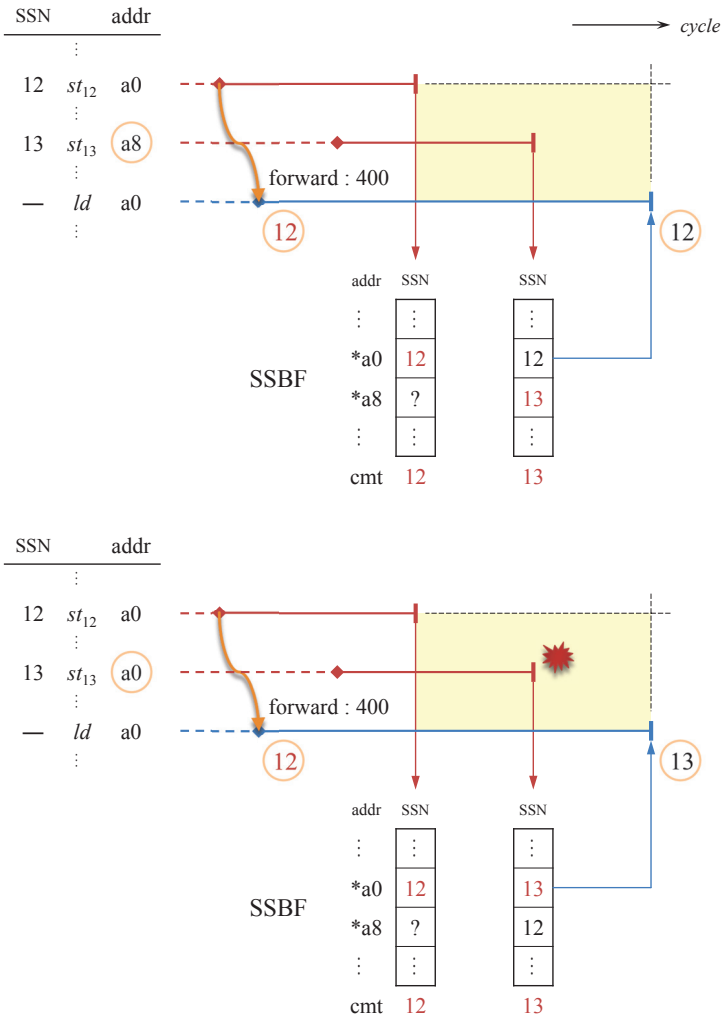

図 4.4: SVW のフォワーディング・ミス検出：
フォワーディング・ミスがない場合（上）とある場合（下）

問題は，依存予測器の学習である．

Store Set などの依存予測器の学習のためには，依存先のロード命令の PC に加えて，依存元のストア命令の PC が必要となる．しかし，コミットしたストアの PC を残しておくことは容易ではない．SQ は高価な資源であるため，SQ 上に PC を残しておくことは無駄が多い．その上，コミットしたストアの PC であるので，いつまで残しておけばよいかが分からない．

依存元のストア命令の情報が得られないと，以下の 2 つの問題が生じる：

確認検査ができない 偽陽性であるか真陽性であるかの判断するためには，ストア命令のターゲット・アドレスが必要である．

学習ができない Store Set をはじめ，依存予測器の学習には，ストア命令の PC が必要である [15, 16].

SVW では，第 1 の問題は，ロード再実行により解決している．ただし，ロード再実行は，コミット・パイプラインを乱すため，IPC 低下の要因となる．

第 2 の問題のためには，Committed Store PC Table と呼ぶテーブルを別途用意している．これは，ターゲット・アドレスをキー，ストア命令の PC をバリューとするタグレスの連想テーブルである．資源量の節約のためにタグレスとしたので，得られる PC が正しいとは限らない．間違っていた場合には依存予測器が汚されることになる．だがこのテーブルの一番の問題は，フィルタ本体に匹敵するほど大きいことである．

提案手法などのように，ii. ld-exec $\rightarrow$ st-exec/cmt とする方式の場合には，先行するストア命令のコミット時に検出が行われ，その時点で後続のロード命令はコミット前であるので，このような問題は起こらない．提案手法では，LQ のシーケンシャル・サーチによって，確認検査とロードの PC を得る．

4.2.3 Delayed Memory Dependence Checking

Delayed Memory Dependence Checking (**DMDC**) [7] は，以下の 2 段のフィルタからなっている：

前半 シーケンス・ナンバを用いて，ロード/ストア命令の実行順序の入れ替わりを検出する．

後半 前半で実行順序が入れ替わったと判定されたロード/ストア命令に対して，ターゲット・アドレスのハッシュ値をキーとするハッシュ・フィルタを用いて，同一アドレスに対するものであるか検証する．

前半部

前半のフィルタは，前節で述べたSVWのほぼ逆になっている．SVWがストア命令に付したSSNを用いるのに対して，DMDCはロードとストアの両方の命令に付したLSN[2]を用いる．SVWでは，SSN_{cmt} レジスタに対して，ストア命令がコミット時にSSNをライトし，ロード命令が実行時にリードするのに対して；DMDCでは，LSN_{exec} レジスタ[2]に対して，ロード命令が実行時にLSNをライト[3]し，ストア命令が実行時にリードする．DMDCでは，ストア命令は，リードしたLSNと自身のLSNを比較し，リードした方がより大きければ，自身より下流のロード命令が既に実行を終えたことが分かる．

この LSN_{exec} は，レジスタではなく，ターゲット・アドレスのハッシュ値をインデクスとするテーブルとしてもよい．

後半部

後半では，前半で実行順序が入れ替わったと判定されたロード/ストア命令に対して，ターゲット・アドレスのハッシュ値をキー，ビットをバリューとするハッシュ・フィルタ (**HF**) を用い，同一アドレスに対するものであるか検証する．このHFへのアクセスは，i. st-cmt → ld-cmt である，すなわち，ストア命令のコミット時にセットされ，ロード命令のコミット時に参照される．ロード命令の参照時にビットがセットされていた場合，順序違反の可能性がある．

順序違反検出

図 4.5（上）は，順序違反がない場合のDMDCの動作を表す．ld_{11}，ld_{13} は，実行時に自身のLSN11，13を LSN_{exec} にライトしている．st_{12} が実行時に LSN_{exec} を

[2]LSN と LSN_{exec} は，文献 [7] では，AGE と YLA (Youngest Load Age) で，AGE が最も大きい命令が youngest である．

[3]LSN_{exec} の更新は，read-compare-write になる，すなわち，既により大きい値がライトされている場合には，上書きしてはならない．これは，最後に実行したロード命令の LSN を保持する必要があるためである．SVW の SSN_{cmt} の場合，コミットは in-order に行われるので，このようなことは必要ない．

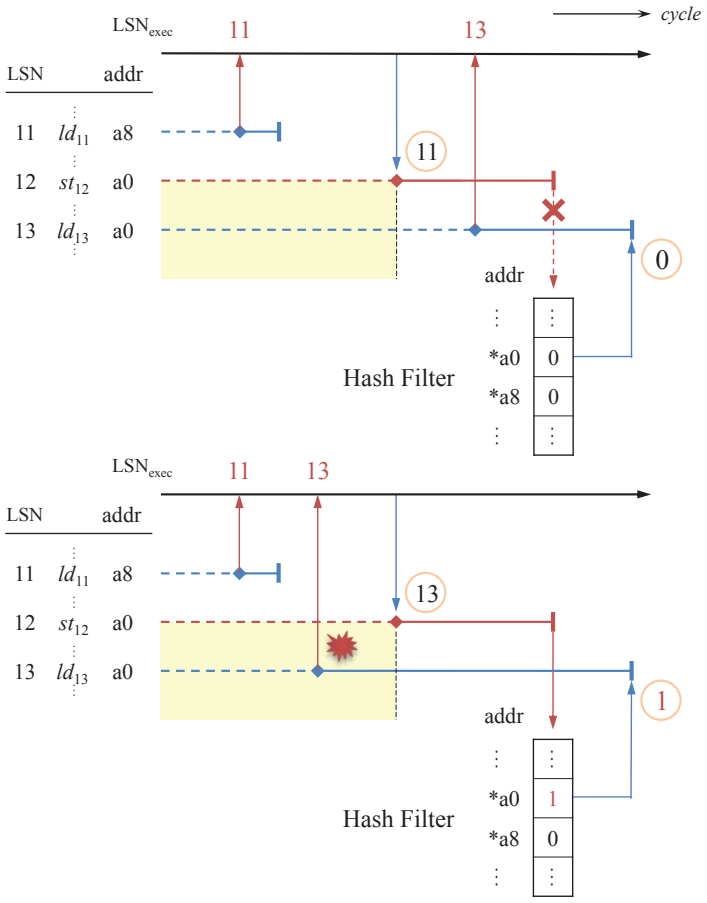

図 4.5: DMDC の順序違反検出：
順序違反がない場合（上）とある場合（下）

リードすると 11 が得られる．これは自身の LSN12 よりも小さいので，実行順序の入れ替わりは起こっておらず，順序違反はないと判断できる．この場合，st_{12}HF へのライトを行わない．

図 4.5（下）では，st_{12} と ld_{13} の実行順序が入れ替わっている．st_{12} が実行時に $\mathrm{LSN}_{\mathrm{exec}}$ を参照すると，自身の LSN12 より大きい 13 が得られる．そのため，st_{12} は後半の検査のためにコミット時に HF のビットをセットする．その結果，ld_{13} がコミット時に HF をリードすると対応するビットがセットされていることになる．これにより，ld_1 は順序違反の可能性があることがわかる．

フォワーディング・ミス検出

DMDC では，CAM によるフォワーディングを仮定している．したがって，この手法の延長線上では SQ の CAM を排除することができない．

CAM を用いたフォワーディングであっても，フォワーディング・ミスは発生することに注意されたい．例えば，図 4.5（下）の例では，ld_{13} の実行時には，st_{12} はまだ実行されていない．st_{12} のターゲット・アドレスも分からないので，CAM を持つ SQ からであっても，ストア・データを得ることはできない．

DMDC の場合，この状況では図 4.5（下）を例に用いたことからわかるように，順序違反として検出されることになる．

i. st-cmt → ld-cmt の問題点

DMDC では，後半のフィルタのアクセスが i. st-cmt → ld-cmt となっており，SVW と同様の問題が生じる，すなわち，依存先のロード命令のコミット時に順序違反が検出さたとき，依存元のストア命令は既にコミットしているため，確認検査ができない，学習ができない という 2 つの問題が生じる．

文献 [7] では，この問題には対処していない，すなわち，陽性であればフラッシュ再実行を行い，依存予測器は用いない (optimistic)．文献で [7] では，optimistic であっても順序違反は極めて稀 (very rare) で，このようにしても問題ないとしているが，Store Set をはじめとして，依存予測器を用いる他の多くの研究の結果と矛盾する [15, 16, 6].

4.2.4 Single Hash Filterを用いた手法

文献[4]には，CBFを用いてCAMに対するアクセス頻度を低減する手法が提案されている．4.1節の分類では，提案手法と同じく，b. ビット と ii. ld-exec → st-execの組み合わせになる．ただし，フォワーディング・ミス検出については考慮されていない．

この提案では，CBFを用いたとしてはいるものの，$k \geq 2$の場合についての言及はない．2種類のハッシュ関数を評価してはいるが，両者を比較しているだけで，それらを同時に用いてはいない．6章でも評価するが，$k = 1$では，mを相当大きくしたとしても偽陽性率はかなり高い．

この高い偽陽性率のため，[4]では，CAMを省略するには至ってない．CAMを排除すると，確認検査に時間がかかるため，それよるIPC低下が許容できなくなるためである．そのため[4]では，このフィルタによってCAMにアクセスする頻度を減らし，省電力化を図るにとどまっている．

4.3 ロード/ストア命令のアクセス・サイズ

一般的な命令セットでは，ロード/ストア命令には，1，2，4，8 Bのサイズが存在する．CAMによる順序違反検出では，マスクを用いることで比較的容易にこのサイズの違いに対応することができる．しかし，フィルタを用いた場合の対応は容易ではない．例えば，アドレス0x08〜0x0fの8 Bにアクセスするストア命令の後に，0x0c〜0x0fの4 Bにアクセスするロード命令があるとする．これらの命令は，始点となるアドレスが異なるが，アクセスされるバイトは重なっているため，当然順序違反検出の対象となる．しかし，単純に始点となるアドレスを用いてPCBFにアクセスすると，これらの命令は異なるアドレスへのアクセスとみなされ，偽陰性が発生してしまう．

例えばSVWでは，該当アクセスを含む8 Bワードを単位として順序違反検出を行っている[6]．先ほどの例で言えば，ストア命令とロード命令のいずれもアドレス0x08〜 0x0fまでの8 Bワードへのアクセスとみなし，アドレスを0x08とすることで順序違反を検出することが可能になる．しかし，この手法では1，2，4Bの隣接するアドレスへアクセスするストアとロードが同一アドレスへのアクセスとみなされ，一部のプログラムでは偽陽性が多発する．

また，DMDC ではこの問題に触れているものの，評価は行っていない．

4.4 各手法の比較

表 4.2に，本章でとりあげた手法を示す．この表は，基本的に前出の表 4.1 (p. 28) と同じものであるが，本章において問題があると指摘した項目を網掛けで示した．提案手法と比較した場合の既存手法の問題点は，以下のようにまとめられる：

- 複数のハッシュ関数に言及，および，評価した既存研究はない．特に，シーケンス・ナンバを用いると，複数のハッシュ関数を用いることはできない．
- i. st-cmt → ld-cmt は，確認検査ができない，依存予測器の学習ができない という 2 つの問題がある．解決のためには，LSQ とは別の手立てを必要とする．
- いくつかの手法はフォワーディング・ミス検出に対応しておらず，LQ の CAM は省略する一方で，SQ の CAM はそのまま残る．
- アクセス・サイズに起因する課題に関しては，触れられてはいるものの，詳しく評価するには至っていない．

表 4.2: 各手法の比較と問題点

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="2">Table</th><th colspan="2">Order Violation</th><th colspan="2">Forwarding Miss</th></tr>
<tr><th>value</th><th># hash func</th><th>write ↓ read</th><th>reset</th><th>write ↓ read</th><th>reset</th></tr>
<tr><td colspan="2">SVW</td><td rowspan="2">a. Sequence Number</td><td rowspan="3">1</td><td>i. st-cmt ↓ ld-cmt</td><td rowspan="2">—</td><td>i. same as OV</td><td>same as OV</td></tr>
<tr><td rowspan="2">DMDC</td><td>1st</td><td>ii. ld-exec ↓ st-cmt</td><td colspan="2" rowspan="3">N/A</td></tr>
<tr><td>2nd</td><td rowspan="3">b. Bit</td><td>i. st-cmt ↓ ld-cmt</td><td>Flash when safe</td></tr>
<tr><td colspan="2">Single Hash Filter</td><td>ii. ld-exec ↓ st-exec</td><td rowspan="2">ld-cmt</td></tr>
<tr><td colspan="2">Proposal</td><td>≧2</td><td>ii. ld-exec ↓ st-cmt</td><td>ii. st-cmt ↓ st-cmt</td><td>same as OV</td></tr>
</table>

第5章

提案手法の詳細

提案手法は，2 章で紹介した**パラレル・カウンティング・ブルーム・フィルタ** (**PCBF**) を用いて順序違反/フォワーディング・ミス検出を行う．提案手法は，4.2.1 節で述べたフィルタの基本構成とアクセス手順に加えて，いくつかの新規提案の要素技術からなる．前者に関しては，基本的には，4.2.1 節で述べたとおりである．本章では主に，これら以外の新規提案の要素技術について詳述する．

まず，BF のように，命令を登録/削除する b. ビット 型のフィルタを採用するためには，ロード/ストア命令のフラッシュによるフィルタの不整合の問題を解決する必要がある．Sethumadhavan らの手法 [4] では，提案と同じタイプのフィルタを採用しているにも関わらず，この問題を解決していない．本論文では，フラッシュされたロード命令の削除を遅延することによって不整合を回避する手法を提案する．5.2 節で，この手法について述べる．

5.3 節からは，PCBF の面積とエネルギー，そして，IPC ペナルティをさらに削減する技術群について述べる．5.3 節では，PCBF の面積とエネルギーを削減するカウンタ機能付き機能メモリの構成について述べる．5.4 節では，PCBF を用いることで発生する衝突や陽性，オーバーフローによるペナルティを削減する手法について述べる．最後に，5.5 節で，既存のフィルタを用いた手法で課題となっていたアクセス・サイズの問題を解決するフィルタ構成について説明する．

5.1　フィルタの基本構成とアクセス手順

提案手法のフィルタの基本構成とアクセス手順に関しては，基本的には 4.2.1 節で説明したとおりであるが，実際には PCBF を用いるため，若干の変更がある．

PCBF の採用に伴って，アクセス手順は以下のように変わる：

パラレル　ロード/ストア命令は，k 個のサブアレイに同時にアクセスする（2.3 節参照）．

カウンティング　ビットのセット/リセットが，カウンタのインクリメント/デクリメントに変わる（2.4 節参照）．

5.2　フラッシュされたロード命令の遅延消去

ロード命令が実行されフィルタをセットした後に，分岐予測ミスや例外処理等でそのロード命令がフラッシュされると，フィルタにフラッシュされたロード命令の情報が残されてしまい，一貫性が保たれなくなってしまう．本節では，この問題に対し，予測ミスした命令をすぐには取り除かない**遅延消去**と呼ぶ方法をとることにより，フィルタの一貫性を保つ手法を提案する．

5.2.1　命令のフラッシュに伴うフィルタの一貫性の欠如

図 5.1は，分岐予測ミスにより誤って実行されたロード命令がフラッシュされた場合のパイプラインを表す．br が予測ミスした分岐命令，ld が分岐予測ミスにより**誤って実行された**ロード命令，st が分岐予測ミスから回復したあと，**正しく実行された**ストア命令である．同図では，ld が実行時にフィルタのビットをセットしたあと，br が実行されて予測ミスが発覚している．このとき，ld がビットをリセットする前にフラッシュされているため，情報がフィルタに残ってしまい，後続の st の検査時に偽陽性が起こる．さらに，今後このビットをリセットする命令が存在しないため，偽陽性が発生し続けてしまう．

最もナイーブには，予測ミス等によりフラッシュが発生した際，フラッシュされる命令を順にたどり，フィルタのセットを行ったロード命令があればリセットする，という方法が挙げられる．しかし，フラッシュされる命令数が多い場合，命令の探

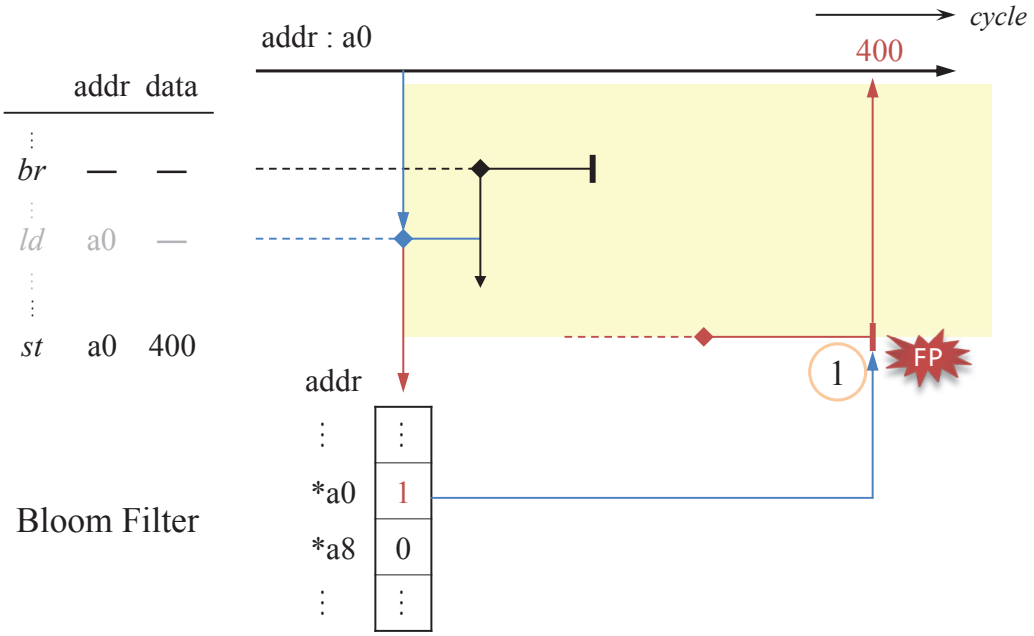

図 5.1: ロード命令の実行後にフラッシュされた場合のパイプライン

索に時間がかかり，分岐予測ミス・ペナルティが大きく増加する．特に，6 章で評価に用いるような LQ のエントリ数が 72 エントリ，読み出しポートが 2 ポート程度のハイエンド・プロセッサの場合，全てのロード命令を読み出すのに最大で数十サイクルもかかってしまうことになる．さらに，この方法ではフラッシュされたロード命令がフィルタのリセットを行うためにアクセス・ポートを使用するので，フラッシュされた命令の探索中は後続のロード命令がフィルタにアクセスできない．そのため，バックエンドをストールさせることになり，さらなる性能低下を招くこととなる．

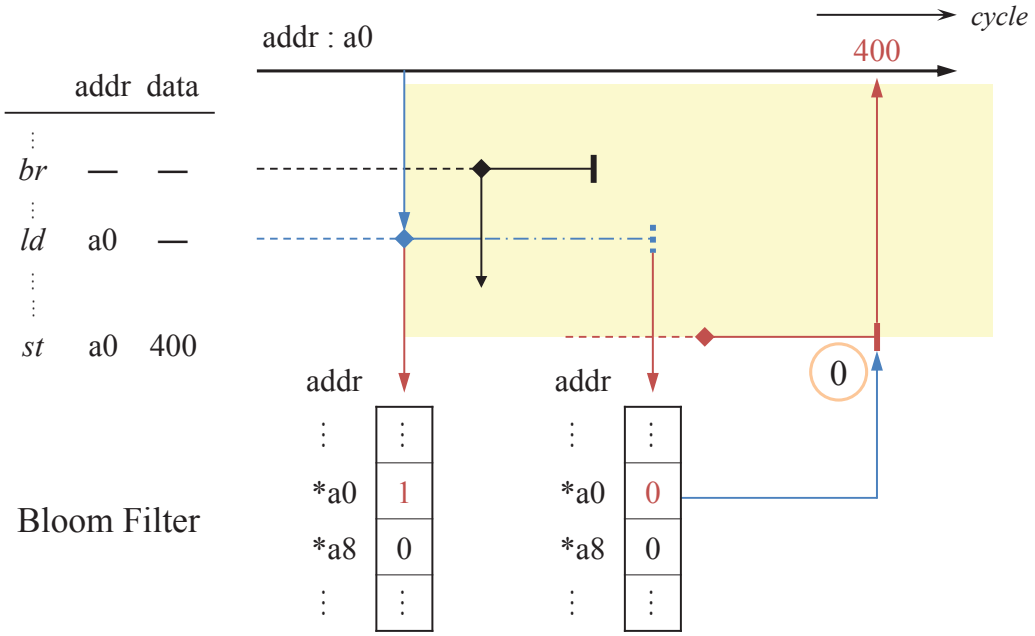

図 5.2: 提案手法:予測ミスした命令を遅延消去する場合のパイプライン

5.2.2　予測ミスした命令の遅延消去

この問題に対し，本論文では**予測ミスした命令を遅延消去する**ことにより，性能低下をほとんど起こすことなく PCBF の一貫性を保つ手法を提案する．図 5.2は，図 5.1と同様の条件において予測ミスした命令を遅延消去した場合のパイプライン図である．同図において，命令が予測ミスした状態にあることを一点鎖線で表している．*br* の分岐予測ミスによって予測ミスした *ld* をすぐには取り除かず，本来コミットが行われるタイミングで取り除く．その際，通常のコミットと同様にフィルタをリセットすることで，フィルタの一貫性を保つ．

ただし，これを実現するには以下の 2 点に注意する必要がある．

1. 予測ミスした命令は本来プログラム上にない誤った命令であるので，レジスタやキャッシュなどのアーキテクチャ・ステート (AS) を更新してはならない．

2. 予測ミスした命令を遅延消去すると，それらの命令がプロセッサの資源を圧迫し，性能低下につながる可能性がある．

以下では，これらに対処する方法について説明する．

5.2.3 アーキテクチャ・ステートの保護

命令のコミットは，3.1 節で示したように，実行結果を ROB/LSQ から in-order に読み出して行うことで，プログラム順に AS を更新する．遅延消去をする際，この読み出し時に遅延消去すべき命令であるかを判断することで，AS を保護することが可能となる．

通常，例外や投機ミス等による命令のフラッシュ時，ROB や LSQ 内の対応するエントリの削除は，以下に示すようなポインタの移動によってペナルティを削減している．

分岐予測ミス時の ROB の動作

図 5.3は，tail ポインタを動かすことによって ROB 内の分岐予測ミスした命令をフラッシュする様子を表している．図 5.3 (A) において，命令 i_1 から i_6 までが ROB に in-order に格納されている．このとき，ROB の head ポインタはプロセッサ内で最も先行する命令 i_1 を，tail ポインタは最も後続の命令 i_6 の次のエントリを指している．

ここで，命令 i_b が分岐命令であり，分岐予測ミスにより i_4 以降の命令がフラッシュされるとする．このフラッシュの動作は，図 5.3 (B) のように，ROB の tail ポインタを i_b の次のエントリ（$= i_4$ が格納されているエントリ）を指すことで，i_4 以降の命令の削除を即座に行っている．その後，図 5.3 (C) のように，正しいパスの命令 i_7〜i_9 を順に上書きして格納することで，ROB 内の命令を in-order に保つ．

なお，LSQ においても同様の手段をとることにより，分岐予測ミスのペナルティを削減しながらプログラムの実行順序を保っている．

遅延消去時の AS の保護

ここで，遅延消去を行う際に，単に ROB や LSQ に予測ミスした命令を残してしまうと，それらの命令がコミットされ，AS が更新されてしまう．そこで，AS を更新してはならないことを表すためのフラグ Update Prohibited Flag (UPF) を ROB

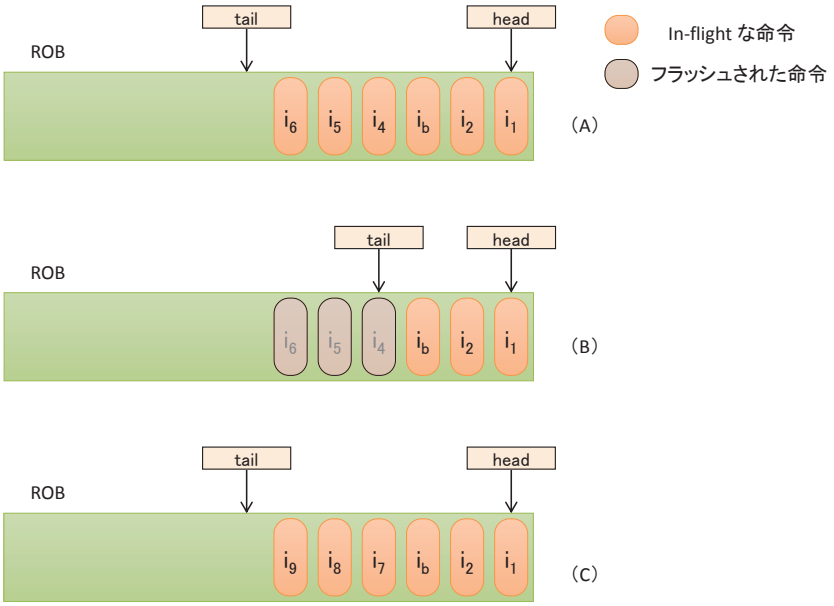

図 5.3: 命令のフラッシュが発生した際の動作:既存

に追加し，予測ミス時に遅延消去される命令に対してそのフラグを立てる．また，フラグが立てられた命令を Update Prohibited Instruction (UPI) と呼ぶ．

図 5.4は，提案手法において予測ミスが発生した場合の ROB の動作を表している．図 5.4 (A) は，図 5.3同様，命令 i_1 から i_6 までが ROB に格納されている様子を表す．ここで，i_b の分岐予測ミスが発覚したとき，tail ポインタを動かすのではなく，i_b 以降の命令 i_4〜i_6 に対して UPF を立てる（図 5.4 (B)）．その後，i_6 に続いて正しいパスの命令である i_7〜i_9 を順に格納する（図 5.4 (C)）．

UPI は通常のコミットと同様の手順により，ROB から順に擬似的にコミット (擬似コミット: Pseudo committment) されるが，AS の更新は行わない．UPI のうち，ロード命令のみ擬似コミット時にフィルタの対応するエントリをリセットないしデクリメントすることで，フィルタの一貫性が正しく保たれるようにする．

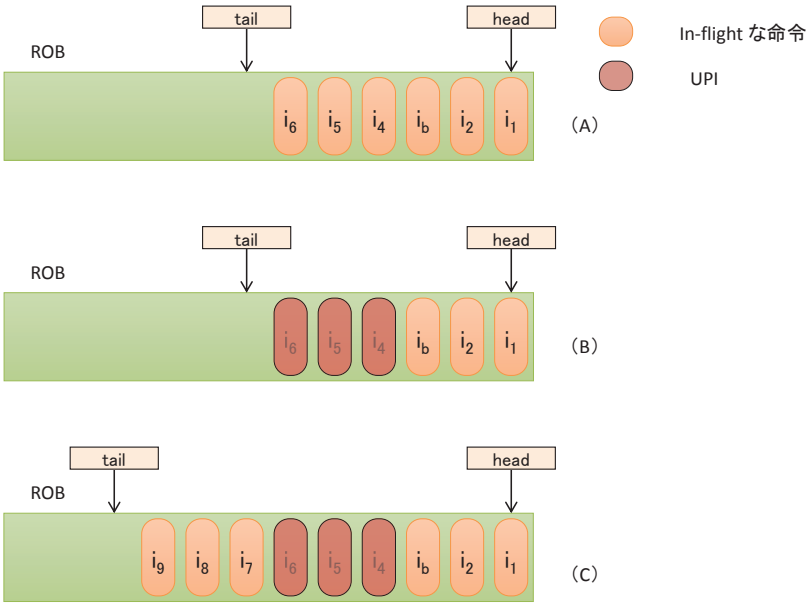

図 5.4: 命令のフラッシュが発生した際の動作:遅延消去

5.2.4 遅延消去による資源圧迫とその軽減

通常，予測ミスした命令はすぐに各種バッファから削除されるため，後続の命令を受け入れるための空きが生じる．しかし，予測ミスした命令を遅延消去する場合，それらの命令，つまり UPI が命令ウィンドウや ROB，LSQ などのバッファを占有する．また，命令ウィンドウに存在する UPI に対してスケジューリングが行われ，不要な演算が行われてしまう．これらの要因により，プロセッサの実行性能が低下する恐れがある．

そこで，擬似コミットするために必要な ROB および LSQ にのみ UPI を残し，命令ウィンドウやパイプライン中の UPI は従来通りに削除することとする．これにより，UPI が命令ウィンドウの資源を圧迫したり，不要な演算が行われることがなくなる．

ここで，通常の命令は，演算が終了しその結果が ROB に格納されるまでコミッ

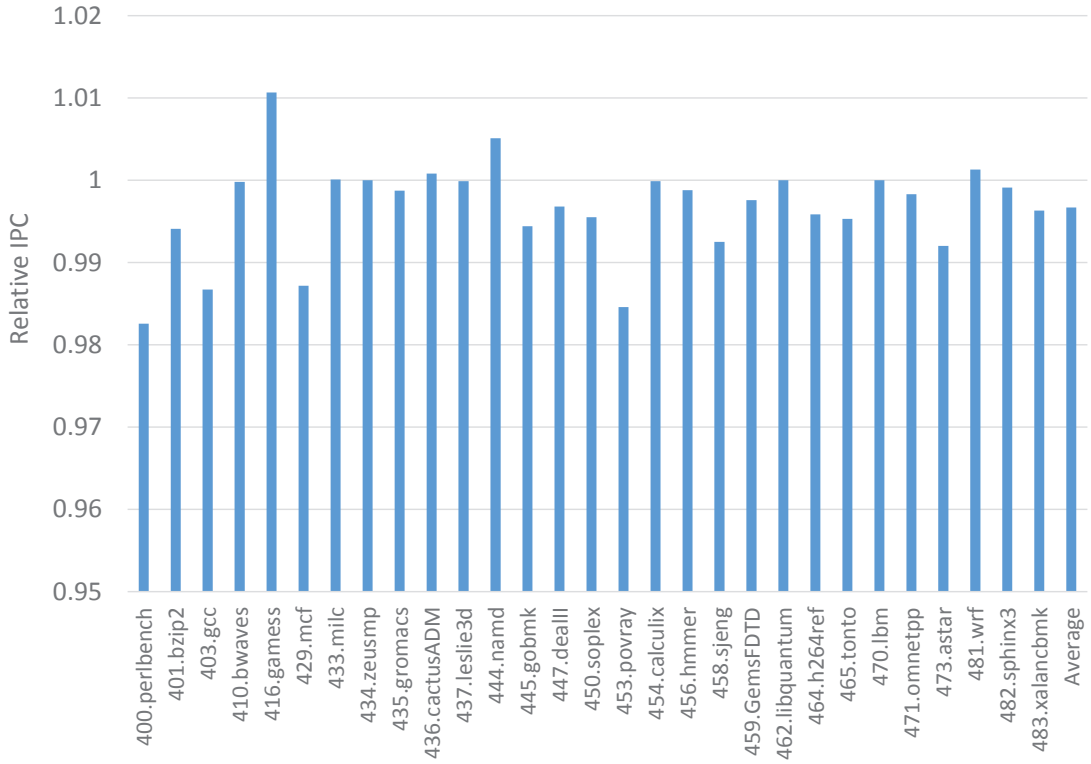

図 5.5: 予測ミスした命令を遅延消去する手法の相対 IPC

ト可能にならない．そこで，UPI はたとえ演算が終了していなくてもコミット可能であるとする．UPI は AS を更新しないため，実行結果が出ていなくても問題は生じない．

ただし，ロード命令に関しては，フィルタをセットしたかどうかによって，擬似コミット時にフィルタをリセットするかどうかを決定する．UPF が立てられたロード命令がフィルタをセットしたかどうかは，そのロード命令のターゲット・アドレスが計算済みで，LQ の対応するエントリに正しいターゲット・アドレスが存在するかどうかによって判断が可能である．

性能への影響

UPI を ROB や LSQ に残すことの影響を調べるため，本節で提案した遅延消去を行う手法をシミュレータに実装し，直ちにフラッシュするモデルとの比較を行った．評価環境，プロセッサの構成及びベンチマークは，後の 6 章で示すものと同様

である．なお，この評価ではPCBFは用いず，いずれもCAMで構成されたLSQで順序違反/フォワーディング・ミス検出を行っている．

図5.5に，ベンチマークごとの，直ちにフラッシュするモデルに対する遅延消去を行う手法の相対IPC (Instruction Per Cycle: サイクルあたりの実行命令数) を示す．平均で99.7%，最悪でも98.3%と，性能への影響は軽微であり，フィルタの一貫性を保つ手法として適当であることがわかる．

5.3 カウンタ機能付き機能メモリを用いたPCBFの構成手法

提案手法では，以下のように，1サイクルのうちに多数の命令が同時にPCBFへアクセスする．

ロード命令の実行 対応するエントリのカウンタをインクリメントする．

ロード命令のコミット 対応するエントリのカウンタをデクリメントする．

ストア命令のコミット 対応するエントリのカウンタが陽性かどうかを確認する

たとえば，6章の評価でもプロセッサ構成の参考としているIntel Haswell [2]では，ロード命令の実行を最大2命令，ロード命令のコミットを最大で8命令行うことができる．この構成に対して提案手法を適用すると，合計10命令が同時にカウンタのインクリメント/デクリメントを行うことになる．

通常，BFは各エントリにカウンタの値を格納するRAMを用いて構成する．RAMに同時アクセスするには，アクセス数に準じたポート数が必要となる．その上，提案手法ではカウント・アップ/ダウンがread-modify-writeとなり，リードのためとライトのために，同時アクセスするロード命令の数の2倍のポート数が必要になる．RAMの面積はポート数の2乗に比例するため，これによる面積増加は深刻である．

そこで本節では，アップ・ダウン・カウンタ機能付きの**機能メモリ** (functional memory) をセルとしたアレイを用いることで，ポート数が同時にアクセスするロード/ストア命令の数に依存しないPCBFの構成を提案する．

以下，5.3.1節でPCBFをRAMで構成した場合について述べ，その問題点を明らかにしたあと，5.3.2節で提案の構成について説明する．その後，5.3.3節で提案

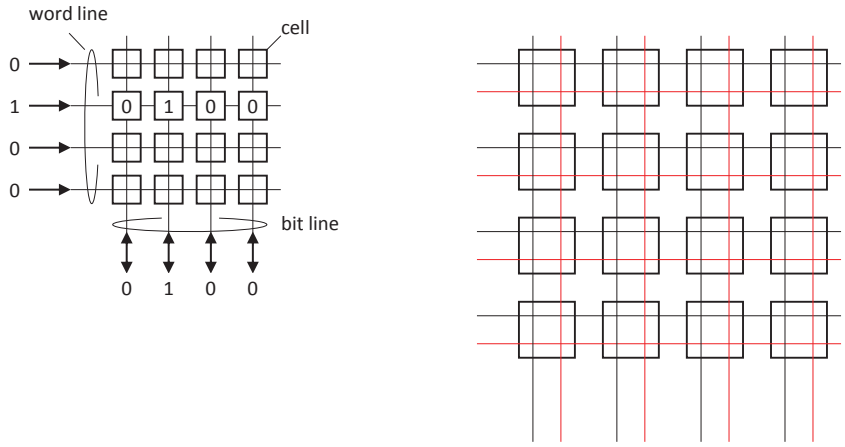

図 5.6: 1 ポート RAM セル・アレイ（左）と 2 ポート RAM セル・アレイ（右）の模式図

の構成を用いた場合のロード/ストア命令の動作について述べる．

5.3.1　PCBF を RAM で構成することの問題点

図 5.6は，1 ポート RAM セル・アレイとと 2 ポート RAM セル・アレイの模式図を表している．RAM セル・アレイは，図のように 1 ビットを記憶するセルが 2 次元のアレイ（配列）として配置され，ワード・ラインとビット・ラインと呼ばれる配線が格子状に接続されている．

RAM セル・アレイへのアクセスは，以下のような手順で行われる．

1. まず，セル・アレイに対し，デコードされたアドレスがワード・ラインの 1 本をアサートすることで行を選択する．
2. リードの場合はビット・ラインに選択されたデータが出力され，ライトの場合はビット・ラインのデータが書き込まれる．

ここで，同時に n カ所をリードもしくはライトしたい場合，基本的には，このワード・ラインとビット・ラインの組を n 組複製することになる．図 5.6の右図で言えば，黒のワード・ライン/ビット・ラインと赤のワード・ライン/ビット・ラインが組となって 2 ポートの RAM セル・アレイを構成している．「RAM の面積はポー

ト数の2乗に比例する」と言われているのは，ポートを増やすためにはこのワード・ラインとビット・ラインの組を複製しなければならないためである．前述のように，RAMで構成されたPCBFに対して最大10命令が同時にカウンタのインクリメント/デクリメントを行うとすると，20ものポート数が必要となり，PCBFの回路面積がその容量の割に非常に大きなものとなってしまう．

5.3.2 カウンタ機能付きメモリを用いたPCBF

上記の問題を解決するため，本論文ではカウンタ機能付き機能メモリを用いてPCBFを構成する手法を提案する．

カウンタ機能付き機能メモリを用いたPCBFの構成

図5.7は，カウンタ機能付き機能メモリを用いたPCBFの図である．図のup/down counterが，カウンタ機能付きメモリであり，すぐ後に示す図5.8に相当する．カウンタを含む，青色で囲まれた部分がPCBFの1エントリである．また，緑で囲まれた部分がPCBFのサブ・アレイを表しており，ロード/ストア命令は全てのサブ・アレイのそれぞれに対して異なるハッシュ関数を用いてアクセスする．ここで，y番目のサブ・アレイのx番目のエントリを(x, y)と表すと，$up_{x,y}$および$down_{x,y}$がインクリメント/デクリメントのための入力，$pos\ chk_{x,y}$が検査のための入力，POSがその検査の結果，$FULL$がPCBFの中で少なくとも1つのカウンタがフルであることを示す出力となっている．

5.3.1節で述べたRAMによるPCBFの構成と比較すると，本構成の各サブ・アレイは，RAMにおいてワード・ラインに相当する配線はエントリあたり3本，ビット・ラインに相当する配線は2本であり，配線数を大幅に削減できることがわかる．また，この本数は同時実行可能なロード/ストア命令の数に依存しない．

カウンタ機能付き機能メモリの詳細

図5.8は，T-FFを用いた3ビットのカウンタ機能付き機能メモリである．図において背景が塗られている部分がT-FFに相当し，それらの出力Q_0，Q_1，Q_2がそれぞれバイナリ・カウンタの1の位，2の位，4の位に相当する．また，入力up，$down$と出力$full$，posが，それぞれ図5.7におけるエントリへの入力$up_{x,y}$および$down_{x,y}$と，アップ・ダウン・カウンタからの出力$full_{x,y}$と$pos_{x,y}$に対応している．

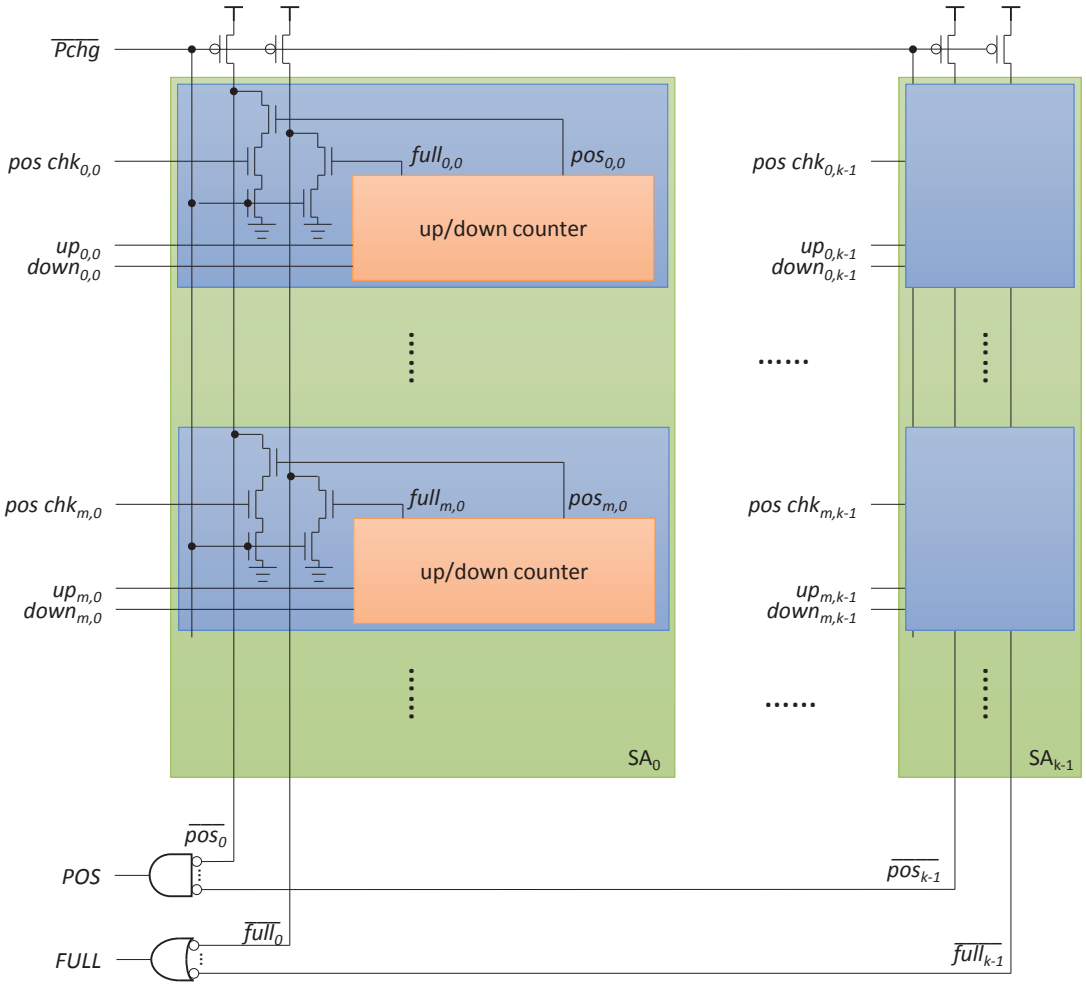

図 5.7: カウンタ機能付き機能メモリを用いた PCBF の構成

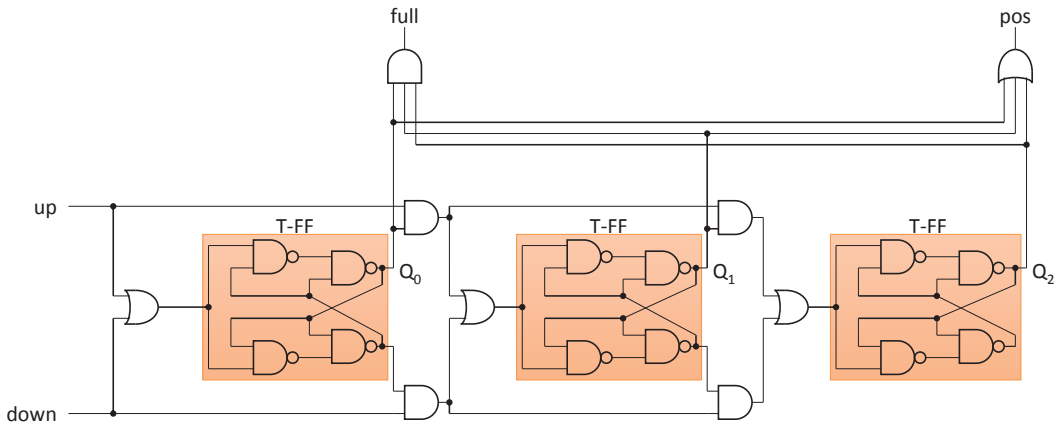

図 5.8: カウンタ機能付き機能メモリの例：3ビットの場合

このカウンタに対し，upのパルスを入力するとインクリメント，downのパルスを入力するとデクリメントが行われる．また，このカウンタはフルであるかどうかと，陽性かどうかの2点がわかればよい．したがって，出力をカウンタの値ではなくこの2点に絞ることで，カウンタのビット数に関係なく出力を2bitとすることができる．

カウンタのフル　ロード命令の実行時に，このカウンタがフルであり，これ以上インクリメントできないことを知る必要がある．カウンタがフルであることはT-FFの全てが1であることに相当するので，T-FFの出力をANDしてやればよい．

カウンタの陽性　「カウンタが陽性である」とは，カウンタが1以上であることである．したがって，ストアのコミット時に陽性かどうかを確かめるには，カウンタが0でないかどうかを確認すれば良い．つまり，T-FFの出力をORし，結果が1であれば陽性であり，そうでなければ陰性である．

提案では，この機能メモリをエントリの構成要素としてPCBFを構成する．

5.3.3 カウンタ機能付きメモリを用いたPCBFの動作

PCBFに対する操作は，主に

1. カウンタのインクリメントおよびデクリメント

2. 陽性の確認
3. フルの確認

の 3 種類に分けられる．以下ではこれらの動作について図 5.7を用いて説明する．

カウンタのインクリメント/デクリメント

PCBF のインクリメント/デクリメントは，ロード命令の実行/コミット時に，ターゲット・アドレスのハッシュをインデクスとして対応するエントリにアクセスすることで行う．このとき，図 5.7のような構成とすれば，サブ・アレイの中で異なるエントリの $up_{x,y}$ または $down_{x,y}$ に対して同時にアクセスできるため，同一サイクルで同一のエントリへのアクセスが発生しない限り，複数のロード命令がそれぞれ対応するエントリに対して同時にインクリメント/デクリメントできる．

ただし，同一サイクルにおいてハッシュ値の衝突が生じ，異なるロード命令が同一エントリに対してインクリメント/デクリメントしようとすることがある．この場合の詳細については，5.4 節で説明する．

陽性の確認

陽性の確認には，**ダイナミック（・プリチャージ）・ロジック (dynamic precharged logic)**[20, 21] を利用する．ダイナミック・ロジックとは，あるノードにプリチャージされた電荷がディスチャージされたかどうかによって出力が決定されるようなロジックである．例えば，図 5.7中のサブ・アレイ SA_0 では $\overline{pos_0}$ がプリチャージ・ノードである．

ダイナミック・ロジックは，プリチャージ期間と評価期間に分かれる．SA_0 を例にとると，まず，プリチャージ期間において，$\overline{Pchg}$ が 0 となり，ノード $\overline{pos_0}$ に電荷がプリチャージされる．次に，評価期間で $\overline{Pchg}$ が 1 となる．ここで，コミットされるストア命令のターゲット・アドレスのハッシュ値が m のとき，$pos\ chk_{m,0}$ がアサートされる．このとき，対応するエントリのカウンタが陽性であれば，$pos_{m,0}$ が high となり，$\overline{pos_0}$ がディスチャージされる．

以上はサブ・アレイのひとつに着目した場合だが，k 個全てのサブ・アレイが 0 であるとき，つまり $\overline{pos_0}$ から $\overline{pos_{k-1}}$ の全てが 0 であるとき，PCBF が実際に陽性となる．よって，それらの NOT の積 POS が，k 個全てのハッシュ値が一致し PCBF が陽性であることを表す．

陽性の確認のためのポート *pos chk* は，同図では簡単のために 1 ポートとしているが，実際には各サブ・アレイにおいて同時にコミット可能なストア命令の数だけ必要となる．この数は，L1D キャッシュの書き込みポート数に一致する．6 章の評価で用いた構成では，同時にコミット可能なストアの命令数は 2 である．

フルの確認

PCBF のエントリを提案のようなカウンタとした場合，インクリメントしようとしたカウンタの 1 つがフルだと手遅れとなってしまう．これは，インクリメントしようとした際にあるサブ・アレイのカウンタのフルが発覚し，そのカウンタのインクリメントは止められても，別のサブ・アレイのエントリはインクリメントされてしまうからである．したがって，事前にインクリメントしようとしているエントリを確認し，どれか 1 つでもフルのカウンタがあるときには，インクリメントしてはならない．

以上の条件を満たすように PCBF のフルを確認する手法として，陽性の確認と同様にダイナミック・ロジックを用いることとする．ただし，陽性の確認の場合のような外部からの入力はなく，例えば SA_0 において 1 エントリでもフルであれば，そのエントリの $full_{y,0}$ の出力が 1 となり，プリチャージ・ノード $\overline{full_0}$ がディスチャージされることでフルであることを確認する．

この実装では，ロード命令がインクリメントしようとしたカウンタかどうかに関係なく，全てのエントリのうちいずれかがフルであることしか確認できない．そのため，フルによるペナルティを削減するためにカウンタのビット数を十分に大きく取る必要があるものの，カウンタの最大値は 1 ビット増やすごとに指数関数的に増加するため，実際に増やさなければならないカウンタのビット数はわずかである．

5.4 PCBF の利用に伴うペナルティの削減手法

提案する PCBF を用いるにあたって，以下の 3 種類のペナルティが発生する．

1. ハッシュの衝突に伴うペナルティ
2. 陽性に伴うペナルティ
3. オーバーフローに伴うペナルティ

本節では，これらのペナルティの発生条件とその対処について説明する．

5.4.1 衝突時の動作

ハッシュが衝突すると，同一サイクルに同一のエントリを複数回インクリメント/デクリメントすることになるが，提案PCBFは1サイクルに1回のみインクリメントまたはデクリメントできる構造になっている．そこで衝突が発生した場合，基本的には，バックエンドをストールさせ，カウンタを順に更新することで対処する．このストールに伴うペナルティは発生頻度が低ければ問題ないが，頻繁に発生する場合は性能に影響するため，衝突の発生率が問題となる．

衝突の発生率

ここで，あるサブ・アレイにおいて，**衝突が発生しない**確率 P_{ncol} は，ハッシュ値が一様に分布している場合，ロード命令のインクリメント/デクリメント数の合計を n，サブ・アレイのエントリ数を m' とすると，以下のように表される：

$$P_{ncol} = \frac{m'}{m'} \times \frac{m'-1}{m'} \times \ldots \frac{m'-(n-1)}{m'} = \frac{m'!}{m'^n(m'-n)!} \tag{5.1}$$

よって，サブ・アレイが k 個のとき，PCBF全体で**少なくとも1ヶ所衝突が発生する**確率 P_{colAll} は以下のようになる：

$$P_{colAll} = 1 - P_{ncol}^k = 1 - \left(\frac{m'!}{m'^n(m'-n)!}\right)^k \tag{5.2}$$

この衝突の発生率 P_{colAll} は，サブ・アレイに対する同時アクセス数 n の増加に伴って非常に大きくなる．例えば $k=4$，$m'=128$ とすると，同時アクセス数 n と衝突率 P_{colAll} は図5.9のような関係となる．6章の評価で使用する構成では，前述のようにロード命令の実行が最大2命令，ロード命令のコミットが最大で8命令同時に行われるため，同時アクセス数は最大で $n=10$ となる．たとえ最大の同時アクセス数である10命令が同時にPCBFにアクセスすることが多くなくとも，衝突によるバックエンドのストールが頻繁に発生し，性能低下を招くことがわかる．この衝突率を下げるには，PCBFのエントリ数 $(m' \times k)$ を増やすか，衝突する可能性のあるアクセスの最大数 (n) を抑える必要がある．

衝突するアクセス数の削減

衝突を回避する方法として，まずロード命令のコミットに着目する．ロード命令のコミット時に同一エントリへのアクセスがあった場合は，同時にコミットされる

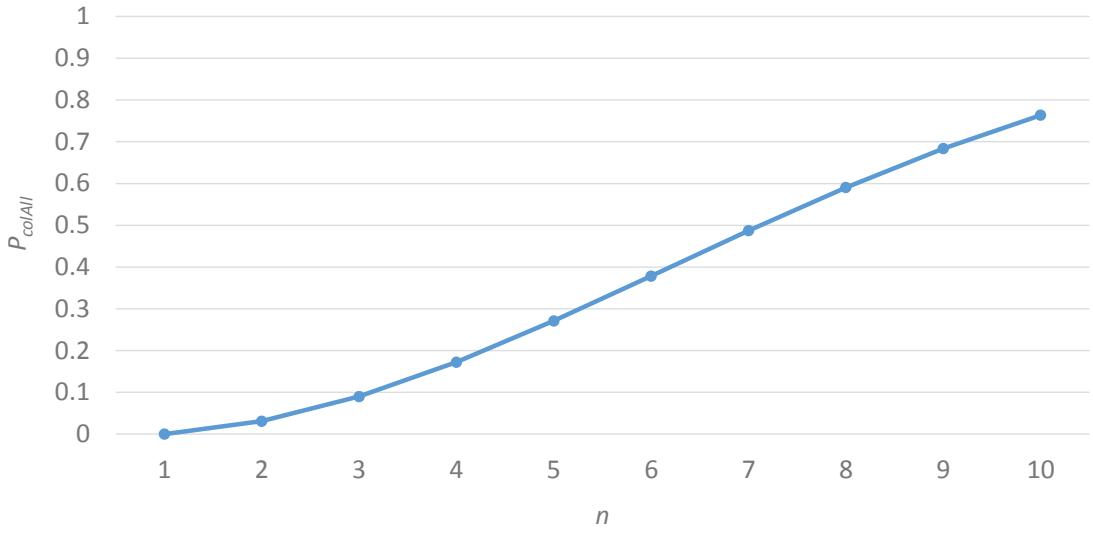

図 5.9: $k = 4$，$m' = 128$ のときの同時アクセス数 n と衝突率 P_{colAll} の関係

命令のグループであるコミット・グループを切り，衝突を起こす後続の命令を次サイクルにコミットする．これにより，後続命令のコミットのタイミングは遅くなるが，コミットするロード命令同士が衝突せず，バックエンドのストールが発生しなくなる．

また，インクリメントとデクリメントが衝突した場合には，対象のカウンタを更新しないことで両方を同時に行ったとみなす．これにより，インクリメントとデクリメントの衝突によるバックエンドのストールが発生しなくなる．

これらの工夫により，ハッシュの衝突によってバックエンドがストールする命令数が最大でロード命令の最大同時実行数（評価では，最大で $n = 2$）となり，衝突率が大幅に削減できる．

5.4.2 陽性時の動作

陽性は，ストアによるリード時に k 個すべてのハッシュ値が一致していると発生する．PCBF が陽性となったとき，実際に順序違反が発生している（真陽性）か否か（偽陽性）を確認検査する．提案手法では，PCBF が低い偽陽性率をもつため，回路面積や消費電力が低い代わりに，時間的コストの高い確認検査の方法を選択することができる．

そのため，提案手法の確認検査は，LQ のシーケンシャル・サーチによって行うこととする．バックエンドをストールし，LQ を検索し，ターゲット・アドレスが一致するロードが存在するかどうか探す．

偽陽性 アドレスが一致するロード命令が見つからなければ偽陽性であるので，何もせずに実行を再開すればよい．

真陽性 実際にアドレスが一致するロード命令が存在すれば真陽性であり，そのロード命令（以降の命令）を再実行する．同時に，Store Set などの依存予測器 [15, 16] にロード/ストア命令の PC を通知し，学習する．

バックエンド・ストール中であるので，LQ のシーケンシャル・サーチには，LQ の発行ポートを用いることができる．6 章の評価では，in-flight なロード命令が最大で 72 命令で，LQ の発行ポートが 2 ポートのとき，確認検査に平均約 18 サイクルかかっている．しかし，PCBF の低い偽陽性率ならば，この程度のペナルティにも耐えることができる．

5.4.3 オーバーフローへの対処

5.3 節で示したように，提案手法では k 個のハッシュに対応するエントリのいずれかではなく，PCBF の全てのエントリのうち 1 個でもフルになっているとオーバーフローすると判断する．したがって，合計のエントリ数が一定ならば k が大きくなるほどオーバーフローが発生しやすくなり，オーバーフローのたびにフラッシュするのは性能低下の要因となる．

しかし，オーバーフローに際して，陽性時のように単にバックエンドをストールすることはできない．当該カウンタをデクリメントするはずのロード命令の実行が停止すると，デッドロックが発生するからである．

オーバーフロー時のペナルティは，以下のようにして軽減することができる．

1. オーバーフローが発生したら，まずバックエンド・パイプランはストールするが，コミット・パイプラインはストールさせない．
2. この状態で，当該カウンタをデクリメントするはずのロード命令が既に実行済みであった場合には，いずれコミットし，カウンタをデクリメントするので，実行を再開することができる．

3. 実行済みのすべての命令がコミットしてもカウンタがフルのままであった場合には，オーバーフローを起こすロード命令と後続の命令をフラッシュし再実行する．

6 章で述べるが，多くの場合フラッシュは必要なく，数サイクル待てば実行を再開することができる．

5.5 ロード/ストア命令のサイズに起因したペナルティの削減手法

4.3 節で述べたように，フィルタを用いた順序違反検出ではサイズの違いに対応するのは容易ではない．我々は，以下の 2 種類の手法を用いることでこの問題に対応する．

1. 4 B や 8 B の命令は，LP64 や LLP64 モデルを用いた多くのプログラムによって利用されている．一方で，1，2 B の命令はあまり使われていない．そこで，4，8 B 命令のための **Main CBF (MCBF)** と，1，2 B 命令のための **Size-aware CBF (SCBF)** の 2 種類の PCBF を用いる．また，1，2 B 命令を振り分けるための **Steering BF (StBF)** も用意する．
2. 2.3 節で述べたように，PCBF は k 個のサブ・アレイに分割される．それら k 個のうちいくつかを 8 B 未満のアクセスを区別するために用いる．

図 5.10に，提案手法の全体構成を示す．

5.5.1 MCBF

MCBF は，主に 4，8 B のロード命令の情報を保持する．MCBF は，$\mathbf{SA_{8B\times1}}$ と $\mathbf{SA_{4B\times2}}$ と呼ぶ 2 種類のサブ・アレイで構成される．$SA_{8B\times1}$ と $SA_{4B\times2}$ はいずれも 8B 単位でアクセスされるが，そのエントリが異なる．$SA_{8B\times1}$ は 8 B 単位のカウンタが 1 つのみだが，$SA_{4B\times2}$ のエントリは 2 つのカウンタがあり，それぞれ 8 B 中の隣接する 4 B に対応している．4 B 命令は $SA_{4B\times2}$ の対応するカウンタのみにアクセスし，8 B 命令は両方のカウンタにアクセスすることにより，4 B アクセスの区別が可能となる．

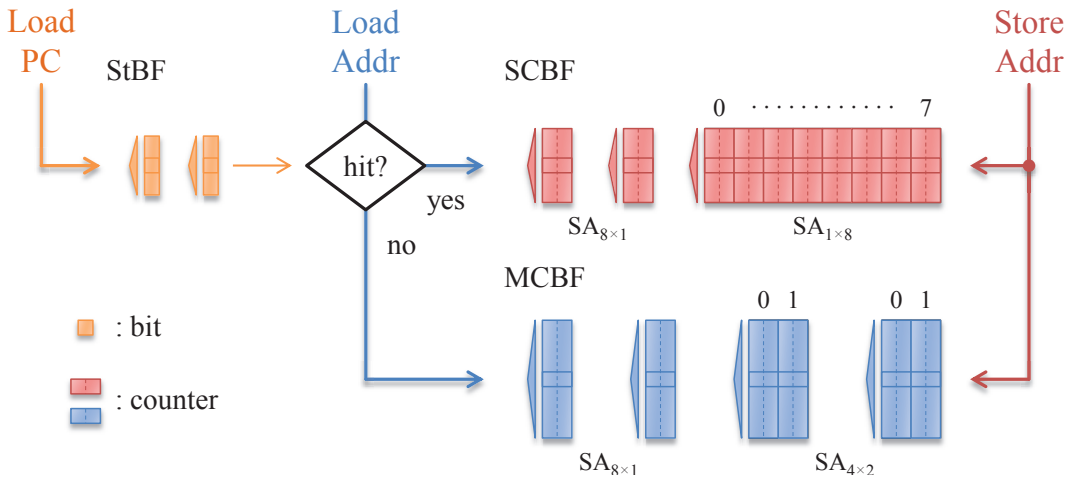

図 5.10: 提案フィルタの構成

サブ・アレイの数

MCBF は $k_{8B\times1}$ 個の $SA_{8B\times1}$ と，$k_{4B\times2}$ 個の $SA_{4B\times2}$ の合わせて $k = k_{8B\times1} + k_{4B\times2}$ 個で構成される．k を一定としたとき，$k_{4B\times2}$ を増やすとサイズによる偽陽性は減少するが，エントリあたりのカウンタが 2 個であるため，面積は増加する．最適な $k_{4B\times2}$ は，6 章で評価している．

5.5.2　SCBF

SCBF は，1 B と 2 B のロード命令の情報のみを保持する．1，2 B のロード命令は少ないため，SCBF も小容量で十分である．SCBF は MCBF と同様に，$SA_{8B\times1}$ と $SA_{1B\times8}$ の 2 種類のサブ・アレイによって構成される．$SA_{1B\times8}$ はエントリが 8 個のカウンタで構成され，8 B 中の対応するカウンタにアクセスする．

5.5.3　StBF

StBF は，ロード命令を振り分ける際に用いる．MCBF や SCBF とは異なり，StBF はカウンタではない通常の PBF である．また，キーはターゲット・アドレスではなく命令の PC である．

ロード命令が PCBF にアクセスする際，まず PC を用いて StBF を参照する．ここで StBF が陽性であれば SCBF を，陰性であれば MCBF を，それぞれインクリメント/デクリメントする．

一方で，ストア命令のアクセス順序違反検出時には StBF を用いず，MCBF と SCBF の両方にアクセスし，いずれか一方でも陽性であれば PCBF が陽性であるとみなす．これは，ストア命令とそれに依存するロード命令の PC が異なるためである．また，ストア命令専用の StBF は用いることができない．これは，StBF が偽陽性であると PCBF が偽陰性を示してしまうためである．

StBF への PC の追加は，MCBF が陽性を示し，さらに確認検査で 4 B 中の異なるアドレスに対するアクセスが原因の偽陽性であると判明した際に行われる．このようにすることで，上記のような偽陽性を起こさない 1，2 B 命令を MCBF で管理することができ，SCBF をさらに小さくすることができる．

StBF のリセットは，一定のサイクル —— 6 章の評価では，128K サイクルが経過するごとに全ビットをフラッシュすることによって行う．

第6章

性能評価

提案手法と既存手法について，主にフィルタのサイズとIPCについての評価を行った．また，提案手法については回路面積と消費エネルギーについても評価した．以下，6.1 節で評価環境についてまとめた後，6.2 節で提案手法の構成について検討する．その後，6.3 節で既存手法との比較を行い，最後に 6.4 節で提案手法の回路面積と消費エネルギーについて評価する．

6.1 評価環境

ベンチマーク

ベンチマークはSPEC CPU 2006 [22] の全29プログラムで，データ・セットは *ref* を使用し，各プログラムは gcc 4.6.1 の -O3 でコンパイルした．評価は最初の 1G 命令をスキップし，直後の 100M 命令をシミュレートする．

シミュレータ

シミュレーションにはcycle-accurateなプロセッサ・シミュレータである鬼斬弐 [23, 24] を用いた．ベースラインとなるプロセッサの構成は表 6.1に示す通りである．各パラメタは，IBM POWER7 [1] や Intel Haswell [2] など，最近のハイエンド・プロセッサを参考にしている．

なお，命令セットはAlphaで，拡張命令セットとしてbyte-word extensionsを適用している．そのため，1 B，2 B のロード/ストア命令が出現する．

また，依存予測器としては，Store Set [15] を用いた．

6.2　提案手法の構成の検討

本節では，提案手法の以下の効果を評価する：

1. フィルタの容量に対する k および c の効果
2. サイズを考慮した PCBF の効果
3. IPC 低下の内訳

表 6.2に，提案手法のデフォルトのパラメタを示す．m' はサブ・アレイあたりの容量，c はカウンタのビット数，b はエントリあたりのカウンタの数，k はサブ・アレイの数である．以降，特に断りのない場合には，表に示す値を用いた．

以下では，偽陽性率とベースラインに対する相対 IPC の，ベンチマークのクラスごとの平均を示す．ベースラインは，CAM を用いて順序違反/フォワーディング・ミス検出を行うモデルで，(例外的な条件を満たす場合を覗いて）フィルタを用いた手法のような偽陽性による性能低下はない．また，予測ミスした命令を直ちにフラッシュする手法をベースラインとして用い，PCBF は 5.2 節で提案した，予測ミスした命令を取り除かない手法を用いる．いくつかのグラフを示すが，特に断りがない限り，横軸はフィルタの総ビット数で，縦軸は偽陽性率，もしくは，ベースラインに対する相対 IPC である．偽陽性率のグラフでは左下にある曲線ほど，相対 IPC のグラフでは左上にある曲線ほど，性能がよいことになる．

6.2.1　フィルタの容量に対する k および c の効果

k の効果

2.2 節では，ハッシュ関数の数 k をわずかに増加させることで陽性率を劇的に減少させることができることを解析的に明らかにした．ここでは，シミュレーションで実際に偽陽性率が減少し，IPC の低下が抑えられることを示す．

図 6.1に，結果を示す．曲線はそれぞれ $k = 1, 2, \ldots, 5$ の場合で，m' を変化させたものである．また，4.3 節で述べたサイズの問題を分離するため，フィルタは k 個の $\mathrm{SA}_{4\mathrm{B}\times 2}$ で構成された MCBF のみを用いた．

$k = 1$ の場合と比べ，$k \geq 2$ の場合に，偽陽性率が劇的に減少し，相対 IPC の低下も低く抑えられている．$k = 4$，$m' = 128$ の場合に良い結果が得られているため，以降ではこれらのパラメータを用いる．

表 6.1: プロセッサの構成

パラメタ	値
ISA	Alpha 21264A w/ byte-word ext.
fetch/issue/commit width	8/8/8 inst./cycle
instruction window	64 entries unified
ROB	192 entries
LQ/SQ	72/42 entries
branch predictor	16KB:g-share/8K:local hybrid
branch miss penalty	15 cycles
BTB	2K-entry, 4-way
L1D	64KB, 8-way, 64B/line, 2 cycles
L2C	512KB, 8-way, 64B/line, 8 cycles
L3C	8MB, 8-way, 64B/line, 24 cycles
main memory	200 cycles

表 6.2: PCBF のパラメータ

BF		$m' \times c \times b \times k = \text{total}$
MCBF	$\text{SA}_{8\text{B}\times1}$	$128 \times 4 \times 1 \times 2 = 1024$
	$\text{SA}_{4\text{B}\times2}$	$128 \times 4 \times 2 \times 2 = 2048$
SCBF	$\text{SA}_{8\text{B}\times1}$	$16 \times 4 \times 1 \times 2 = 128$
	$\text{SA}_{1\text{B}\times8}$	$8 \times 3 \times 8 \times 1 = 192$
StBF	—	$64 \times 1 \times 1 \times 2 = 128$
total		3520

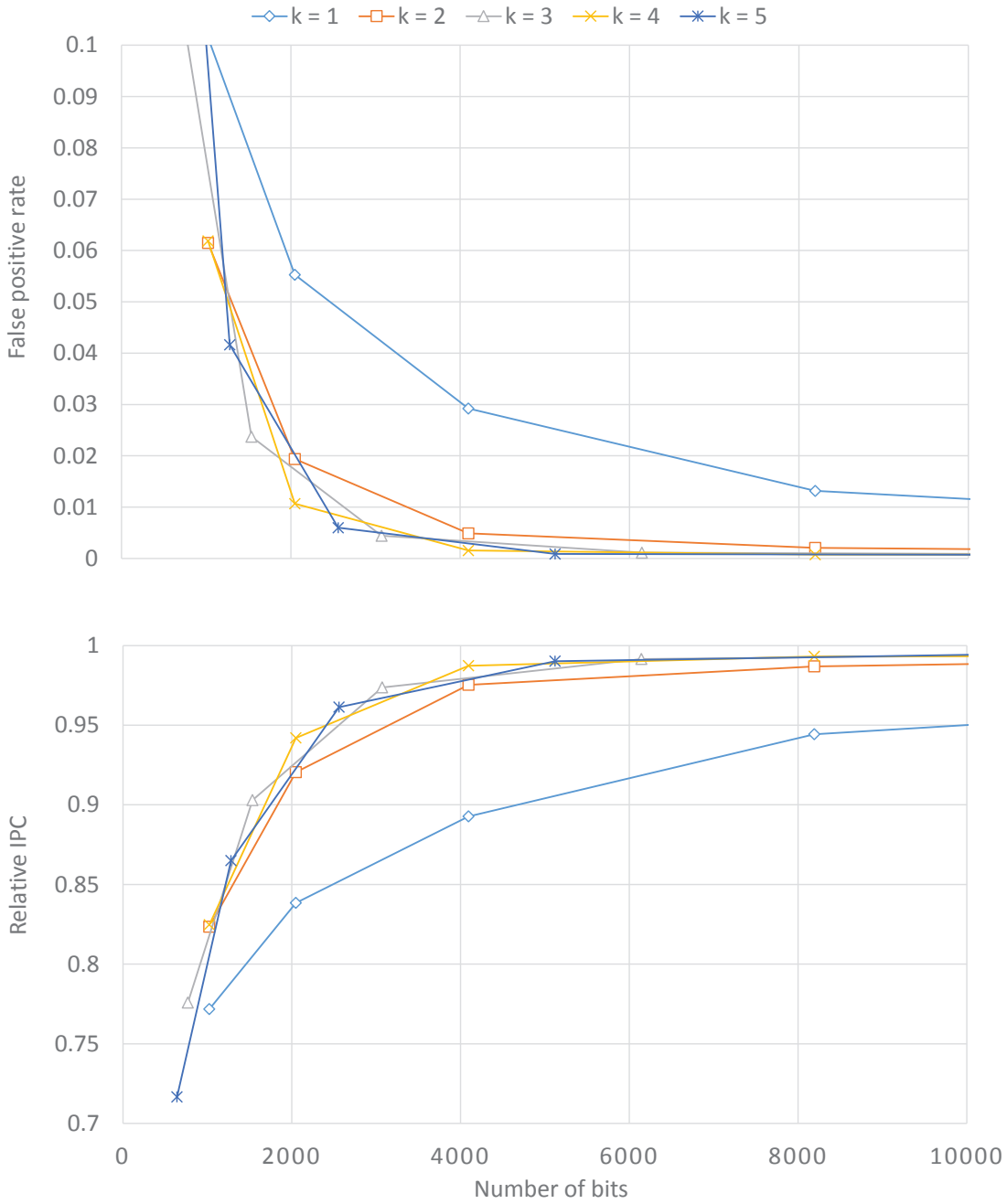

図 6.1: k の効果 偽陽性率（上）と相対 IPC（下）

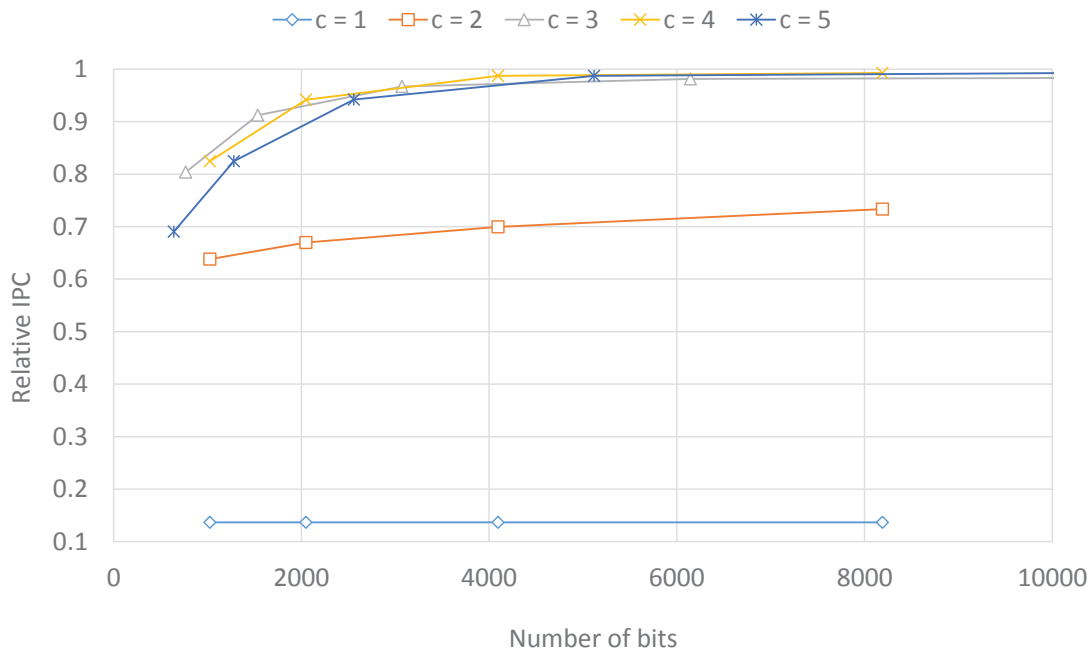

図 6.2: c の効果 相対 IPC

c の効果

ここでは，m' を増加させるよりも c をわずかに増加させるほうがカウンタのオーバーフローを効率的に回避できることを示す．

図 6.2に結果を示す．曲線は，$c = 1, \ldots, 4$ ごとに m' を変化させたものである．また図 6.1と同様に，評価には k 個の $\mathrm{SA}_{4\mathrm{B}\times 2}$ で構成された MCBF のみを用いている．

$c = 1, 2$ の場合は，オーバーフローによる性能低下が大きく，m' を増やしてもその効果は小さい．一方で，$c \geq 3$ では IPC が改善している．特に $c = 4$ の場合にベースラインとほぼ同等の IPC を達成していることがわかる．したがって，以降 $c = 4$ に固定する．

6.2.2 サイズを考慮した PCBF の効果

MCBF の効果

ここでは，5.5 節で提案した，ロード/ストア命令のサイズに対応した PCBF の効果について評価する．

図 6.3は，MCBF の k を固定し，$\mathrm{SA}_{8\mathrm{B}\times 1}$ と $\mathrm{SA}_{4\mathrm{B}\times 2}$ の数を変化させたときの平均相

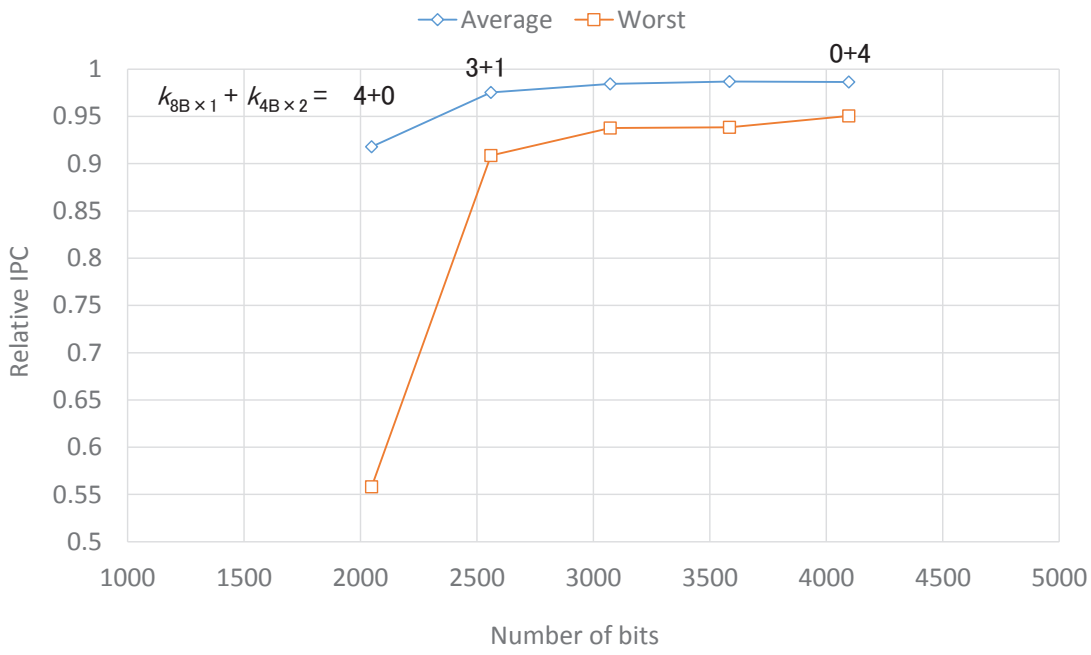

図 6.3: MCBF の構成 相対 IPC

対 IPC と最悪相対 IPC である．図では，サブ・アレイの数を $k = 4 = k_{8\mathrm{B}\times1}+k_{4\mathrm{B}\times2} = 4+0, 3+1, 2+2, 1+3, 0+4$ と変化させている．$\mathrm{SA}_{8\mathrm{B}\times1}$ を $\mathrm{SA}_{4\mathrm{B}\times2}$ に置き換えるたびにエントリあたりのカウンタ数が増加するため，合計のビット数も増加する．

図の結果では，$k_{8\mathrm{B}\times1}+k_{4\mathrm{B}\times2} = 4+0$ の場合に IPC が大きく低下している．一方で，$k_{8\mathrm{B}\times1}+k_{4\mathrm{B}\times2} = 3+1$ 以降の構成では相対 IPC が大きく改善している．これは，4 B のロード/ストア命令が多くのプログラムで広く利用されており，フィルタを用いた手法においてはサイズの問題に対処することが重要であることを示している．評価によると，MCBF の構成が $k_{8\mathrm{B}\times1}+k_{4\mathrm{B}\times2} = 2+2$ の場合で十分であるので，以降この構成を用いる．

SCBF の効果

5.5 節で示したように，SCBF は 1, 2 B のロード/ストア命令によるサイズの問題が発生するプログラムの性能を改善するために用いる．そこで，SCBF の効果を測るために SPEC CPU 2006 の中でこの問題が発生するプログラムである gcc, gobmk, h264ref, astar, xalancbmk の 5 種類のベンチマークを評価に用いた．

図 6.4にその評価結果を示す．点で示された “Average: MCBF” および “Worst:

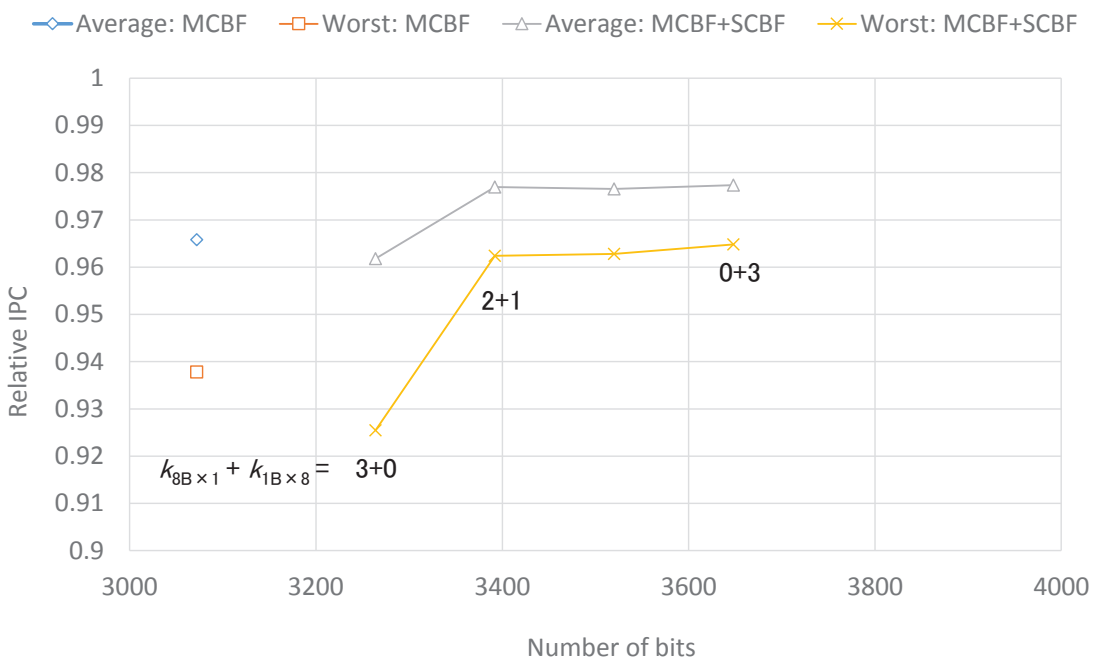

図 6.4: SCBF の構成 相対 IPC

MCBF” が SCBF を用いなかった場合の IPC の平均値と最悪値であり，曲線で結ばれた “Average: MCBF+SCBF” および “Worst: MCBF+SCBF” が SCBF をそれぞれ異なる構成 $k = 3 = k_{8\mathrm{B}\times1} + k_{1\mathrm{B}\times8} = 3+0, 2+1, 1+2, 0+3$ で用いた場合の IPC の平均値と最悪値である．図 6.3と同様に，$\mathrm{SA}_{8\mathrm{B}\times1}$ が $\mathrm{SA}_{1\mathrm{B}\times8}$ に置き換わるたびにカウンタ数が増加するため，$k_{1\mathrm{B}\times8}$ を大きくするとビット数も増加する．

評価では，$k_{8\mathrm{B}\times1} + k_{1\mathrm{B}\times8} = 3+0$ のときに，SCBF を用いなかった場合よりも性能が低下している．これは，$k_{1\mathrm{B}\times8}$ が存在しないため 1, 2 B のロード/ストア命令によるサイズの問題が解決できないばかりか，SCBF はエントリ数が少ないために偽陽性が頻発しているためである．一方で，$k_{8\mathrm{B}\times1} + k_{1\mathrm{B}\times8} = 2+1$ 以降では SCBF を用いない場合よりも性能が改善しており，SCBF の効果が発揮されていることがわかる．評価により，SCBF の構成は $k_{8\mathrm{B}\times1} + k_{1\mathrm{B}\times8} = 2+1$ の場合で十分であることがわかった．

これまでの評価結果に基づき，以降の評価では，表 6.2の構成を用いることとする．

6.2.3　IPC 低下の内訳

オーバーフロー時のストールによる効果

5.4.3 節で述べたように，オーバーフロー時にはすぐにフラッシュするのではなく，バックエンドをストールさせ，カウンタのデクリメントを待ってからフラッシュした方がよい．これを示すため，我々は 5.4.3 節で述べた手法と，オーバーフロー時にすぐにフラッシュする手法を評価・比較した．ここで，フラッシュに伴う正確なペナルティを測定することは困難であるため，フラッシュ発生時のペナルティを 15 サイクルとし，フラッシュ数に乗算し，ストール・サイクルに加算することで合計のペナルティを算出した．

図 6.5に評価結果を示す．図はベンチマークごとのペナルティの合計サイクル数を表し，グラフの左がすぐにフラッシュするモデル，右がストールによりフラッシュを可能な限り回避するモデルである．ここで，ストールするモデルには，ストールしたサイクル数の他，カウンタのデクリメントが行われなかった場合のフラッシュ・ペナルティも含まれている．

評価結果をみると，ストールするモデルでは，特にペナルティの大きい leslie3d において，フラッシュするモデルに比べてペナルティが約 1/6 になっていることがわかる．

ベンチマークごとのペナルティと相対 IPC

ここでは，提案手法を用いることによる性能低下の要因と，その影響を示す．

図 6.6（上）は，プログラムごとの性能低下要因の内訳を示しており，陽性，オーバーフロー，衝突によるペナルティをそれぞれ積み上げ，ペナルティのサイクル数の合計を表している．真陽性時のフラッシュによるペナルティは，オーバーフローと同様に 15 サイクルとし，フラッシュ数に乗算してストール・サイクルに加算している．

また，図 6.6（下）にベンチマークごとのベースラインに対する相対 IPC を示す．これらの図を比較すると，相対 IPC はペナルティと強く相関していることがわかる．しかし，その原因は様々である．たとえば，perlbench や gromacs，lbm，astar は衝突によるペナルティが原因で IPC が低下している一方，gamess や zeusmp は陽性によって IPC が低下している．また，h264ref と tonto は衝突と陽性の両方が原因となって IPC が低下している．

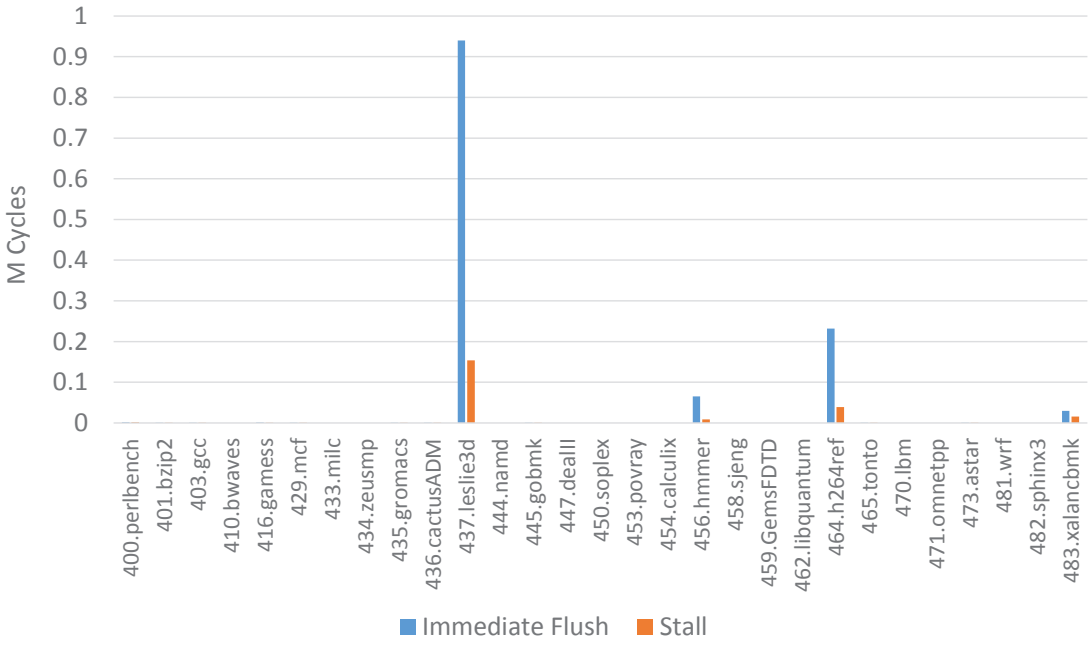

図 6.5: オーバーフロー時のペナルティ

また，dealII と GemsFDTD では，ベースラインの CAM を用いた手法よりも IPC が向上している．これは，CAM を用いた LSQ と Store Set を組み合わせた場合特有の偽陽性が発生しているためである．

6.2.4 CAM を用いた LSQ と Store Set を組み合わせたときの偽陽性

図 6.7に，CAM を用いた LSQ と Store Set を組み合わせた際に特有の偽陽性が発生しているパイプライン図を示す．図において，st_0，st_1，ld のいずれも同一のアドレス a0 にアクセスするとする．ここで，図 6.7（上）のように $st_1 \rightarrow ld \rightarrow st_0$ の順で実行されるとき，st_1 から ld へストア・データ 401 が正しくフォワーディングされる．しかし，フォワーディング後に st_0 が実行されると，ld が実行済みであるため，アクセス順序違反として検出され st_0 と ld に依存関係があるとして Store Set に学習される．すると，次回以降の実行時に命令の実行順序が図 6.7（下）のように $st_0 \rightarrow st_1 \rightarrow ld$ となり，ld に依存する後続の命令の実行が遅れ性能低下につながる．

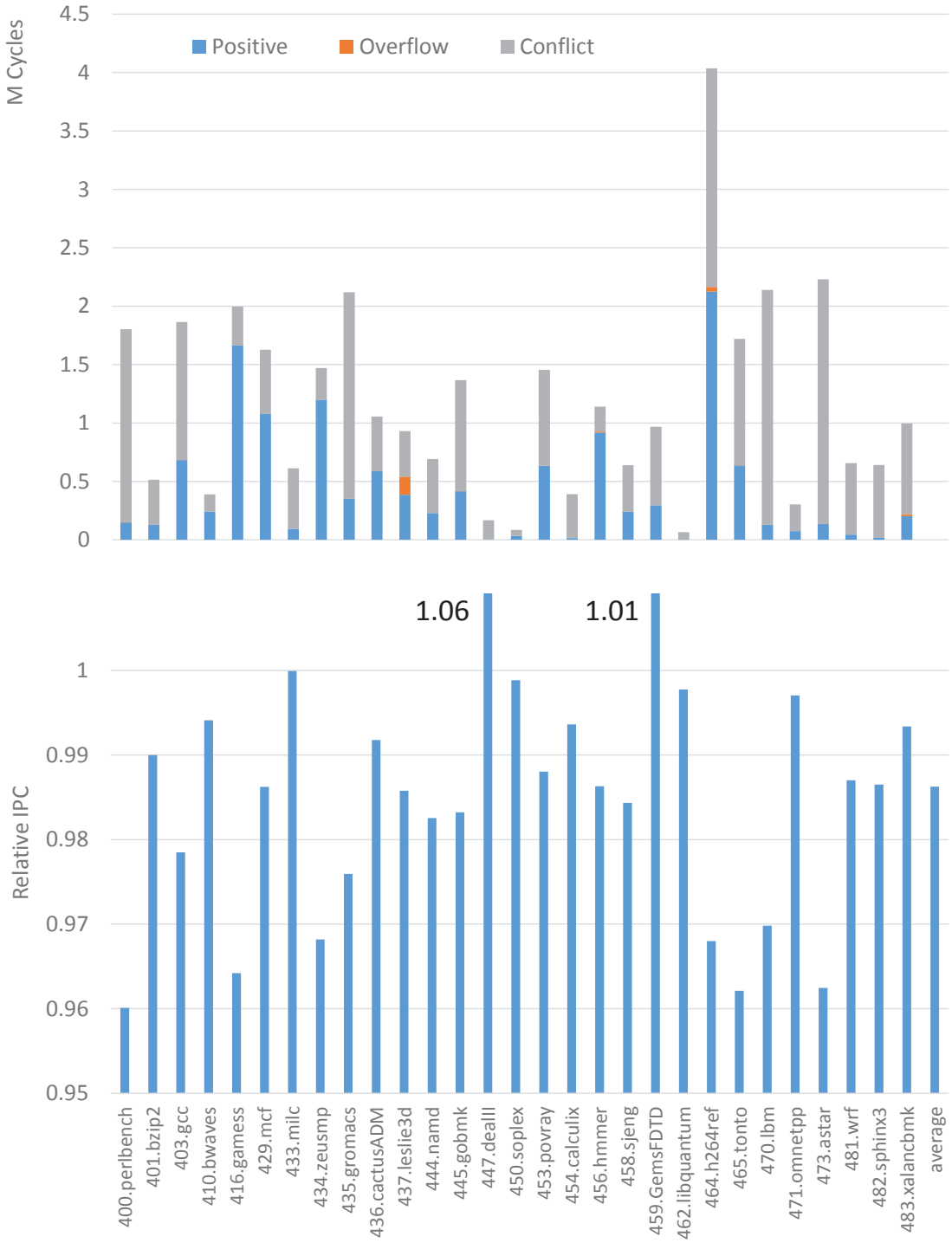

図 6.6: PCBF によるペナルティの内訳（上）と相対 IPC（下）

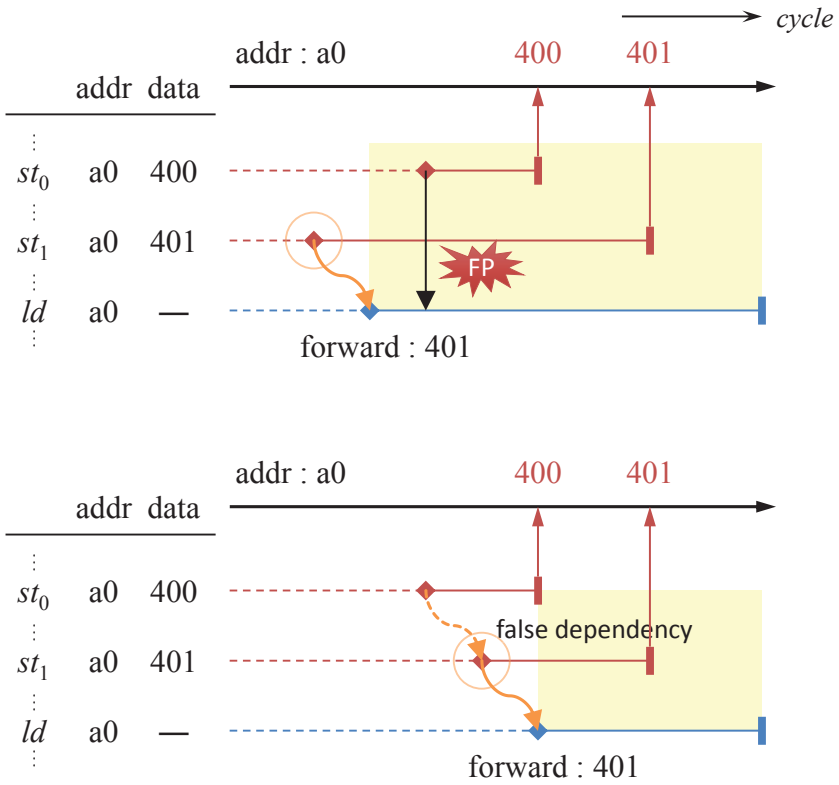

図 6.7: CAM を用いた手法におけるフォワーディング時の偽陽性（上）と，
Store Set による学習後の動作（下）

このような偽陽性は，提案手法では発生しない．図 6.8に，提案手法で図 6.7（上）と同様の実行順序となる状態が発生したときのパイプライン図を表す．st_1 から ld へストア・データがフォワーディングされると，フィルタのセットはフォワーディング元の st_1 のコミット時に行われる．フィルタをセットするときには，st_0 はコミット，つまり順序違反の検出を行った後なので，CAM を用いた LSQ で発生するような偽陽性は起こらないのである．図 6.6において dealII と GemsFDTD が CAM を用いた手法を超える性能を達成しているのは，このような状態が多く発生しているためであると考えられる．

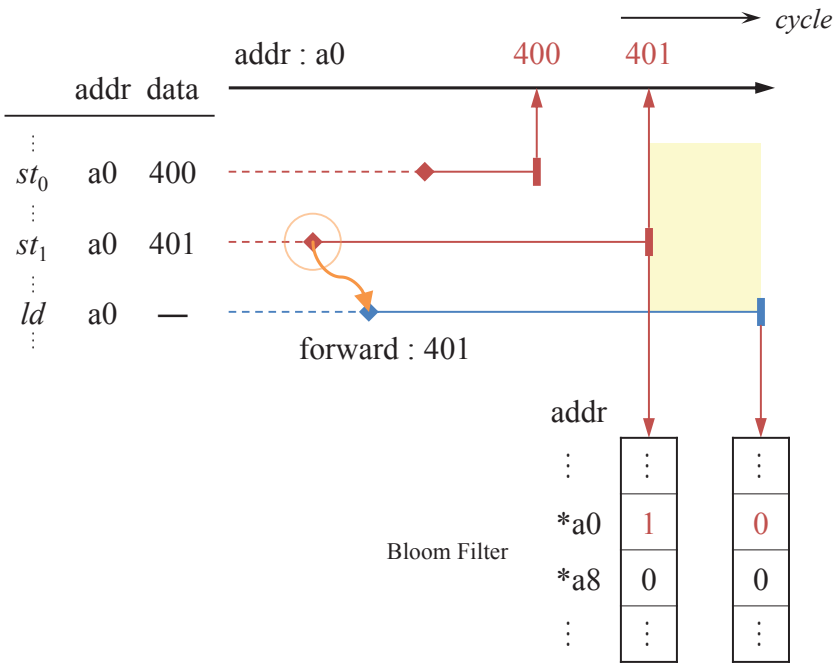

図 6.8: 提案手法におけるフォワーディング時の動作

6.3　既存手法との IPC の比較

ここでは，既存手法と，サイズと IPC との関係を比較する．5.5 節で述べたロード/ストアのサイズの問題に対しては，各手法の認識はまちまちである．文献 [7, 6] ではわずかに言及があるものの詳細は不明であり，その他の文献では言及がない．そこで本節では，既存手法のフィルタは 8 B ワードを単位としている．ただし，提案手法の総ビット数には，SCBF と StBF を含んでいる．

6.3.1　評価モデル

表 6.3に，評価したモデルを示す．ただし，既存手法の網掛けの部分は以下のように理想化されている：

- SQ-CAM/LQ-CAM は，SQ/LQ がそれぞれ CAM で構成されていることを表

す．後に示すグラフの横軸はフィルタのビット数であり，CAM による面積増はグラフには反映されてない．

- LQ-CAM による確認検査は，1 サイクルで可能である．
- ロード再実行は，陽性が検出された直後の 1 サイクルで可能である．

既存手法のパラメタは，各文献で示された値とした．ただし，SPCT（Store PC Table）のエントリ数は，文献 [6] には明記されていないので，我々の事前の評価で最も高い IPC を示した 16 とした．

6.3.2 評価結果

図 6.9の結果から，以下のことが分かる：

- SVW は効率が非常に悪く，IPC を維持するには大きな容量が必要となる．これは，テーブルのエントリがシーケンス・ナンバであり，ビット・テーブルと比較してテーブルの利用効率が悪いのに加えて，複数のハッシュ関数を用いられず偽陽性率が高くなるためである．
- DMDC は本来陽性が偽であるか真であるかを区別できないが，シミュレータでは確認が可能なため，それに基づいて偽陽性率を算出している．その結果，DMDC の偽陽性率は，提案手法と同等であった．しかし，依存予測器を用いないため，偽陽性に加えて真陽性も頻発するので IPC は大きく低下している．この結果は，Store Set などの依存予測器に関する研究における optimistic モデルの評価結果と整合的である [15, 16]．
- SHF は，偽陽性率に関しては提案手法の $k = 1$ の場合とほぼ同一となる．ただ

表 6.3: 評価モデル

	Filter	Forwarding	Confirmatory Test	Predictor Learning	False Positive Stall Cycles
Baseline	—	SQ CAM	N/A	LQ CAM	0
SVW	SVW	Speculative	Load Re-Exec	SPCT	1
DMDC	DMDC	SQ CAM	N/A		(Pipeline Flush)
SHF	CBF ($k = 1$)		LQ CAM		1
PCBF	PCBF ($k \geq 2$)	Speculative	LQ Sequential Search		16.9 (avg)

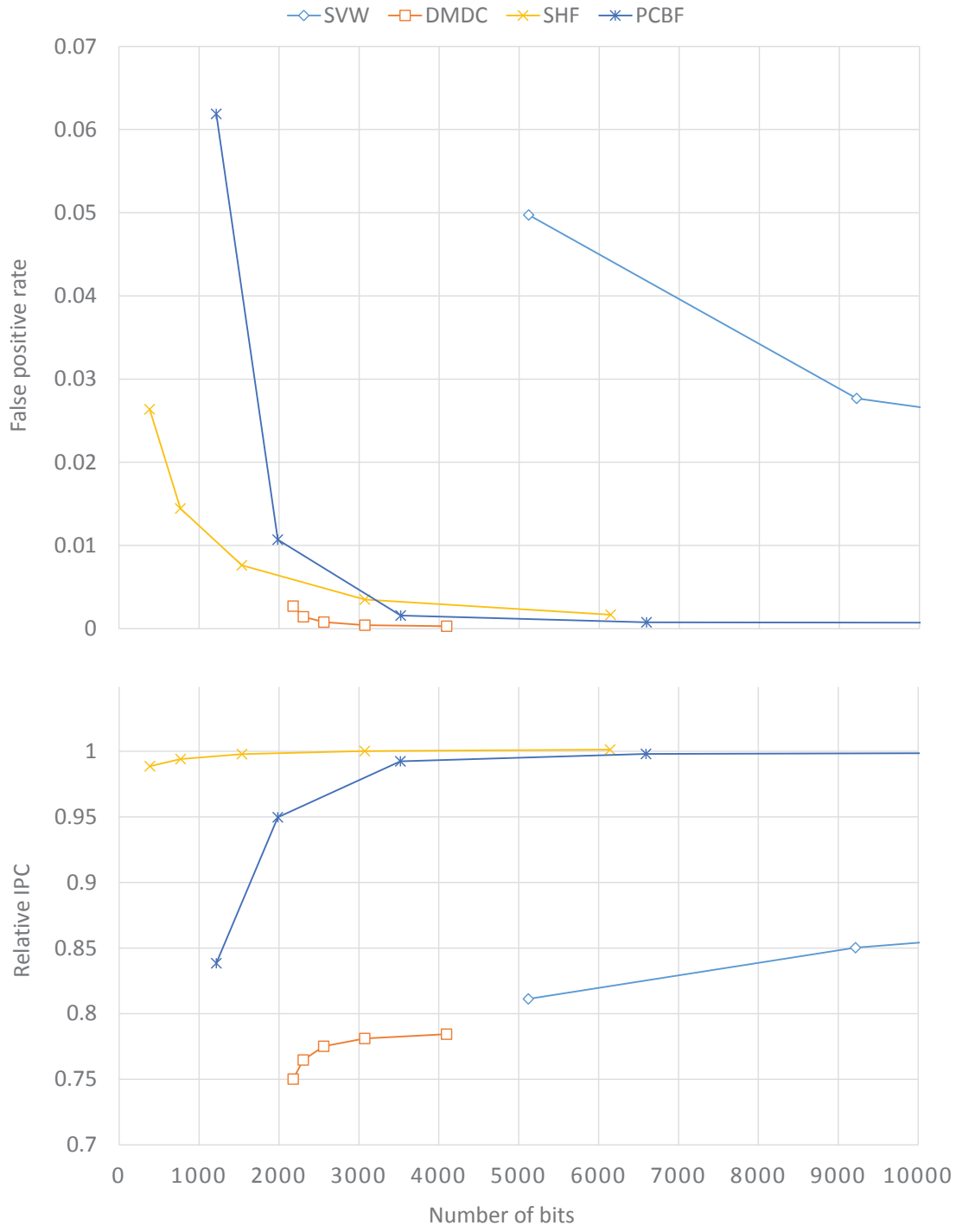

図 6.9: 表 6.3の評価モデル 偽陽性率（上）と相対 IPC（下）

しSHFは，LQ-CAMによって1サイクルで確認検査が行われるため，シーケンシャル・サーチを行う提案手法と比較すると，IPCが同等になっている．逆に言えば，提案手法はLQ-CAMを省略しながら，低い偽陽性率によってLQ-CAMを用いた場合と同等のIPCを達成していると言える．

6.4 提案手法の回路面積と消費エネルギーの評価

ここでは，LSQの回路面積と消費エネルギーを，表6.4で示す各モデルについて評価する．評価ツールには，CACTI 5.3 [25] ITRS 45nm を利用した．

6.4.1 評価モデル

Non-spec fwd w/ CAM ベースラインとなる，CAMで構成されたLSQを用いて順序違反/フォワーディング・ミス検出を行うモデル．

Non-spec fwd w/ Filtered CAM Non-specfwd w/ CAM のLQに $k = 1$ のフィルタを追加したモデル．SHFはこのモデルに相当する．

Spec-fwd w/ CAM Non-specfwd w/ CAM で投機的フォワーディングを行うモデル．フォワーディングにCAMを用いないが，順序違反/フォワーディング・ミス検出はCAMで構成されたLQを用いる．

Spec-fwd w/ Filtered CAM Spec-fwd w/ CAM のLQに $k = 1$ のフィルタを追加したモデル．SHFに投機的フォワーディングを実装したモデルに相当する．

Spec-fwd w/ Filtered RAM Spec-fwdモデルに提案フィルタを追加し，LQをRAMで構成したモデル．提案手法．

なお，Spec-fwd w/ Filtered CAMはフィルタのエントリ数をパラメータとして複数のモデルを用意し，Spec-fwd w/ Filtered RAMについてはフィルタをRAMで構成したモデルと5.3節で提案したカウンタで構成したモデルの2種類を評価した．また，提案カウンタのモデルに関しては，FreePDK45nm[26] セル・ライブラリを用いて面積を計算し，その値をRAMのセル面積としてCACTIに代入することで見積もっている．

6.4.2　評価結果

回路面積

回路面積の評価結果を図 6.10に示す．各モデルで SQ，LQ，フィルタの回路面積を積み上げ，Non-spec fwd w/ CAM モデルの合計回路面積で正規化した相対回路面積で表している．なお，参考として，IPC の評価でも用いた Intel Haswell[2] の L1D キャッシュを想定した 32K エントリ，8-way のキャッシュを右端に図示する．

Non-spec fwd の各モデルと比べ，Spec-fwd の各モデルは SQ の回路面積が大きく削減されている．これは，投機的フォワーディングを行うことで SQ が CAM ではなく RAM で構成されるようになったためである．同様に，Spec-fwd w/ Filtered RAM の LQ の面積が Spec-fwd w/ Filtered CAM と比べて大きく削減されているのは，LQ が CAM から RAM で構成されるようになったためである．

さらに，提案フィルタをカウンタで構成することで，RAM で構成するよりも大幅に回路面積が削減できている．最終的には，Non-spec fwd w/ CAM と比較して，Spec-fwd w/ Filtered RAM のフィルタをカウンタで構成したモデルは 20.3%まで削減されている．また，Non-spec fwd w/ CAM では LSQ の面積がキャッシュの面積に対して 84.0%もの大きさを占めていたのに対し，Spec-fwd w/ Filtered RAM ではフィルタを含めても 17.1%まで削減されている．

表 6.4: 回路面積と消費エネルギーの評価モデル

model name	Issuing	Forwarding	remarks
Non-spec fwd w/ CAM	speculative CAM	non-spec N/A	spec fwd を行わず，CAM によって fwd. 順序違反検出も CAM.
Non-spec fwd w/ Filtered CAM	speculative Filtered CAM	↑	順序違反検出の CAM にフィルタを追加. SHF.
Spec-fwd w/ CAM	speculative CAM		spec fwd を行う. 順序違反・fwd ミス検出はCAM.
Spec-fwd w/ Filtered CAM	speculative Filtered CAM		順序違反・fwd ミス検出のための CAM にフィルタを追加
Spec-fwd w/ Filtered RAM	speculative Filtered RAM		CAM は用いず，RAM によって確認検査. 提案.

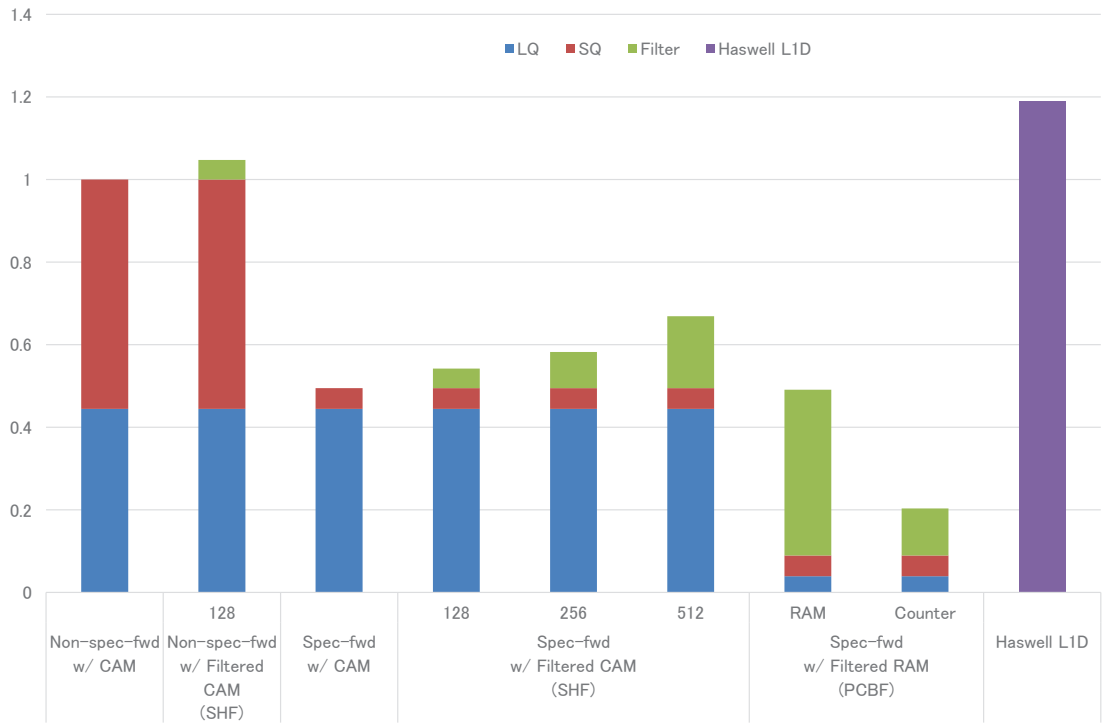

図 6.10: 表 6.4の各モデルと L1D キャッシュの相対回路面積

消費エネルギー

各モデルにおける，SPEC CPU2006 の全プログラムの平均消費エネルギーを図 6.11に示す．表 6.5に示す各操作で生じる消費エネルギーを積み上げ，Non-spec fwd w/ CAM モデルの平均消費エネルギーで正規化した平均相対消費エネルギーで表している．

まず，Non-spec fwd w/ CAM では，順序違反検出/フォワーディングのための LQ/SQ サーチの消費エネルギーが支配的であることがわかる．

Non-spec fwd w/ Filtered CAM では，順序違反検出のための LQ サーチを SHF によってフィルタリングすることにより，そのエネルギーを削減している．しかし，フォワーディングのための SQ サーチを排除できていないため，全体としては効果が薄い．

Spec-fwd w/ CAM では，投機的フォワーディングを行っている．つまり，CAM で構成された SQ をサーチするかわりに，RAM で構成された SQ をリードするため，SQ の消費エネルギーは大きく削減されている．また，SQ の CAM を RAM としたことにより，SQ のライトとリードのエネルギーがそれぞれ削減されている．

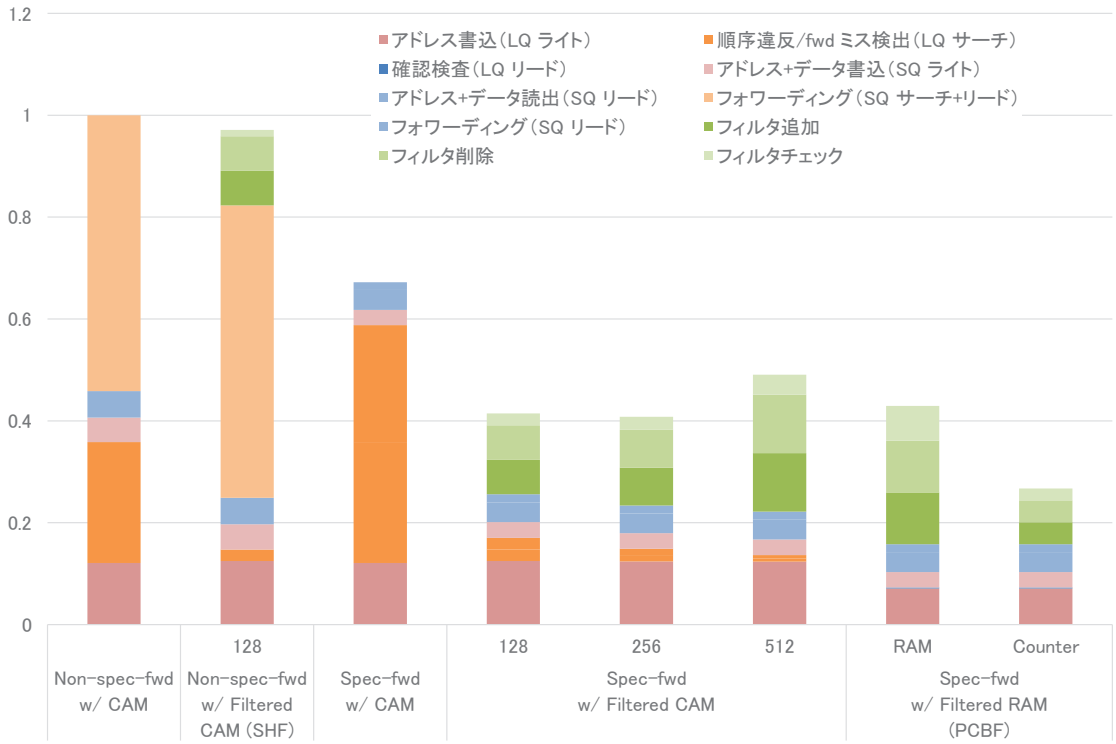

図 6.11: 表 6.4の各モデルの平均相対消費エネルギー

しかし，順序違反検出に加えて，フォワーディング・ミス検出のために LQ をサーチする必要があるため，それらの消費エネルギーが大きくなっている．

Spec-fwd w/ Filtered CAM では，その順序違反/フォワーディング・ミス検出をフィルタリングすることで消費エネルギーを削減することに成功している．また，フィルタのエントリ数を増加させると偽陽性率が減少するため，LQ のサーチのエネルギーがさらに削減されるが，フィルタ自体のエネルギーが大きくなるため，このモデルでは 256 エントリ程度のフィルタで消費エネルギーが最小となる．

Spec-fwd w/ Filtered RAM では，確認検査のための消費エネルギーがごくわずかであるが，これは Spec-fwd w/ Filtered CAM のモデルで順序違反/フォワーディング・ミス検出をフィルタリングした場合と大きく変わるものではない．LQ を CAM から RAM に変更したことの利点は，むしろ LQ に対するリード/ライトのエネルギーが減少している点にある．ただし，PCBF を RAM で構成すると，フィルタの消費エネルギーが大きいため，全体としては Spec-fwd w/ Filtered CAM と同程度の消費エネルギーとなる．この結果から，Spec-fwd w/ Filtered RAM のフィルタを SHF とすればよいと思われるかもしれないが，そのモデルは 6.2 節における

$k = 1$ の場合と同様で偽陽性率が大きいため，IPC が低下してしまう．

この問題は，提案カウンタを用いることで解決されている．フィルタに多ポートの RAM ではなくカウンタを用いたことでその消費エネルギーが削減され，最終的には Non-spec fwd w/ CAM と比較して 24.4%と大幅な消費エネルギーの削減を達成している．

表 6.5: ロード/ストアの動作と LSQ，フィルタに対する操作の関係

L/S	操 作	stage	CAM		RAM		Filter	
			LQ	SQ	LQ	SQ	RAM	counter
load	アドレス書込	ex	write		write			
	フォワーディング	ex		search read		read		
	フィルタ追加	ex					read-modify-write	inc
	フィルタ削除	cmt					read-modify-write	dec
store	アドレス＋データ書込	ex		write		write		
	アドレス＋データ読出 (L1D 更新のため)	cmt		read		read		
	順序違反 fwd ミス検出	ex (cmt)	search		read (確認検査)		read	check

第7章

関連研究

本章では，本論文の提案に関連する研究を紹介する．以下まず 7.1 節で，LSQ 自体の構成やアクセスを工夫し，LSQ を縮小する手法を紹介する．次に 7.2 節，本論文の基礎となる投機的発行と投機的にフォワーディングに関する各手法を示す．7.3 節では，LSQ を用いない順序違反/フォワーディング・ミス検出手法であるロード再実行と，それに連なる提案について説明する．最後に，7.4 節で説明した以外のフィルタを用いた順序違反/フォワーディング・ミス検出手法を紹介する．

7.1 LSQ を縮小する手法

LSQ 自体の構成やアクセス方法を工夫し，LSQ を縮小する手法を紹介する．

LSQ の非対称分割

[27, 28, 29] では，LSQ を，フル・アソシアティブである CAM で構成された部分と，セット・アソシアティブや FIFO の RAM で構成された部分とに分割する手法を提案している．前者は高速だが面積大・消費エネルギー大であり，後者は低速だが面積小・消費エネルギー小であるので，組み合わせることで面積効率・エネルギー効率の向上を図るものである．

これらの手法は，CAM をある程度削減できはするものの，後述する他の手法のように 0 にできる訳ではない．

ポート数・ビット数・エントリ数の削減

[30] では，LSQ を複数のセグメントに（対称的に）分け，各セグメントのポート数を削減する手法が提案されている．メモリを複数のセグメントに分けてポートを削減する手法自体は，レジスタ・ファイルなどではごく一般的に用いられている手法である．

[31] では，アドレスの上位ビットを圧縮することで LSQ の面積と消費エネルギーを削減する手法が提案されている．この手法ではハッシュを用いるのではなく，CAM で構成された変換テーブルによりアドレスの圧縮/展開を行っている．

また，[32] では，フロントエンドではなくロード/ストア命令の発行時に割り当てることで LSQ のエントリ数の削減を可能にする手法を提案している．割り当てを遅延させることでエントリ数を削減する手法自体は，物理レジスタ・ファイルなどにおいては既に提案されている [33].

これらの手法は，（必要であれば）提案手法と同時に採用することができ，提案手法と競合するものではない．

7.2　メモリ依存予測と投機的フォワーディング

ロード/ストア命令の投機的発行とそれに伴うメモリ・アクセス順序違反検出，そして，投機的フォワーディングとそれに伴うミス検出は，本論文の基礎となる技術である．

Store Set

[15] は，投機発行的発行にメモリ依存予測を用いることを提案した論文である．この論文は，すべてのロード/ストア命令は依存するかもしれないと（予測）して悲観的に発行する方法，すべてのロード/ストア命令は依存しないものと（予測）して楽観的 (optimistic) に発行する方法，予測器を用いて予測した結果に基づいて発行する方法を比較し，予測器を用いる方法が最も効率がよいことを示した．

この論文ではまた，予測器として **Store Set** を提案した．Store Set については，3.3 節で詳しく説明した．Store Set は，メモリ依存予測器として最初のものであるにもかかわらず，十分に高い予測精度を持っているため，これ以降メモリ依存予測器の研究はほとんどない．6 章の評価でも，Store Set を用いている．

SQIP, Fire-and-Forget, NoSQ

Store Queue Index Prediction (**SQIP**) [17] は，投機的フォワーディングを提案した手法である．投機的フォワーディングでは，Store Set などによる予測の結果に基づいて，ロード/ストア命令間で直接フォワーディングを行うため，SQ の CAM を RAM に置き換えることができる．なおこの手法では，投機的フォワーディング・ミスの検出には，後述するロード再実行を用いている．

7.3 ロード再実行

LSQ を用いない順序違反/フォワーディング・ミス検出手法として，Cain らによって提案された**ロード再実行** (load re-execution) がある [34].

ロード再実行では，ロード命令は，実行時に out-of-order にキャッシュからデータを読み出し，もう一度，コミット時に in-order にキャッシュからデータを読み出す．コミット・ステージではキャッシュは in-order に更新されるため，後者は正しいことが保証されている．したがって，前者と一致しているかどうかによって，順序違反検出を行うことができる．

この手法では，ストア命令が LQ にアクセスすることがないので LQ の CAM は不要となるが，各ロード命令がキャッシュに 2 回ずつアクセスするため，キャッシュの読み出しポートが圧迫されるという課題が残る．

SQIP と Fire-and-Forget, NoSQ

SQIP (Store Queue Index Prediction) [17] は，先ほどは投機的フォワーディングを提案した手法と紹介したが，研究の系譜上はロード再実行の系統に属する．前述のとおり，投機的フォワーディング・ミスの検出にロード再実行を用いている．

Fire-and-Forget [35] と **NoSQ** （No Store Queue） [18] は，SQIP をさらに推し進め，SQ の RAM までも省略する手法である．ただしこれらの手法では，コミット時にアドレス計算のためにレジスタ・ファイルを読み出す必要がある．そのため，通常の命令の実行のためのレジスタ・ファイル読み出しとの間でポートの競合の問題が生じる．

この問題を解決するためには，レジスタ・ファイルにロード再実行専用の読み出しポートを追加するか，あるいは，この競合までを考慮して命令スケジューリン

グを行うことなどが考えられる．しかし，いずれも極めて高コストであり，SQ の RAM を省略することには見合わないと考えられる．

SVW

4 章で詳しく説明した SVW (Store Vulnerability Window) [6] は，本来は，このロード再実行をフィルタすることによってこのポート数の問題を緩和するために提案された手法である．

4 章では，SVW を，本来のロード再実行のフィルタとして用いる手法ではなく，順序違反/フォワーディング・ミス検出のフィルタとして用いる手法を仮定して説明した．

SMDE

Slackened Memory Dependence Enforcement (**SMDE**) [36] は，ロード再実行に**Level-0 キャッシュ**を追加した手法である．後述する Store Forwarding Cache (SFC) と Memory Disambiguation Table (MDT) を用いた手法 [37] と，構成的によく似ている．ロード/ストア命令は Level-0 キャッシュには投機的にアクセスするため，順序違反/フォワーディング・ミス検出が必要となる．順序違反/フォワーディング・ミス検出は，SFC/MDT を用いる手法では MDT というフィルタによって行われるが，この手法はロード再実行によって行われる．

SMDE では，ロード命令の実行時の読み出しは Level-0 キャッシュに，再実行時の読み出しは Level-1 キャッシュへと分散されるため，各キャッシュの読み出しポートの圧迫を緩和することができる．

しかし SMDE では，CAM を用いた LSQ と同等の性能を得るために，大容量の Level-0 キャッシュを必要とする．評価では，64/48 エントリの LQ/SQ を用いた構成で，16KB もの Level-0 キャッシュを用いている．

7.4　その他のフィルタ手法

フィルタを用いて順序違反/フォワーディング・ミス検出を行う手法には，4 章で詳しく説明したもの以外にも，様々な亜種が存在する．

SFC/MDT を用いる手法

[37] では，Store Forwarding Cache (**SFC**) と Memory Disambiguation Table (**MDT**) というフィルタを用いる手法が提案されている．前節で述べたように，この手法は SMDE (Slackened Memory Dependence Enforcement) [36] とよく似ている．SMDE における Level-0 キャッシュは，この手法では Store Forwarding Cache (**SFC**) と呼ばれており，投機的フォワーディングの機能に主眼が置かれている．順序違反/フォワーディング・ミス検出は，SMDE ではロード再実行によって，この手法ではフィルタ MDT を用いて行われる．

MDT は，DMDC の前半部で利用する $\mathrm{LSN}_{\mathrm{exec}}$ と似ている．つまり，ロード命令が実行時にシーケンス・ナンバを MDT にライトし，ストア命令が実行時にリードすることで順序違反検出を行う．これは，4.1 節の分類で言えば a. シーケンス・ナンバ と ii. ld-exec → st-exec の組み合わせに相当する．

DMDC の前半部と異なるのは，この手法が MDT のみで順序違反/フォワーディング・ミス検出を行うため，偽陽性を削減するために MDT を大容量にしなければならない点にある．評価では，48/32 エントリの LQ/SQ の構成に対し，CAM を用いた LSQ と同等の性能を得るために，2KB の SFC と 16KB 程度の MDT を必要とする．

SMDE の Level-0 キャッシュと異なり SFC の容量が小さいのは，ロード再実行を用いた手法に必要な，Level-1 キャッシュの読み出しポートの圧迫を緩和するという役割を持たないためである．SMDE ではロードの実行時に Level-0 キャッシュのみにアクセスするため大容量である必要がある一方，この手法では SFC には in-flight なストア命令のデータのみを格納し，ロード命令は実行時に SFC と Level-1 キャッシュを同時にアクセスすればよい．

OFS

[38] では，4 章で詳しく説明した DMDC の前半部分を一部用いる手法を提案している．DMDC の前半と同様の方法によって LQ に対する CAM アクセスを削減するとともに，相似形の方法によって SQ に対する CAM アクセスも削減する．そのため，4.1 節の分類では a. シーケンス・ナンバ と ii. ld-exec → st-exec の組み合わせに相当する．

この手法では，DMDC と同様に，Youngest issued Load Age (**YLA**) を用いる．YLA は，発行された中で最も後続のロード命令のシーケンス・ナンバで，YLA レ

ジスタに保存される．ストア命令は，発行時にこの YLA レジスタを確認し，自身のシーケンス・ナンバよりも小さければ，後続のロード命令がストア命令を追い越していないことがわかる．よって，ストア命令が LQ に対する CAM アクセスを行わない．

この手法では，YLA に加えて，Oldest in-Flight Store (**OFS**) というシーケンス・ナンバを用い，YLA のそれと相似形の方法で SQ に対する CAM アクセスをフィルタする．

DMDC と同様に，YLA や OFS の値を格納するレジスタは，それぞれアドレスをインデックスとしたテーブルに保存することで，FP 率が減少する．しかし，4 章で説明したように，シーケンス・ナンバを用いているため，複数のハッシュ関数を用いることができず，LSQ の CAM を排除できるほどの偽陽性率は達成できていない．

ALQ/BNLQ

[39, 5, 40] では，前述した分割手法と，4.2.4 節で説明した SHF を組み合わせたような複雑な LQ が提案されている．この手法では，LQ を Associative Load Queue (**ALQ**) と呼ぶ CAM を用いた LQ と，Banked Non-associative Load Queue (**BNLQ**) と呼ぶ FIFO で構成された LQ の 2 種類に分け，更に BNLQ における順序違反検出にフィルタを用いる．

この手法では，依存予測器とは別の予測器を用い，順序違反を起こす可能性が高いと予測されたロード命令を ALQ に，低いと予測されたロード命令を BNLQ に格納する．その結果，ほとんどのロード命令が BNLQ で扱われ，消費エネルギーが削減できるとしている．

BNLQ の順序違反検出は，4.2.4 節で説明した SHF [4] と同様と考えてよく，4.1 節の分類でも b. ビット と ii. ld-exec → st-exec の組み合わせである．BNLQ に格納されるロード命令は，Exclusive Bloom Filter と呼ぶフィルタに登録され，BNLQ に対する確認検査の頻度を減らす．この EBF も SHF と同様のハッシュ・テーブルであり，複数のハッシュ関数に関する言及はない．

またこの手法でも，SHF と同様，投機的フォワーディングについては考慮されていない．

第8章

結論

8.1 本論文のまとめ

ハイエンド out-of-order スーパスカラ・プロセッサの拡大が進み，CAM で構成された LSQ の回路面積や消費エネルギーが問題となってきている．そうした中，RAM によって構成されたフィルタを用いて順序違反/フォワーディング・ミス検出を行うことで，LSQ の CAM の電力を削減したり，CAM を省略する手法が提案されてきた．

本論文では，その順序違反/フォワーディング・ミス検出のためのフィルタとして，ブルーム・フィルタ (BF) を用いる手法を提案した．既存手法が高い偽陽性率に苦しむ中，BF の特長である複数のハッシュ関数を用いることで極めて低い偽陽性率を達成できる．

また，提案手法と既存手法を 1. フィルタの基本構成と，2. フィルタへの基本的なアクセス手順 の 2 点を基準として分類し，提案手法がより問題が少ないことを示した．

BF を用いるにあたっては様々な問題が発生するが，本論文では以下に示すような方法で解決した．

1. フィルタを更新したロード命令が予測ミスだった場合，コミットせずにフラッシュされるとフィルタに不整合が生じてしまう．そのため，予測ミスした命令をすぐにフラッシュせず遅延消去することによってフィルタの一貫性を保つ．また，この手法によって生じる性能低下を抑える方法も合わせて提案した．
2. BF 自体の回路面積と消費エネルギーを削減するため，パラレル・カウンティン

グ・ブルーム・フィルタ (PCBF) を採用した．また，同時実行可能なロード/ストア命令が多いハイエンドのプロセッサ・コアにおいて，この PCBF を RAM で構成するとポート数が非常に大きくなり，フィルタ自体の回路面積や消費エネルギーが問題となる．そこで，カウンタ機能付き機能メモリ (functional memory) を用いて PCBF を構成することで，回路面積や消費エネルギーを大幅に削減した．

3. PCBF を用いる場合，偽陽性が発生したとき以外にも，カウンタがフルになったり，ハッシュが衝突したときにペナルティが生じるため，それぞれのペナルティに削減する手法を提案した．さらに，フィルタを用いた手法には，ロード/ストア命令のアクセス・サイズに起因してペナルティが増大するという問題がある．本論文では，MCBF と SCBF という 2 種類の異なるアクセス・サイズを扱う PCBF を組み合わせることにより，このサイズに起因するペナルティを削減した．

これらの提案をシミュレータに実装し評価を行った結果，平均 98.6%の IPC を維持しながら，回路面積を 20.3%，消費エネルギーを 24.4%にまで削減できることが示された．

8.2　今後の課題

今後検討すべき課題としては，より大きなコアへの適用や，マルチスレッド・プロセッサへの適用が考えられる．

2013 年に発表された POWER8 は，各バッファのエントリ数等の詳細が公表されていないものの，ロード 2 命令とロード/ストア 2 命令が同時に実行可能であるとされる [3]．これは，評価で用いた同時実行可能数の倍であり，CAM を用いた順序違反/フォワーディング・ミス検出では LSQ の面積が単純には 4 倍となることが考えられる．一方で，本論文で提案したフィルタを用いれば，ポート数が同時実行可能な命令数と無関係なため，本論文での提案の効果がさらに高まることが期待される．

また，マルチスレッド・プロセッサに対する適用についても検討する余地がある．投機的マルチスレッディング [41, 42, 43, 44, 45, 46] や Switch-on-Future-Event マルチスレッディング [47, 48] では，ひとつのプログラムを複数のスレッドで実行す

るため，スレッド間のメモリ・アクセスの先行制約を満たさなければならない．本論文の成果は，こうしたマルチスレッド・プロセッサにおいて特に有効であると考えられ，今後はそれらに対して適用することを考えている．

謝辞

本研究をすすめるにあたり，指導教員である坂井修一教授には卒論，修士課程，博士課程と 6 年間の長きにわたり，御指導，御鞭撻を頂きました．ここに深く感謝の意を表します．

国立情報学研究所の五島正裕教授には研究の方針から細部に至るまで数多くの有益なご助言をいただき，また研究以外に関しても様々な機会に幅広くご相談させていただきました．本当にありがとうございました．

浅見徹教授，田浦健次朗准教授，豊田正史准教授には，審査において大変有益なご助言を頂きました．

名古屋大学の塩谷亮太助教には，研究に関する数多くのご協力，ご助言のほか，研究室での生活において多くのご指導をいただきました．

当時研究室の学生であった藤田晃史氏には，本研究の基礎となるアイディアに関する議論を通じて，多くのご協力をいただきました．

秘書の八木原晴水氏，長谷部環氏には，研究室での生活や事務手続きに関する数多くのサポートをしていただきました．

その他，研究室に在籍中多くの皆様に，研究生活を通じて様々なご協力，ご支援を頂きました．

ここに深甚なる謝意を表します．

本論文の研究は，一部文部科学省科学研究費補助金 No. 23300013，26280012 によります．

参考文献

[1] Sinharoy, B., Kalla, R., Starke, W. J., Le, H. Q., Cargnoni, R., Van Norstrand, J. a., Ronchetti, B. J., Stuecheli, J., Leenstra, J., Guthrie, G. L., Nguyen, D. Q., Blaner, B., Marino, C. F., Retter, E. and Williams, P.: IBM POWER7 multicore server processor, *IBM Journal of Research and Development*, Vol. 55, No. 3, pp. 1:1–1:29 (online), doi:10.1147/JRD.2011.2127330 (2011).

[2] Krewell, K.: INTEL'S HASWELL CUTS CORE POWER, Vol. 2 (2012).

[3] Stuecheli, J.: Next Generation POWER microprocessor, *Hot Chips 25*, (2013) (the photograph is an excerpt from: http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.20-Processors1-epub/HC25.26.210-POWER-Studecheli-IBM.pdf).

[4] Sethumadhavan, S., Desikan, R., Burger, D., Moore, C. R. and Keckler, S. W.: Scalable Hardware Memory Disambiguation for High ILP Processors, *36th International Symposium on Microarchitecture (MICRO'03)*, pp. 399–410 (2003).

[5] Castro, F., Chaver, D., Pinuel, L., Prieto, M., Tirado, F. and Huang, M.: Load-store queue management: an energy-efficient design based on a state-filtering mechanism, *International Conference on Computer Design 2005 (ICCD'05)*, pp. 617–624 (online), doi:10.1109/ICCD.2005.70 (2005).

[6] Roth, A.: Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 458–468 (online), doi:10.1109/ISCA.2005.48 (2005).

[7] Castro, F., Pinuel, L., Chaver, D., Prieto, M., Huang, M. and Tirado, F.: DMDC: Delayed Memory Dependence Checking through Age-Based Filtering, *39th International Symposium on Microarchitecture (MICRO'06)*, pp. 297–308 (online), doi:10.1109/MICRO.2006.21 (2006).

[8] 倉田成己，塩谷亮太，五島正裕，坂井修一:ブルーム・フィルタを用いたメモリ・アクセス順序違反検出機構，情報処理学会研究報告 2014-ARC-212，No. 17, pp. 1–15 (2014).

[9] 倉田成己，塩谷亮太，五島正裕，坂井修一:ブルーム・フィルタを用いたメモリ・アクセス順序違反検出機構の評価，情報処理学会研究報告 2014-ARC-213，No. 11, pp. 1–10 (2014).

[10] Bloom, B. H.: Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM*, Vol. 13, No. 7, pp. 422–426 (online), doi:10.1145/362686.362692 (1970).

[11] Huang, K., Zhang, J., Zhang, D., Xie, G., Salamatian, K., Liu, A. and Li, W.: A Multi-partitioning Approach to Building Fast and Accurate Counting Bloom Filters, *2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS'13)*, pp. 1159–1170 (online), doi:10.1109/IPDPS.2013.51 (2013).

[12] Sanchez, D., Yen, L., Hill, M. D. and Sankaralingam, K.: Implementing Signatures for Transactional Memory, *40th International Symposium on Microarchitecture (MICRO'07)*, pp. 123–133 (online), doi:10.1109/MICRO.2007.24 (2007).

[13] Almeida, J. and a.Z. Broder: Summary cache: a scalable wide-area Web cache sharing protocol, *IEEE/ACM Transactions on Networking*, Vol. 8, No. 3, pp. 281–293 (online), doi:10.1109/90.851975 (2000).

[14] 五島正裕，倉田成己，塩谷亮太，坂井修一:タイミング・フォールト耐性を持つ Out-of-Order プロセッサ，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, pp. 17–30 (2013).

[15] Chrysos, G. Z. and Emer, J. S.: Memory Dependence Prediction Using Store Sets, *25th International Symposium on Computer Architecture (ISCA'98)*, pp. 142–153 (online), doi:10.1145/279358.279378 (1998).

[16] Roesner, F., Burger, D. and Keckler, S. W.: Counting Dependence Predictors, *35th International Symposium on Computer Architecture (ISCA'08)*, pp. 215–226 (online), doi:10.1109/ISCA.2008.6 (2008).

[17] Martin, M. and Roth, A.: Scalable Store-Load Forwarding via Store Queue Index Prediction, *38th International Symposium on Microarchitecture (MICRO'05)*, pp. 159–170 (online), doi:10.1109/MICRO.2005.29 (2005).

[18] Sha, T., Martin, M. M. K. and Roth, A.: NoSQ: Store-Load Communication Without a Store Queue, *39th International Symposium on Microarchitecture (MICRO'06)*, pp. 285–296 (online), doi:10.1109/MICRO.2006.39 (2006).

[19] Moshovos, A. and Sohi, G. S.: Speculative Memory Cloaking and Bypassing, *International Journal of Parallel Programming*, Vol. 27, No. 6, pp. 427–456 (online), doi:10.1023/A:1018776132598 (1999).

[20] 五島正裕:ディジタル回路，数理工学社 (2007).

[21] 五島正裕:Out-of-order ILP プロセッサにおける命令スケジューリングの高速化の研究，博士論文 (2004).

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

[23] 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 SACSIS2009，pp. 120–121 (2009).

[24] Sakai Lab: *Processor Simulator Onikiri 2* *http://www.mtl.t.u-tokyo.ac.jp/~onikiri2/*.

[25] Thoziyoor, S., Muralimanohar, N., Ahn, J. and Jouppi, N.: CACTI 5.1., Technical report, HP Laboratories (2008).

[26] North Carolina State University: NCSU EDA Wiki. http://www.eda.ncsu.edu/wiki/NCSU_EDA_Wiki.

[27] Gandhi, A., Akkary, H., Rajwar, R., Srinivasan, S. T. and Lai, K.: Scalable Load and Store Processing in Latency Tolerant Processors, *32nd International Symposium on Computer Architecture (ISCA'05)*, ISCA '05, IEEE Computer Society, pp. 446–457 (online), doi:10.1109/ISCA.2005.46 (2005).

[28] Baugh, L. and Zilles, C.: Decomposing the Load-store Queue by Function for Power Reduction and Scalability, *IBM Journal of Research and Development*, Vol. 50, No. 2/3, pp. 287–297 (online), doi:10.1147/rd.502.0287 (2006).

[29] Pericàs, M., Cristal, A., Cazorla, F. J., González, R., Veidenbaum, A., Jiménez, D. A. and Valero, M.: A Two-Level Load/Store Queue Based on Execution Locality, *35th International Symposium on Computer Architecture (ISCA'08)*, ISCA '08, IEEE Computer Society, pp. 25–36 (online), doi:10.1109/ISCA.2008.10 (2008).

[30] Park, I., Ooi, C. L. and Vijaykumar, T. N.: Reducing Design Complexity of the Load/Store Queue, *36th International Symposium on Microarchitecture (MICRO'03)*, MICRO 36, IEEE Computer Society, pp. 411– (online), http://dl.acm.org/citation.cfm?id=956417.956555 (2003).

[31] Tsai, Y.-Y., Hsu, C.-J. and Chen, C.-H.: Power-efficient and Scalable Load-/Store Queue Design via Address Compression, *The 2008 ACM Symposium on Applied Computing (SAC'08)*, SAC '08, ACM, pp. 1523–1527 (online), doi:10.1145/1363686.1364042 (2008).

[32] Sethumadhavan, S., Roesner, F., Emer, J. S., Burger, D. and Keckler, S. W.: Late-binding: Enabling Unordered Load-store Queues, *34th International Symposium on Computer Architecture (ISCA'07)*, ISCA '07, ACM, pp. 347–357 (online), doi:10.1145/1250662.1250705 (2007).

[33] TANAKA, Y. and ANDO, H.: Register File Size Reduction through Instruction Pre-Execution Incorporating Value Prediction, *IEICE transac-*

tions on information and systems, Vol. 93, No. 12, pp. 3294–3305 (online), doi:10.1587/transinf.E93.D.3294 (2010).

[34] Cain, H. W. and Lipasti, M. H.: Memory ordering: A value-based approach, *31st International Symposium on Computer Architecture (ISCA'04)*, Vol. 24, pp. 90–101 (online), doi:10.1109/ISCA.2004.1310766 (2004).

[35] Subramaniam, S. and Loh, G. H.: Fire-and-Forget: Load/Store Scheduling with No Store Queue at All, *39th International Symposium on Microarchitecture (MICRO'06)*, MICRO 39, pp. 273–284 (online), doi:10.1109/MICRO.2006.26 (2006).

[36] Garg, A., Rashid, M. W. and Huang, M.: Slackened Memory Dependence Enforcement: Combining Opportunistic Forwarding with Decoupled Verification, *33rd International Symposium on Computer Architecture (ISCA'06)*, ISCA '06, IEEE Computer Society, pp. 142–154 (online), doi:10.1109/ISCA.2006.36 (2006).

[37] Stone, S. S., Woley, K. M. and Frank, M. I.: Address-Indexed Memory Disambiguation and Store-to-Load Forwarding, *38th International Symposium on Microarchitecture (MICRO'05)*, MICRO 38, pp. 171–182 (online), doi:10.1109/MICRO.2005.10 (2005).

[38] Castro, F., Chaver, D., Pinuel, L., Prieto, M. and Tirado, F.: Using Age Registers for a Simple Load-store Queue Filtering, *Journal of Systems Architecture: the EUROMICRO Journal*, Vol. 55, No. 2, pp. 79–89 (online), doi:10.1016/j.sysarc.2008.09.005 (2009).

[39] Castro, F., Chaver, D., Pinuel, L., Prieto, M., Huang, M. C. and Tirado, F.: A Power-efficient and Scalable Load-store Queue Design, *Proceedings of the 15th International Conference on Integrated Circuit and System Design: Power and Timing Modeling, Optimization and Simulation*, PATMOS'05, Berlin, Heidelberg, Springer-Verlag, pp. 1–9 (online), doi:10.1007/11556930_1 (2005).

[40] Castro, F., Chaver, D., Pinuel, L., Prieto, M., Huang, M. and Tirado, F.: LSQ: a power efficient and scalable implementation, *IEE Proceedings - Computers and Digital Techniques*, Vol. 153, No. 6, pp. 389–398 (2006).

[41] Sohi, G., Breach, S. and Vijaykumar, T.: Multiscalar processors, *22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pp. 414–425 (1995).

[42] 小林良太郎，小川行宏，岩田充晃，安藤秀樹，島田俊夫:非数値計算応用向けスレッド・レベル並列処理マルチプロセッサ・アーキテクチャSKY，情報処理学会論文誌， Vol. 42, No. 2, pp. 349–366 (2001).

[43] Park, I., Falsafi, B. and Vijaykumar, T. N.: Implicitly-multithreaded processors, *30th International Symposium on Computer Architecture (ISCA'03)*, ACM, pp. 39–51 (online), doi:10.1145/859618.859624 (2003).

[44] Akkary, H. and Driscoll, M. A.: A dynamic multithreading processor, *31st International Symposium on Microarchitecture (MICRO'98)*, pp. 226–236 (1998).

[45] Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M. and Olukotun, K.: The Stanford Hydra CMP, *IEEE Micro*, Vol. 20, No. 2, pp. 71–84 (online), doi:10.1109/40.848474 (2000).

[46] Ohsawa, T., Takagi, M., Kawahara, S. and Matsushita, S.: Pinot: Speculative Multi-threading Processor Architecture Exploiting Parallelism over a Wide Range of Granularities, *38th International Symposium on Microarchitecture (MICRO'05)*, pp. 81–92 (online), doi:10.1109/MICRO.2005.26 (2005).

[47] 倉田成己，塩谷亮太，五島正裕，坂井修一:Switch-On-Future-Event マルチスレッディングの改良と評価，先進的計算基盤システムシンポジウム SACSIS 2011，pp. 82–91 (2011).

[48] 塩谷亮太，倉田成己，中島　潤，五島正裕，坂井修一:Switch-On-Future-Event マルチスレッディング，先進的計算基盤システムシンポジウム SACSIS 2010，pp. 157–165 (2010).

著者発表論文

雑誌論文

[1] Kurata, N., Shioya, R., Goshima, M. and Sakai, S.: Address Order Violation Detection with Parallel Counting Bloom Filters（条件付採録）, *IEICE Transactions on Electronics, Special Section on Low-Power and High-Speed Chips* (2015).

[2] 吉田宗史, 広畑壮一郎, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 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, 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, pp. 1–16 (2013).

[3] 五島正裕, 倉田成己, 塩谷亮太, 坂井修一: タイミング・フォールト耐性を持つ Out-of-Order プロセッサ, 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, p. 17–30 (2013).

[4] Shioya, R., Kurata, N., Toyoshima, T., Goshima, M. and Sakai, S.: Register Indirect Jump Target Forwarding, *IEICE Transactions on Information and Systems*, Vol. E96-D, No. 2, pp. 278–288 (2013).

国際会議

[5] Yamada, T., Kurata, N., Yamaguchi, R. S., Goshima, M. and Sakai, S.: Minimal Additional Function to Secure Processor for Application Authentication, *Western European Workshop on Research in Cryptology (WEWoRC 2013)*, pp. 301–312 (2010).

口頭発表

[6] 吉田宗史, 広畑壮一郎, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: タイミング・フォールト耐性を持つ Out-of-Order プロセッサの検出/回復方式, 先進的計算基盤システムシンポジウム SACSIS 2013, pp. 10–19 (2013).

[7] 浅見公輔, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 命令グループごとのキャッシュ・パーティショニング, 先進的計算基盤システムシンポジウム SACSIS 2013, pp. 65–69 (2013).

[8] 有馬慧, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: タイミング・フォールト耐性を持つ Out-of-Order プロセッサ, 先進的計算基盤システムシンポジウム SACSIS 2012, pp. 270–279 (2012).

[9] 吉田宗史, 広畑壮一郎, 倉田成己, 五島正裕, 坂井修一: 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, 先進的計算基盤システムシンポジウム SACSIS 2012, pp. 382–389 (2012).

[10] 伊達三雄, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: レジスタ・リネーミングとディスパッチ・ネットワークを最小化するプロセッサ・アーキテクチャ, 先進的計算基盤システムシンポジウム SACSIS 2012, pp. 280–288 (2012).

[11] 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: Switch-On-Future-Event マルチスレッディングの改良と評価, 先進的計算基盤システムシンポジウム SACSIS 2011, pp. 82–91 (2010).

[12] 塩谷亮太, 倉田成己, 中島潤, 五島正裕, 坂井修一: Switch-On-Future-Event マルチスレッディング, 先進的計算基盤システムシンポジウム SACSIS 2010, pp. 157–165 (2010).

[13] 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: ブルーム・フィルタを用いたメモリ・アクセス順序違反検出機構の評価, 情報処理学会研究報告 2014-ARC-213, No. 11, pp. 1–10 (2014).

[14] 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: ブルーム・フィルタを用いたメモリ・アクセス順序違反検出機構, 情報処理学会研究報告 2014-ARC-212, No. 17, pp. 1–15 (2014).

[15] 山田淳二, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: レジスタ・キャッシュ・システムにおけるレジスタ・ファイルのマルチ・バンク化, 情報処理学会研究報告 2014-ARC-212, No. 18, pp. 1–14 (2014).

[16] 福田隆, 倉田成己, 五島正裕, 坂井修一: 帰納的なシミュレーション・ポイント選出手法の改良, 情報処理学会研究報告 2014-ARC-212, No. 20, pp. 1–8 (2014).

[17] 福田隆, 倉田成己, 五島正裕, 坂井修一: 既存アーキテクチャのシミュレーション結果を用いる汎用シミュレーションポイント検出手法, 情報処理学会研究報告 2014-ARC-211, No. 12, pp. 1–7 (2013).

[18] 藤田晃史, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 面積効率を指向するプロセッサ「雷上動」の設計と実装, 電子情報通信学会研究報告 CPSY 2013, pp. 229–234 (2014).

[19] 山田淳二, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: レジスタ・キャッシュ・システムにおけるレジスタ・ファイルへの書き込みの削減手法, 情報処理学会研究報告 2014-ARC-208, No. 11, pp. 1–12 (2014).

[20] 早川薫, 倉田成己, 五島正裕, 坂井修一: 可変長セグメントを用いたフェーズ検出手法の改良, 情報処理学会研究報告 2014-ARC-206, No. 26, pp. 1–9 (2013).

[21] 山田淳二, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: レジスタ・キャッシュ・システムの省電力化手法, 情報処理学会研究報告 2014-ARC-206, No. 3, pp. 1–9 (2013).

[22] 西川卓, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: マルチスレッド・プロセッサにおけるレジスタ・キャッシュ・システムの評価, 情報処理学会研究報告 2014-ARC-206, No. 4, pp. 1–7 (2013).

[23] 吉田宗史, 広畑壮一郎, 倉田成己, 五島正裕, 坂井修一: 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適用手法の評価, 情報処理学会研究報告 2014-ARC-206, No. 6, pp. 1–13 (2013).

[24] 広畑壮一郎, 吉田宗史, 倉田成己, 五島正裕, 坂井修一: 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適用手法の実装, 情報処理学会研究報告 2014-ARC-204, No. 11, pp. 1–9 (2013).

[25] 西川卓, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: レジスタ・キャッシュのマルチスレッド・プロセッサへの適用, 情報処理学会第 75 回全国大会, pp. 159–160 (2013).

[26] 宮永瑞紀, 伊達三雄, 塩谷亮太, 五島正裕, 坂井修一: フロントエンド・パイプラインを最小化する命令キャッシュ・アーキテクチャの置換アルゴリズムの改良, 情報処理学会第 75 回全国大会, pp. 161–162 (2013).

[27] 阿部高大, 倉田成己, 五島正裕, 坂井修一: 過去の競合命令にチェックポイントを設定するトランザクショナル・メモリ, 情報処理学会研究報告 2014-ARC-201, No. 2, pp. 1–8 (2012).

[28] 堀口達也, 吉田宗史, 倉田成己, 五島正裕, 坂井修一: 耐故障 FPGA アーキテクチャ, 情報処理学会研究報告 2014-ARC-201, No. 5, pp. 1–7 (2012).

[29] 浅見公輔, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 命令グループごとのキャッシュ・パーティショニングの予備評価, 情報処理学会研究報告 2014-ARC-201, No. 14, pp. 1–11 (2012).

[30] 広畑壮一郎, 吉田宗史, 倉田成己, 五島正裕, 坂井修一: 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適用手法, 情報処理学会研究報告 2014-ARC-201, No. 20, pp. 1–8 (2012).

[31] 浅見公輔, 倉田成己, 塩谷亮太, 三輪忍, 五島正裕, 坂井修一: 命令グループのワーキング・セットに着目したキャッシュ・マネジメント, 情報処理学会研究報告 2014-ARC-200, No. 14, pp. 1–7 (2012).

[32] 浅見公輔, 倉田成己, 塩谷亮太, 三輪忍, 五島正裕, 坂井修一: キャッシュの利用効率の向上に関する研究, 情報処理学会第 74 回全国大会, pp. 61–62 (2012).

[33] 吉田宗史, 有馬慧, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 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, 電子情報通信学会研究報告 CPSY 2011, No. 7, pp. 13–18 (2011).

[34] 伊達三雄, 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: レジスタ・リネーミングとディスパッチ・ネットワークを不要とするトレース・キャッシュ・アーキテクチャ, 情報処理学会研究報告 2014-ARC-196, No. 25, pp. 1–10 (2011).

[35] 伊達三雄, 倉田成己, 伊藤悠二, 塩谷亮太, 五島正裕, 坂井修一: ディスパッチト・イメージ・キャッシュ, 情報処理学会第 73 回全国大会, pp. 1–67–1–68 (2011).

[36] Kurata, N., Shioya, R., Nakashima, J., Goshima, M. and Sakai, S.: An Improvement of Switch-on-Future-Event Multithreading, 情報処理学会研究報告 2010-ARC-190, No. 27, pp. 1–9 (2010).

[37] 倉田成己, 塩谷亮太, 五島正裕, 坂井修一: 繰り返し構造に着目した分岐プレディシジョンの改良, 情報処理学会第 72 回全国大会, pp. 1–213–1–214 (2010).

[38] 広畑壮一郎, 神原太郎, 吉田宗史, 倉田成己, 五島正裕, 坂井修一: 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適用手法の予備評価, 先進的計算基盤システムシンポジウム SACSIS 2013, pp. 143–144 (2013 (poster)).

[39] 広畑壮一郎, 吉田宗史, 倉田成己, 五島正裕, 坂井修一: 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適用, 先進的計算基盤システムシンポジウム SACSIS 2012, pp. 12–13 (2012 (poster)).

[40] 倉田成己, 樋口和英: CUDA による O(ND) アルゴリズムの並列化, 先進的計算基盤システムシンポジウム SACSIS 2009, pp. 112–113 (2009 (poster)).

受賞

[41] 倉田成己: 第 204 回計算機アーキテクチャ研究会 若手奨励賞，2014 年

[42] 倉田成己: 情報処理学会創立 50 周年記念（第 72 回）全国大会学会推奨卒業論文認定，2010 年

[43] 倉田成己，樋口和英: GPU Challenge 2009 規定課題部門第 3 位入賞，2009 年 5 月