修士論文

Bloom-like SVW の提案

Proposal of Bloom-like SVW

平成28年02月04日提出

指導教員

坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-136427　西川 卓

概要

実行順序違反検出やフォワーディングを行うための，LSQ の CAM は，その構造上，面積・消費電力が大きい．そこで，LSQ の CAM を，検出に僅かな偽陽性を許すフィルタに置き換えることで面積削減を図る手法がいくつかある．

なかでも，フィルタを用いた実行順序違反検出手法のひとつである Store Vulnerability Window(SVW) においては，偽陽性率が高く，IPC 低下に見合うだけの面積削減が見込めないという問題があった．本論文では，偽陽性の小さいフィルタであるブルーム・フィルタを一般化し，それを SVW に応用した偽陽性の低いフィルタ Bloom-like SVW を提案する．

本論文では，Bloom-like SVW のフィルタとしての性能の評価と，投機フォワーディングを適用した時の評価を行った．評価では，提案手法は，5,000 ビット程度のフィルタで，1% 未満の IPC 低下におさまる性能を示した．また，提案手法に投機フォワーディングを適用した結果，その IPC 低下は 2% 以内におさまった．

目次

第1章　はじめに **7**

第2章　ロード/ストア命令の投機実行 **10**

2.1 ロード/ストア命令の実行とコミット 10
2.2 ロード/ストア命令の投機実行を支える技術 11
2.2.1 本論文で用いるパイプライン・チャート 12
2.2.2 メモリ・アクセス順序違反 12
2.2.3 メモリ依存予測 14
2.2.4 投機フォワーディング 15
2.3 順序違反検出/フォワーディングと LSQ の CAM 17

第3章　フィルタを用いたメモリ・アクセス順序違反手法 **20**

3.1 Store Vulnerability Window 21
3.1.1 SVW の順序違反検出 21
3.1.2 SVW による フォワーディングの処理 23
3.1.3 SVW の問題点 25
3.2 パラレル・カウンティング・ブルーム・フィルタを用いた手法 25
3.2.1 PCBF の順序違反検出 26
3.2.2 フォワーディングの処理 28
3.3 Delayed Memory Dependence Checking 29
3.3.1 DMDC の順序違反検出 31
3.3.2 DMDC の問題 32

第4章　ブルーム・フィルタ **34**

4.1 ハッシュ・フィルタの動作例 34
4.2 ブルーム・フィルタの動作例 36
4.3 ブルーム・フィルタの解析解 37
4.4 BF の問題とその解決策 38

第 5 章　Bloom-like SVW **40**
5.1　ブルーム・フィルタの原理とその一般化 40
5.2　Bloom-like SVW 41
5.2.1　Bloom-like SVW の順序違反検出 42
5.2.2　Bloom-like SVW におけるフォワーディングの処理 43

第 6 章　評価 **44**
6.1　評価環境 44
6.2　フィルタの容量に対する性能評価 44
6.3　ベンチマーク毎の偽陽性率と相対 IPC 45
6.4　提案手法への投機フォワーディング適用時の評価 47
6.4.1　評価モデル 47
6.4.2　評価 49
6.5　考察 54

第 7 章　おわりに **57**
7.1　本論文のまとめ 57
7.2　今後の課題 57

参考文献 **59**

表目次

6.1 プロセッサの構成 45
6.2 依存予測器 の構成 49

図 目 次

1.1 POWER8 のチップ写真 [1] 7
1.2 L1D と LSQ の CAM の面積 8

2.1 一般的なパイプライン・チャート (上) と本稿で用いるパイプライン・チャート (下) 12
2.2 メモリ・アクセス順序違反の例 13
2.3 依存距離の測り方 15
2.4 フォワーディングおよびフォワーディング・ミス 17

3.1 SVW の順序違反検出 22
3.2 SVW におけるフォワーディングの処理 : ミスがない場合 (上) とミスがある場合 (下) 24
3.3 PCBF における順序違反検出:順序違反がない場合（上）とある場合（下）[2] 27
3.4 PCBF におけるフォワーディングの処理 : ミスがない場合 (上) とミスがある場合 (下)[2] 28
3.5 DMDC における順序違反検出:順序違反がない場合（上）とある場合（下） 31

4.1 ハッシュ・フィルタ の例 34
4.2 ブルーム・フィルタの例 36
4.3 ブルーム・フィルタの解析解 37

5.1 Bloom-like SVW の順序違反検出:順序違反がない場合（上）とある場合（下） 42

6.1 フィルタの容量に対する FalsePositive 発生率 (上) と相対 IPC(下) . . 46
6.2 ベンチマーク毎の偽陽性発生率 (上) と相対 IPC(下) 47
6.3 ハッシュ関数の数 : k = 4 における 偽陽性率とフィルタの総容量 . . 48

6.4 ベースラインに対する相対 IPC 50
6.5 予測テーブルの容量と相対 IPC 50
6.6 依存予測の結果 (INT 系) 53
6.7 依存予測の結果 (FP 系) 53
6.8 実際に依存はないが依存があると予測されるロードの割合 53
6.9 LQ-CAM によるフォワーディング時の偽陽性 54
6.10 サイズの違いによる偽陽性 55
6.11 サイレント・ストアによる偽陽性 56

第1章　はじめに

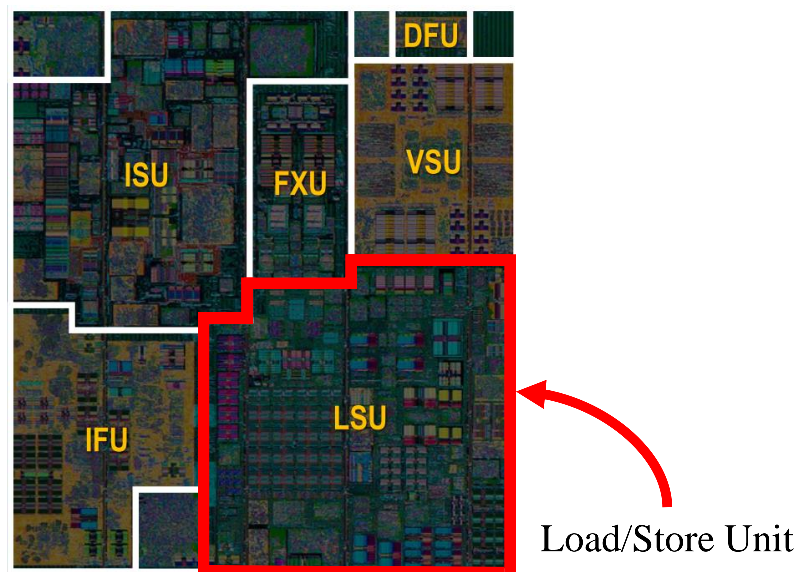

図 1.1: POWER8 のチップ写真 [1]

ロード/ストア・キュー (LSQ) は，out-of-order スーパスカラ・プロセッサの構成要素の中で最も高コストなものの 1 つとなっている．

LSQ と CAM　out-of-order スーパスカラ・プロセッサにおいて，LSQ は，ロード/ストア命令の依存による先行制約を守りつつ，out-of-order 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/ストア命令の依存関係は「曖昧」である，すなわち，先行制約を満たすためには，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のための，動的なターゲット・アドレスの比較が必須である．ターゲット・アドレスの比較は，従来，**CAM** を用いた 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，回路面積と消費電力が大きい．

LSQ 規模の増加　ハイエンドの out-of-order スーパスカラ・プロセッサの規模の拡大は，ゆっくりとだが確実に続いている [1, 3]．特に，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 のエントリ数は拡大の一途をたどっている．また，in-flight 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り，ポート数の二乗に比例して CAM の面積は大きくなる．

これら 2 つの理由により，最近のハイエンド・プロセッサでは，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** は，IBM POWER8 プロセッサのチップ写真であるが [1]，この図から見てわかるように LSU はコアの 1/4 程度の領域を占める大きなものとなっている．また**図 1.2** は Intell Haswell プロセッサ [3] における L1D の面積と LSQ の CAM の面積を CACTI [4] によって算出したものである．このグラフから見てわかるように，LSQ の CAM は L1D に匹敵するほどの大きな面積になっていることがわかる．

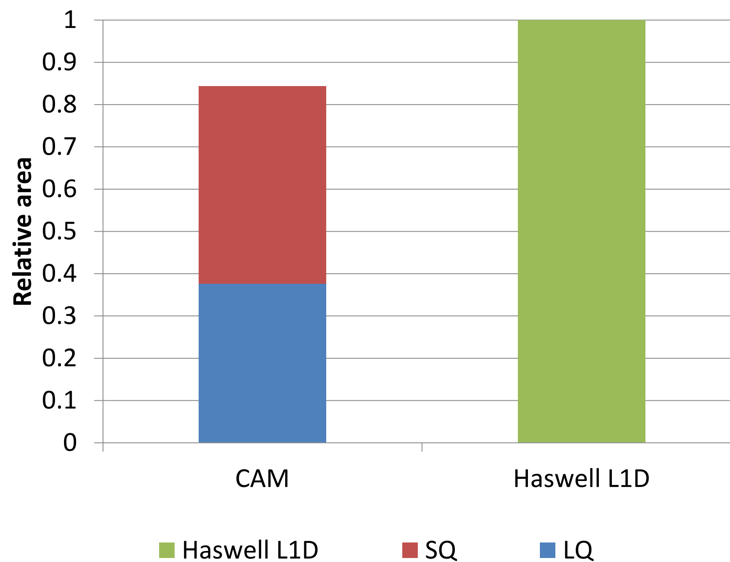

図 1.2: L1D と LSQ の CAM の面積

フィルタを用いた順序違反検出　LQ の CAM を排除する手法に RAM で構成されるフィルタを用いて，ロード/ストアの先行制約を守る手法がいくつか提案されている [5, 2, 6]．これらの手法は**フィルタに要素を登録/参照/消去**を行い，メモリ・アクセス順序違反を検出することに大きな特徴がある．検出に用いられるフィルタ

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．我々は，かつて偽陽性が低いブルーム・フィルタ [7] を応用した，**パラレル・カウンティング・ブルーム・フィルタ (PCBF)** を用いた手法 [2] を提案している．

ブルーム・フィルタの一般化　[2] では，複数のハッシュ関数を用いるフィルタ:ブルーム・フィルタを使うことでメモリ・アクセス順序違反検出に伴う偽陽性率を削減できることが示されてる．だが，[2] では，SVW のようなシーケンス・ナンバを読み書きするようなフィルタには複数のハッシュ関数を適用する方法は知られておらず，偽陽性率が大きくなると示されていた．そこで，本稿ではブルーム・フィルタの考え方をより一般的なフィルタへと拡張し，SVW のようなシーケンスナンバを読み書きするフィルタでも，複数のハッシュ関数を適用可能であることを示す．そして，SVW に複数のハッシュ関数を適用した Bloom-like SVW を提案する．

本論文の構成　本論文では，まず 2 章にて，ロード/ストア命令の out-of-order 実行が投機的な実行になることを示してから，メモリ・アクセス順序違反を含めたロード/ストア命令の投機実行を支える技術について述べる．そして，フィルタによってメモリ・アクセス順序違反を検出する手法について 3 章で述べた後に，4 章で本手法の要となるブルーム・フィルタについて述べる．そして 5 章では，ブルーム・フィルタの一般化について述べ，提案手法について説明を行う．続く，6 章において，提案手法の評価を行う．最後の，7 章にて，本論文をまとめる．

第2章　ロード/ストア命令の投機実行

ロード/ストア命令にかぎらず，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．そのため，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．

しかし，ロード/ストア命令のようにメモリを介する命令間の依存に関しては，実行前に依存を解決することは出来ない．ロード/ストア命令がアクセスを行うメモリのターゲット・アドレスは，オペランドのレジスタ値や即値から**動的**に決定されるからだ．特に out-of-order コアにおいては，ロード命令に関しては，依存元のストア命令がメモリに値を書込む，すなわち依存元のストア命令のコミットまでは依存関係は完全には分からない．このように，ロード/ストア命令間の依存関係がわからないことを，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，その他の命令よりも困難となる．

このメモリ曖昧性を除去して out-of-order にロード/ストアを実行するために，ロード/ストア命令間に存在する依存関係を予測し，事後的に予測と同じ依存関係が実際にあったかどうかの検証を行う投機的な手段が取られている．

本章では，まずロード/ストア命令が実際にメモリにアクセスするタイミングについて説明を行い，その後にロード/ストア命令の out-of-order 実行を可能とする技術について説明を行う．

2.1　ロード/ストア命令の実行とコミット

out-of-order プロセッサにおいて，命令の実行は out-of-order に行われるが，命令のコミットは in-order に行われる．コミットとは，レジスタやメモリなどのアーキテクチャ・ステートの不可逆的更新のことである．この実行とコミットに対してメモリ命令は，以下のように動作する．

ロード命令　実行時にアドレスを計算しメモリ (L1D) から値を読込む．コミット時はメモリに関しては何も行わない．

ストア命令　実行時にはアドレスの計算のみを行う．またコミット時にメモリ (L1D) にデータを書込む，つまりアーキテクチャ・ステートの更新を行う．

このように，ロード命令とストア命令では，メモリ・アクセスのタイミングが異なる．そのため，out-of-order 実行では，依存関係にあるロード命令とストア命令において，ロード命令の実行とストア命令のコミットが入れ替わってしまうと，ロード命令が本来読み込むべきデータを読み込むことができない．これが，メモリ・アクセス順序違反である．

ロード/ストア命令の out-of-order 実行ではプログラムの意味を壊さないために，メモリ・アクセス順序違反の検出を行っている．

2.2　ロード/ストア命令の投機実行を支える技術

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) メモリ依存予測 : 実行の履歴に基づいて，ロード/ストア命令間の依存関係を予測し，ロード/ストア命令を投機的に実行する．
(2) メモリ・アクセス順序違反検出: ロード命令が依存関係にあるストア命令を追い越していたらメモリ・アクセス順序違反として検出する．

本来ならば，(1)，(2) の時系列順に説明をするべきだが，(1) の予測に関しては，依存がない (と予測) としてロード/ストア命令の実行を行い，(2) にてメモリ・アクセス順序違反が検出されたら，予測テーブルに学習する (2) → (1) が予測器の学習の基本的な手順である．そのため，本節ではこれら 2 つの手順を，(2)，(1) の順で説明を行う．また，これら (1)，(2) に加えて，SQ を簡略化する手法として，フォワーディングを投機的に行う手法を提案手法に適用したモデルの評価を 6.4 で行うので，本節でこの投機フォワーディングの説明を行う．

本題に入る前に，まず本論文で用いる簡略化したパイプライン・チャート (**図 2.1**) の見方を説明する．

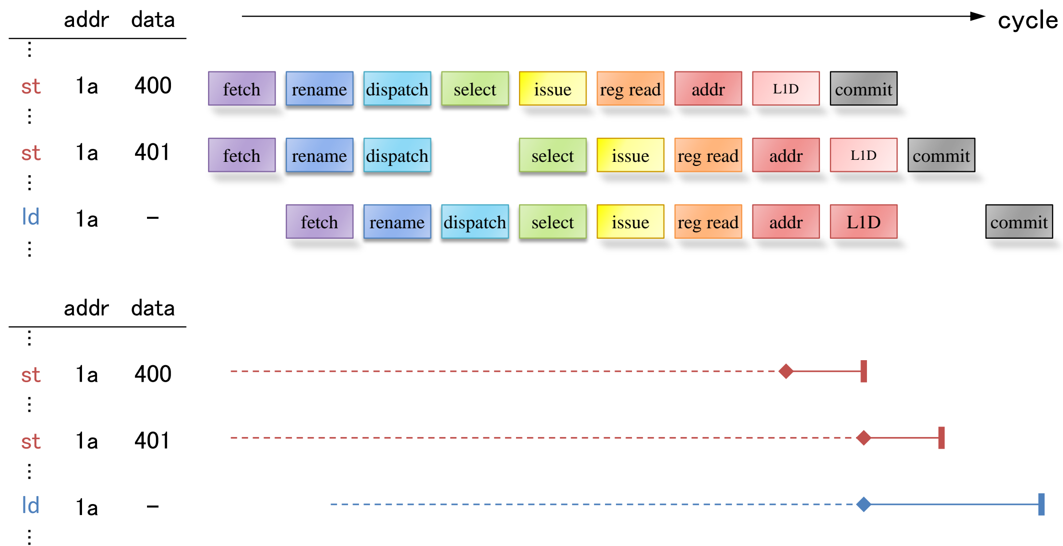

図 2.1: 一般的なパイプライン・チャート (上) と本稿で用いるパイプライン・チャート (下)

2.2.1 本論文で用いるパイプライン・チャート

2.1 節でも説明したように，メモリ・アクセス順序違反とは，依存関係にあるロード命令の実行と，ストア命令のコミット (もしくは実行) のタイミングが入れ替わることで，正しいストア・データをロードできない現象である．

そのため，メモリ・アクセス順序違反検出において意味があるのは，実行ステージとコミットステージである．そのため，図 2.1 上図のような一般的なパイプライン・チャートではなく，図 2.1 下図のようなパイプライン・チャートを用いる．同下図では，フェッチステージから実行ステージまでを破線部，実行ステージからコミットステージまでを実線部で，実行ステージを菱型，コミットステージは縦棒で表している．また書き込みをおこなう処理は赤で，読込を行う処理は青で示されている．

2.2.2 メモリ・アクセス順序違反

図 2.2 に，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

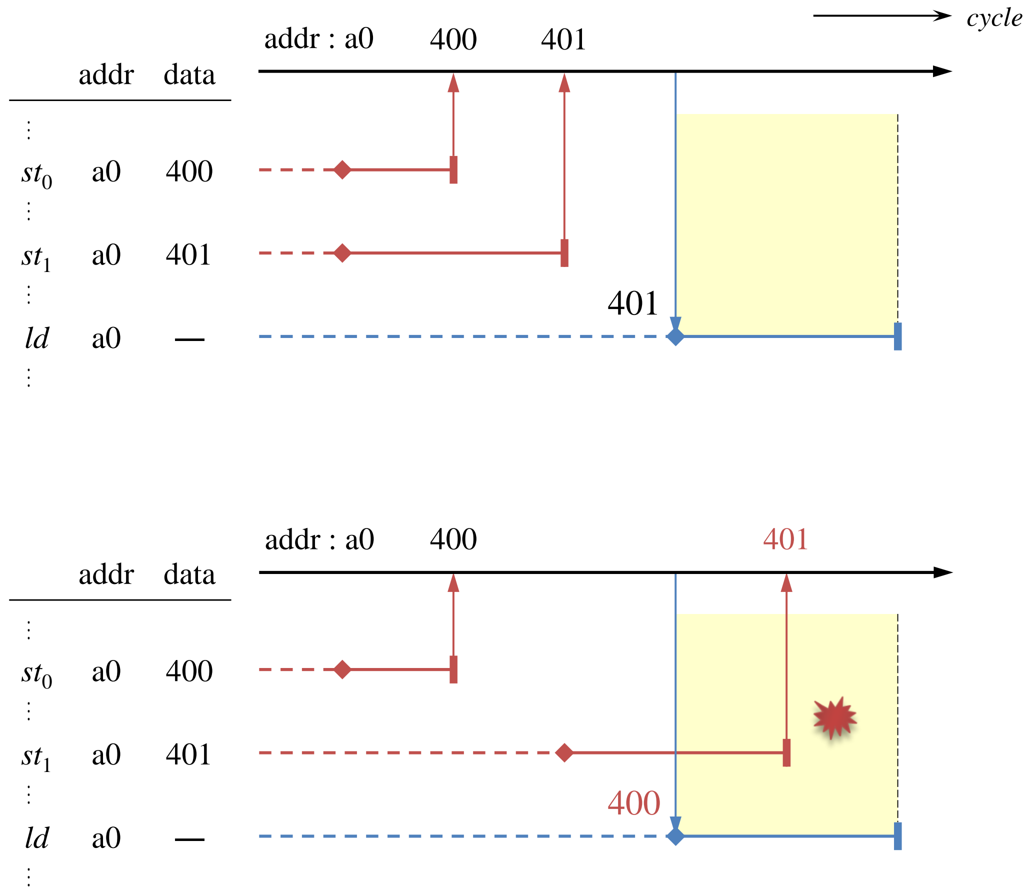

図 2.2: メモリ・アクセス順序違反の例

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，同下図では，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しなければならない．

2.2.3 メモリ依存予測

メモリ依存予測はロード/ストア命令間の依存関係を実行履歴に基づいて予測する．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 [8] や距離ベースの依存予測器 [9, 10] がある．本節ではそれぞれの説明を行う．

2.2.3.1 Store Set 依存予測器

Store Set とは，あるロード命令に対してかつて依存した (主に実行順序違反を引き起こした) ことがあるストア命令の集合を指す．Store Set 予測器では，ロード命令はそれぞれ Store Set を持つ．ある Store Set に登録されているストア命令は，ある Store Set に登録されている全てのストア命令と依存があると予測する．ある Store Set に登録されているストア命令の (予測上の) 依存関係を解消するために，ある Store Set 内のストア命令は全て in-order に実行される．ロード命令は，自身が持つ Store Set に登録されている命令のうち，フェッチ順で直前にあたるストア命令に依存があると予測する．そして，依存元であると予測されたストア命令の実行後に，ロード命令を実行する．

2.2.3.2 距離ベースの依存予測器

距離ベースの依存予測器とは，かつてメモリ・アクセス順序違反を引き起こしたロード/ストア命令の命令間距離を予測テーブルに学習する手法である．そしてロード命令の PC と分岐履歴から生成されたインデクスを用いて，予測テーブルに依存距離を書込 (学習) や読込 (予測) を行う．ここでは，簡単に依存距離の学習についてまとめる．

学習

ロード/ストア命令間の距離の測り方は様々である．例えば，[10] では，依存距離は順序違反を引き起こしたロード命令とストア命令の間に存在するストア命令

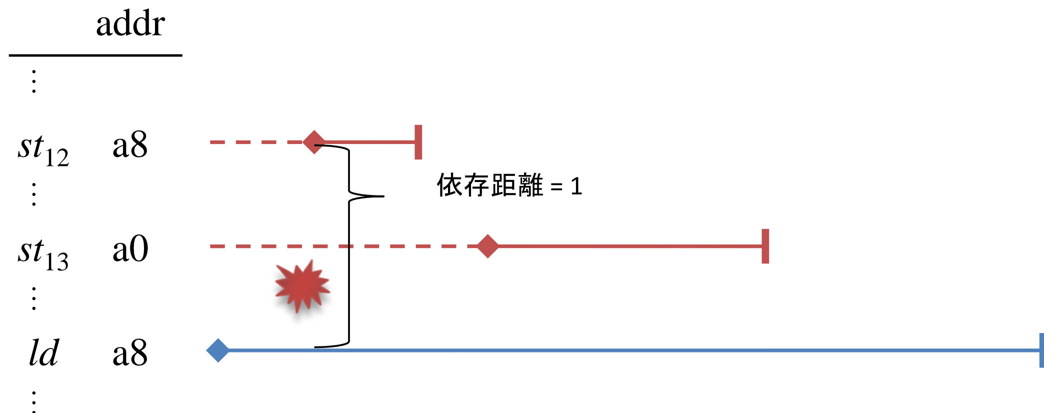

図 2.3: 依存距離の測り方

の数である．**図 2.3** は st12 と Ld とで実行順序違反を引き起こした様子を示している．同図での依存距離は st13 が間に存在するため，1 となる．この依存距離を，ld のリネーム時の分岐履歴とロードの PC から生成されたインデックスを用いて予測テーブルに学習を行う．

2.2.4　投機フォワーディング

先行するストア命令のコミットと，ロード命令の実行が逆転が起きなければメモリ・アクセス順序違反は生じ得ない．しかし，メモリ・アクセス順序違反を引き起こさないために，プロセッサ内に依存するデータが存在するにも関わらず，ロード命令が依存するストア命令のコミットを待つのはやや無駄がある．この無駄な待ち時間を無くすために，バッファからストア・データをロード命令に供給することをストア・データ・フォワーディング (本論文ではこれをフォワーディングと略している) という．フォワーディングを依存予測に基づいて，投機的にデータの授受を行うことを投機フォワーディングという．

投機フォワーディングと CAM による非投機的なフォワーディングで異なる点は，フォワーディングされた命令を実行順序違反として検出する恐れがあるか否かにある．CAM による非投機的なフォワーディングを行うアーキテクチャでは，ロード/ストア命令の実行時に，実行されたすべてのロード/ストア命令のターゲット・アドレスがわかる．ゆえに，ロード命令は実行時に，自分が参照するであろうデータを SQ から入手することが可能である．そのため，先行するストア命令の

実行と後続のロード命令の実行順序の入れ替わりを検出すれば，ロード命令が本来読み込むべきデータと比べ古いデータを参照したこと，つまり実行順序違反が発生したことがわかる．

一方，投機的なフォワーディング手法では，ロード/ストア命令の実行時にわかるのは，せいぜい依存関係にあると予測された命令のターゲット・アドレスくらいである．そのため，投機的なフォワーディングを行うモデルで，実行順序違反検出を行うには，先行するストア命令のコミットと後続のロード命令の実行の入れ替わりを検出しなければ，プログラムの意味を正しく保つことが出来ない．しかし，フォワーディングでは，実行済み未コミットのストア命令から，後続のロード命令へとデータが渡っている．そのため，投機的なフォワーディングを行うアーキテクチャでは，フォワーディングを実行順序違反として検出する恐れがある．

そのため，投機フォワーディングを行うアーキテクチャでは，投機フォワーディングに関わったロード/ストア命令を検出しないようにする必要がある．フォワーディングを検出しないための処理については順序違反検出の手法によって異なるので，本節ではその処理について説明を行わず，3 章などの以降の章に任せる．

ここでは，投機フォワーディングと非投機的なフォワーディングで同様の処理である，フォワーディングと，フォワーディング・ミスの検出について述べる．

図 2.4 はフォワーディング及び，フォワーディング・ミスの様子を表した図である．同上図では，st_1 と ld がアドレスが一致しているので，ストア・データ 399 がフォワーディングが行われている．一方で同下図では，フォワーディングが行われる st_1 と ld の間に，同じターゲット・アドレスにデータ 400 を書き込む st_2 が存在している．これが，フォワーディング・ミスである．

フォワーディングの方法としては，ロード命令が実行時に，先行実行されたストア命令のターゲット・アドレスのターゲットアドレスの等しい物をサーチしてストア・データを得る非投機的手法と，先行するストア命令との依存関係を予測して，依存関係にあると予測したストア命令からストア・データを得る投機的手法がある．とくに，ロード命令が先行実行されたストア命令のターゲット・アドレスをサーチする手法では，優先順位付きの検索が必要となる．

Store Queue Index Prediction

Store Queue Index Prediction(SQIP)[11] などの投機フォワーディングでは，ロード命令は依存予測に基づいて，ストア命令からフォワーディングをうける．SQIP

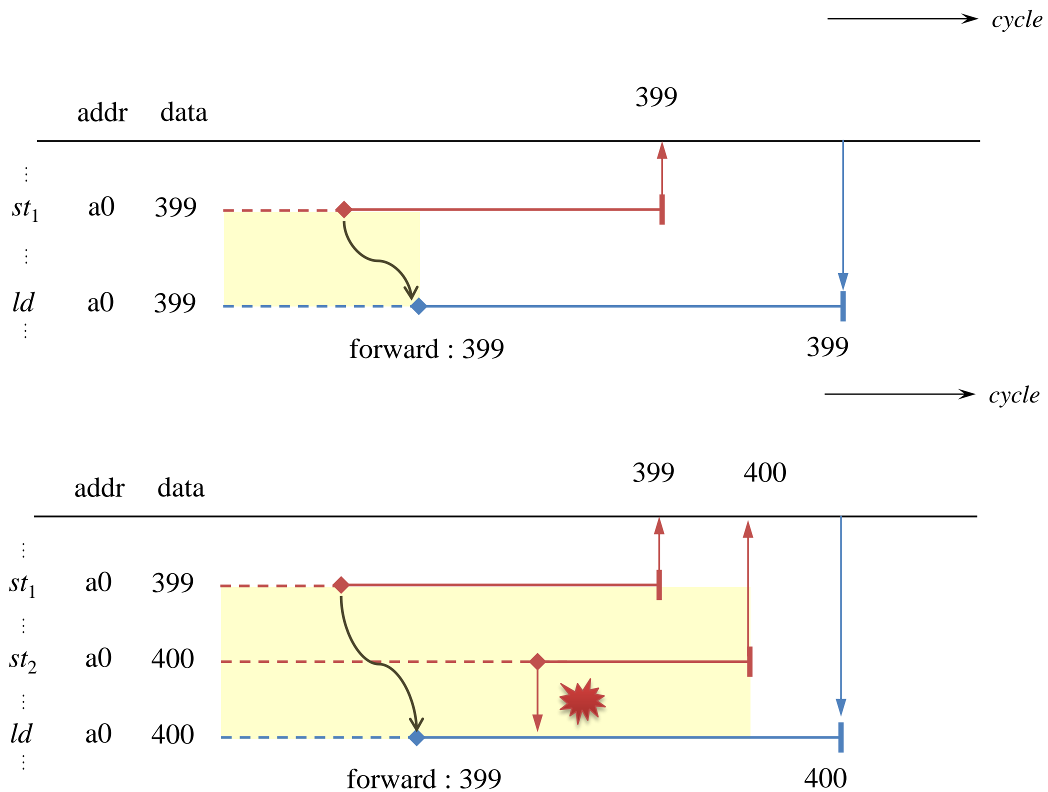

図 2.4: フォワーディングおよびフォワーディング・ミス

では，ロード命令は依存するストア命令が格納されている SQ のインデックスを予測し，そこから依存元と予測されたストア命令のターゲット・アドレスとストア・データを読込む．この依存元と予測されたストア命令のターゲット・アドレスとロード命令のターゲット・アドレスが一致しているときに限り，ストア命令からロード命令に投機的にフォワーディングが行われる．ターゲット・アドレスが一致していない時には，ロード命令は先行するストア命令とは依存が無かったとしてメモリからデータを読み込む．また，SQIP による順序違反検出には，3.1 節で述べる SVW が用いられている．

2.3　順序違反検出/フォワーディングと LSQ の CAM

近年の out-of-order コアでは，ロード・ストア・キューをロード・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・アドレスとストア・データをそれぞれ保持することが一般的になっている．

ロード/ストア命令の投機実行を行う場合には，LQ と SQ は，実行順序違反検出/フォワーディング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 による動的なターゲット・アドレスの比較を行う．

LSQ の CAM の果たす役割

CAM を用いた実装では，LQ/SQ に対して，優先順位付きの連想検索を行うことで，実行順序違反検出/フォワーディングを行っている．そのため，ロード命令は実行時に，SQ 上の実行された全てのストア命令との依存関係をこの連想検索によって知ることができる．ストア命令もまた実行時に，LQ 上の実行された全てのロード命令との依存関係をこの連想検索によって知ることができる．

CAM を用いて実行順序違反検出/フォワーディングを行うモデルでは，実行済みではあるがコミットはしていないストア命令からロード命令はストア・データを受け取ることができる．そのため，2.1 節で説明したような．ストア命令のコミットとロード命令の実行との入れ替わりでは無く，ストア命令の実行とロード命令の実行との入れ替わりで，メモリ・アクセス順序違反の検出が可能となる．

故に，フォワーディングを検出しないような別途制御はLSQ に CAM を用いている限りは必要が無い．ここでは，そのことを念頭に置いて，LQ，SQ のそれぞれの役割について述べる．

SQ

2.1 節で述べたように，L1D の更新は不可逆的に行われる．一方でフォワーディングはその不可逆的更新に先立って，ストア命令の実行〜コミットの間に依存関係にあるロード命令にそのストア・データを渡す．このストア・データを受け取るために，ロード命令は実行時に自身のターゲット・アドレスをキーとして SQ に連想検索をかけ，ターゲット・アドレスが一致するストア命令があれば，このエントリのストア・データを受け取る．

LQ

メモリ・アクセス順序違反の検出はストア命令の実行時に自身のターゲット・アドレスをキーとして LQ に連想検索をかけることによって行う．このとき，LQ に

自分より後続の実行済みロード命令でかつターゲット・アドレスが一致するものが存在すれば，これをメモリ・アクセス順序違反として検出を行う．

LSQ の CAM の省略

LQ のポート数はたかがかストア命令の同時発行数あれば十分なのに対し，SQ のポート数はロード命令の同時発行数必要である．一般的に，ロード命令の数はストア命令の数よりも多いので，ロード/ストアの同時発行数はロード命令の方が大きくなる．そのため，LQ と比べて，SQ はエントリ数は小さいがポート数は多い構成になる．RAM は構造上，その面積はポート数の 2 乗に比例して大きくなるため，SQ の面積はエントリ数の割に大きくなってしまう．

ゆえに，LSQ の CAM を省略するのにあたって，LQ の CAM だけでなく SQ の CAM を省略するのが，より重要となる．

第3章　フィルタを用いたメモリ・アクセス順序違反手法

ロード/ストア命令を out-of-order に実行するには，同一アドレスへの実行順序の逆転を検出して，回復処理を行う必要がある．out-of-order に実行されるロードのうち，実際にメモリ・アクセス順序違反を引き起こす命令はわずか 0-7% であることが知られている [8]．そのため，実行される全てのロード命令に対して，メモリ・アクセス順序違反が発生したかを調べるために，先行する全てのストア命令とアドレスの一致比較を行って確認検査するのは無駄が多い．また，2.3 節でも述べたように，CAM を用いた確認検査手法では面積の増加と，それに伴う消費電力の増大といった問題もある．

そこで，メモリ・アクセス順序違反を引き起こさないと判明した命令をフィルタリングすることで，メモリ・アクセス順序違反の確認検査の頻度を減らすフィルタを用いた手法が有る．

フィルタを用いた手法ではロード/ストア命令がフィルタに要素を登録/参照/削除といった手続きを踏むことで，メモリ・アクセス順序違反を生じ得ない命令を判定する．この判定には，偽陽性はあるが，偽陰性はない．それゆえ，実際にメモリ・アクセス順序違反が起きていない偽陽性の判定については，確認検査が必要となる．この確認検査は必ずしも，面積コストの高い LQ-CAM で行う必要はない．フィルタによる手法での確認検査は，微々たる性能低下を伴うが，面積コストに優れる検査方法を採用しているものもある．

本章では，順序違反検出を行うフィルタとして，Store Vulnerability Window, パラレル・カウンティング・ブルーム・フィルタ，そして Delayed Memory Dependence Checking について説明を行う．特に以下の説明で，フィルタに登録される要素や，登録などのフィルタに対する手続きの数が，手法によって異なることに留意していただきたい．

3.1 Store Vulnerability Window

Store Vulnerability Window(SVW) [5] では，ロード/ストア命令がフィルタに要素を登録/参照をすることで，先行するストア命令のコミットを追い越して実行されたロード命令を検出している．SVW でフィルタに登録/参照される要素は，ストア命令にフェッチ順に割り振られた，Store Sequence Number(SSN) とよばれるシーケンス・ナンバである．SVW でのフィルタへの要素の登録/参照手順は，
(1) ストア命令がコミット時に，フィルタの該当エントリへ自身の SSN を登録する．
(2) ロード命令が実行時に，フィルタに書き込まれている最大の要素を参照する．
(3) ロード命令がコミット時に，フィルタの該当エントリから要素を読込む．
の 3 つの手順からなっており，(2) と (3) で読み込まれる要素の値の大小によって，実行順序違反を引き起こし得るロード命令を検出する．この大小比較によって検出された命令が，実際にメモリ・アクセス順序違反を引き起こしているかの確認検査は，ロード再実行によって行われる．

3.1.1 SVW の順序違反検出

SVW[5] では，読み書きが行われるフィルタをブルーム・フィルタとしているが，ハッシュ関数を 1 つしか用いないフィルタであるので，正確にはブルーム・フィルタではない．本稿ではブルーム・フィルタではないことを明確にするため，SSN を登録するフィルタを SSN Table(SSNT) と呼ぶことにし，ストアに動的に割り当てられるシーケンス・ナンバは [5] に同じく StoreSequence Number(SSN) とした．

図 3.1 は，SVW においてメモリ・アクセスが正しく行われている様子 (同図上) と，同一アドレスへ読み書きを行う命令のアクセス順序が入れ替わり違反が検出されている様子 (同図下) を表した図である．

この図においてはストア命令は st と，ロード命令は ld としている．そして，st_{12}，st_{13}，ld の順にフェッチされており，ストアに割り当てられている SSN は上から順に，12，13 となっている．また，SSNT はメモリ命令のターゲット・アドレスをハッシュ値として SSN を記録するテーブルであり，max は SSNT に書き込まれている最大値，つまり最後に書き込んだ (コミットした) ストア命令の SSN である．図の横軸上部の 400，401 などの数値は，アドレス $a0$ に格納されてるデータである．

図 3.1 上図では，ld がメモリから読み込む値は，st_{13} がメモリに書き込むデー

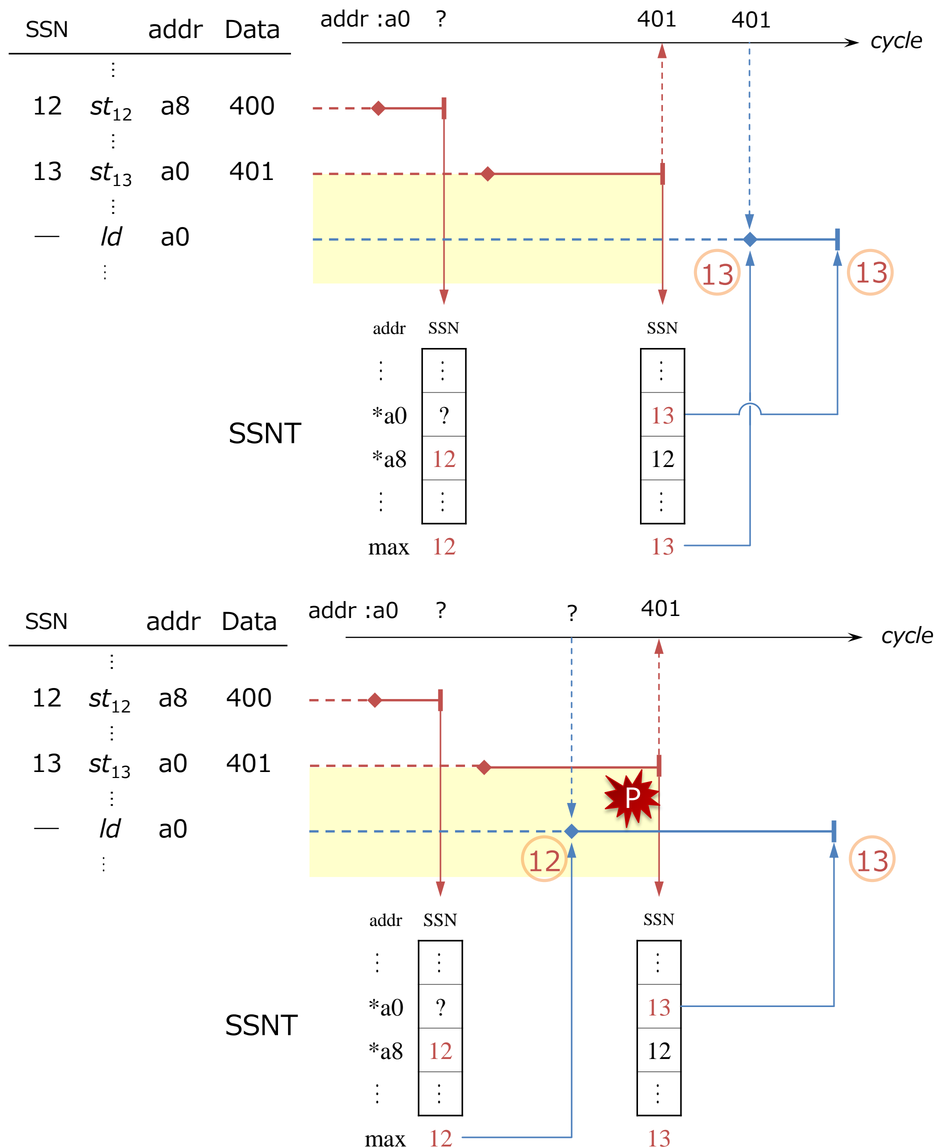

図 3.1: SVW の順序違反検出：順序違反がない場合（上）とある場合（下）

タ 401 を読み込むので，メモリ・アクセス順序違反は生じていない．図 3.1 上図では，st_{12} はコミット時に SSNT の該当エントリである $*a8$ に 12 を，st_{13} は $*a0$ に 13 を書き込んでいる．ld は実行時に SSNT の最大値 13 を，コミット時に SSNT の該当エントリ $*a0$ から 13 を読み込む．同図において，ld が実行時とコミット時に SSNT から読み込む値は等しい．これは，st_{13} のコミットよりも後に ld が実行されたことを意味する．このように，ロード命令がコミット時に SSNT から読み込む値が，実行時に SSNT から読み込む値よりも小さい，もしくは等しければ，メ

モリから正しいデータを読み込んだことが保証され，メモリ・アクセス順序違反は検出されない．

図 3.1 の下図は，ld の実行が早まることで，本来読み込むべきデータ 401 ではないデータ 400 を読み込む，つまりメモリ・アクセス順序違反が生じている例である．同下図ではロード命令の実行が早まったことで，実行時に読み込む値は 12 となるが，コミット時に読み込む値は同上図とは変わらず 13 となる．

そのため，ロード命令がコミット時に SSNT から読み込む値 13 が，実行時に SSNT から読み込む値 12 と比べて大きくなる．これは，先行制約のあるストア命令のコミットに先んじて，ロード命令が実行されたことを意味する．このように，ロード命令がコミット時に SSNT から読み込む値が，実行時に読み込む値よりも大きい時に，SVW ではロード命令を メモリ・アクセス順序違反を引き起こしたとして検出する．

3.1.2　SVW による フォワーディングの処理

フォワーディングとは，ロード命令がメモリを介さずに，ストア命令からストア・データを受け取ることである．ストアのコミットに先んじて，ロードが実行されることにより，データが授受されることを意味する．そのため，フォワーディングするストア命令のコミットと，フォワーディングされるロード命令の実行の順序は当然，逆転する．フィルタによる順序違反検出手法においては，フォワーディングによるストアのコミットとロードの実行の逆転を検出しない制御を行う必要がある．SVW では，フォワーディングされたロード命令が実行時に読み込む値を，フォワーディングを行ったストア命令の SSN にすることによって，この問題を回避している．

図 3.2 は SVW におけるフォワーディングの制御を表した図である．同図では，実際の依存関係どおりにフォワーディングが正しく行われている様子 (同図上) と，実際の依存関係とは異なるストア命令からフォワーディングが行われたときに順序違反として検出される様子 (同図下) を示している．

ここでは，上図においては st_{12} と ld，下図においては st_{13} と ld が依存関係にある命令である．図 3.2 では依存関係にあると判断された st_{12} と ld の間でフォワーディングが行われているが，下図では依存関係に無いため，フォワーディング・ミスが生じている．

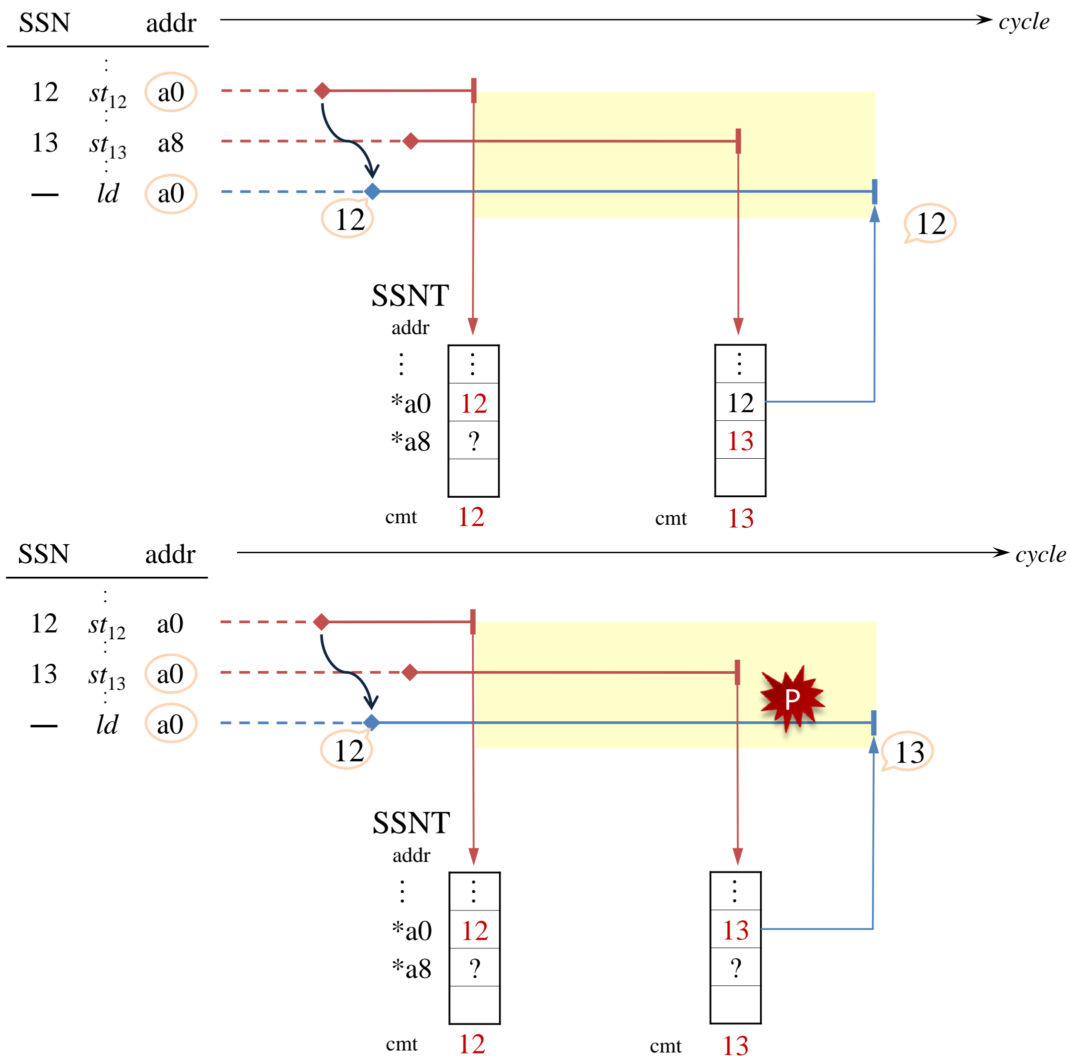

図 3.2: SVW におけるフォワーディングの処理：ミスがない場合 (上) とミスがある場合 (下)

二つの図におけるロード命令は，どちらも実行時に SSNT の最大値を読み込むのではなく，フォワーディングを行った st_{12} から 12 を読み込んでいる．上図ではコミット時に 12 を読み込んでいる．このとき，実行時とコミット時で値が等しいことから，フォワーディングを行った st_{12} と ld において，正しくフォワーディングが行われたことが保証される．一方，下図ではコミット時に 13 を読み込むため，実行時とコミット時で読み込む値が異なる．これにより，st_{12} と ld の間でフォワーディングした命令と異なる命令による書込があったことを検出している．

3.1.3 SVW の問題点

SVW は，LQ の CAM を排除することができるが，フィルタの構成要素がシーケンス・ナンバであること，フィルタの登録がストア命令のコミット時であることに問題がある．

シーケンス・ナンバ:SSN の問題　これは SSN のビット数が 16bit と多くなり，フィルタの容量が大きくなる，またフィルタに複数のハッシュ関数を用いることが知られていないことに問題がある．後者の問題は，順序違反検出において，偽陽性発生率が高くなるため大きな問題であったが，後に述べる提案手法により緩和される．

アクセス手順の問題　アクセス手順の問題は，Store Set 依存予測器 [8] のような依存関係にあるメモリ命令を特定するようなメモリ依存予測の学習において大きな問題が生じる．Store Set 予測器では過去に起きた，メモリ・アクセス順序違反をもとに学習を行うが，SVW において違反の検出時には実際に依存していたストア命令はリタイアしている．そのため，違反したロード命令と依存関係にあるストア命令を特定するためには，別途制御 (SPCT) が必要である．

3.2 パラレル・カウンティング・ブルーム・フィルタを用いた手法

パラレル・カウンティング・ブルーム・フィルタ (PCBF) を用いた手法では，ロード/ストア命令がフィルタに要素を登録/参照/削除することで，ロード命令の実行からコミットまでに，実行される先行ストア命令を検出している [2, 12]．PCBF はブルーム・フィルタ (BF) の一種であるが，BF については，4 で詳しく述べるため，ここではフィルタに要素を登録/参照/削除することで，要素が集合に登録されているかどうかを判定するためのフィルタと認識していただきたい．PCBF のフィルタの構成要素はカウンタになっており，カウンタへの要素の登録/削除はカウンタのインクリメント/デクリメントで実現されている．

PCBF でのフィルタの登録・参照・削除の手順は，
(1) ロード命令が実行時に，フィルタの該当エントリへ要素を登録 (インクリメント)

(2) ストア命令がコミット時に，フィルタの該当エントリから要素を参照
(3) ロード命令がコミット時に，フィルタの該当エントリから要素を削除 (デクリメント)
の 3 つの手順からなっている．あるエントリについて，ロード命令が要素を (1) 登録してから，(3) 削除するまでに，ストアによる (2) 参照が有れば，このストア命令を構造のロード命令によってメモリ・アクセス順序違反されうる命令として検出する．またこのときの検出が偽陽性であるか否かの確認検査は，検出を行ったストア命令のターゲット・アドレスで LQ をシーケンシャル・サーチすることで実現されている．

本節では，本手法のよるメモリ・アクセス順序違反の検出を最も簡単な BF を用いて説明を行う．BF については，後の 4 章で詳しく説明を行う．そのため，ここでは，BF とはアドレスをハッシュ値として，ビットを登録/参照/削除するテーブルである程度の理解にとどめてもらいたい．

3.2.1 PCBF の順序違反検出

図 3.3 は，上図がメモリ・アクセスが正しく行われている様子で，下図が同一アドレスへ読み書きを行う命令のアクセス順序が入れ替わり違反が検出されている様子である．

同図においてはストア命令は st とロード命令は ld としている．また st_0，st_1，ld の順にフェッチされている．同図中の Bloom Filter はアドレスをハッシュ値とするテーブルで，図の横軸上部の 400，401 などの数値は，アドレス $a0$ に格納されているデータである．

図 3.3 上図では，ld がメモリから読み込む値は，st_1 がメモリに書き込むデータ 401 であるので，メモリ・アクセス順序違反は生じていない．図 3.3 上図では，st_0，st_1 はコミット時に，それぞれの該当エントリから 0 を参照 (読込) している．また同上図の ld は実行時に BF の該当エントリを登録 (インクリメント) するが，ld のコミットまでに先行するストア命令による同一アドレスの書込が生じていない．そのため，同上図ではメモリ・アクセス順序違反は検出されず ld はコミット時に該当エントリを削除 (デクリメント) を行っている．

図 3.3 の下図は，st_1 のコミットが遅れることで，ld が本来読み込むべきデータ 401 ではないデータ 400 を読み込む，つまりメモリ・アクセス順序違反が生じている例である．同下図では st_1 のコミットが遅れたことにより，ld によって登録され

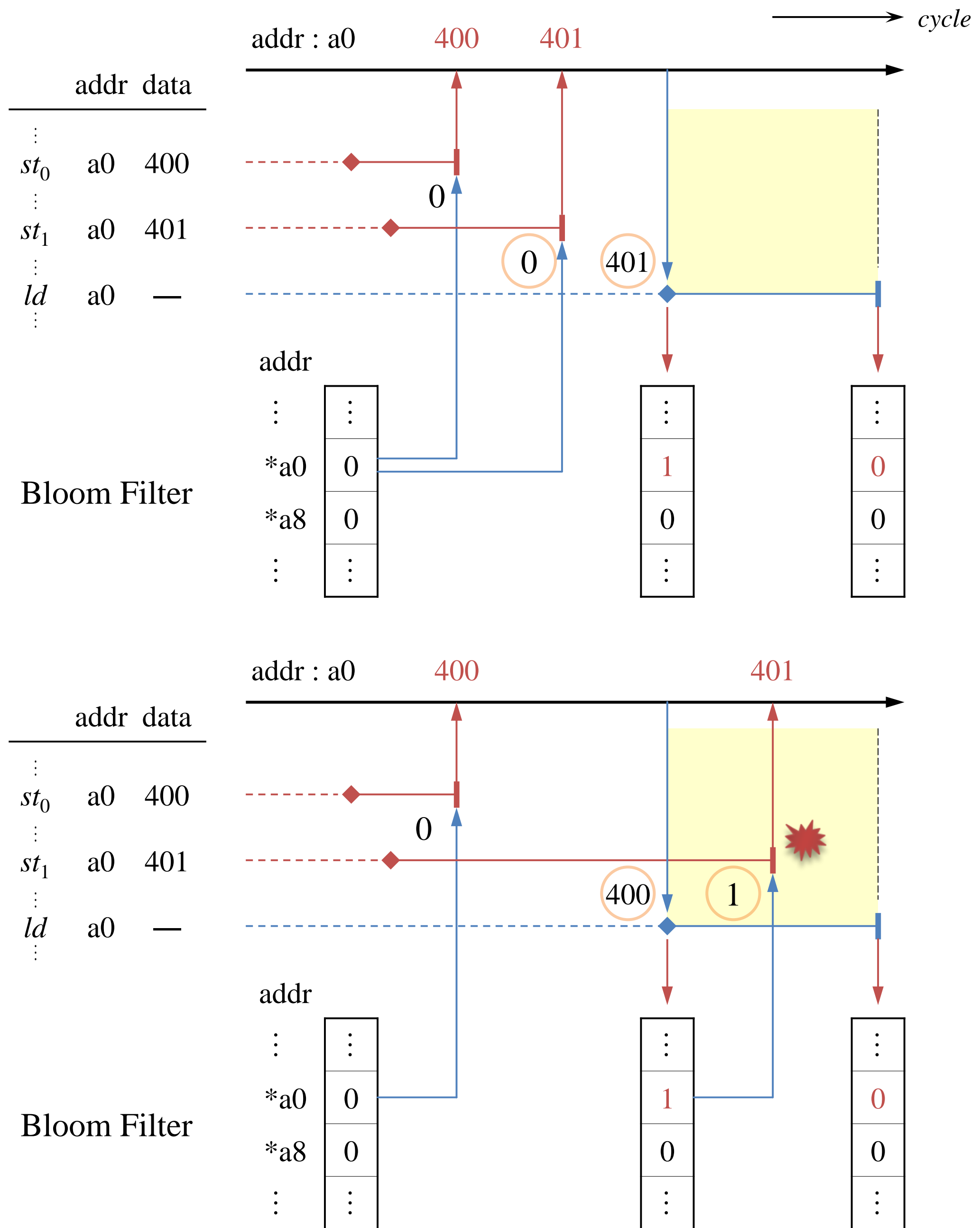

図 3.3: PCBF における順序違反検出:順序違反がない場合（上）とある場合（下）[2]

た要素が消去される間に BF の参照を行う．st_1 は参照したエントリに要素が登録されていることから，メモリ・アクセス順序違反された可能性のある命令として検出される．

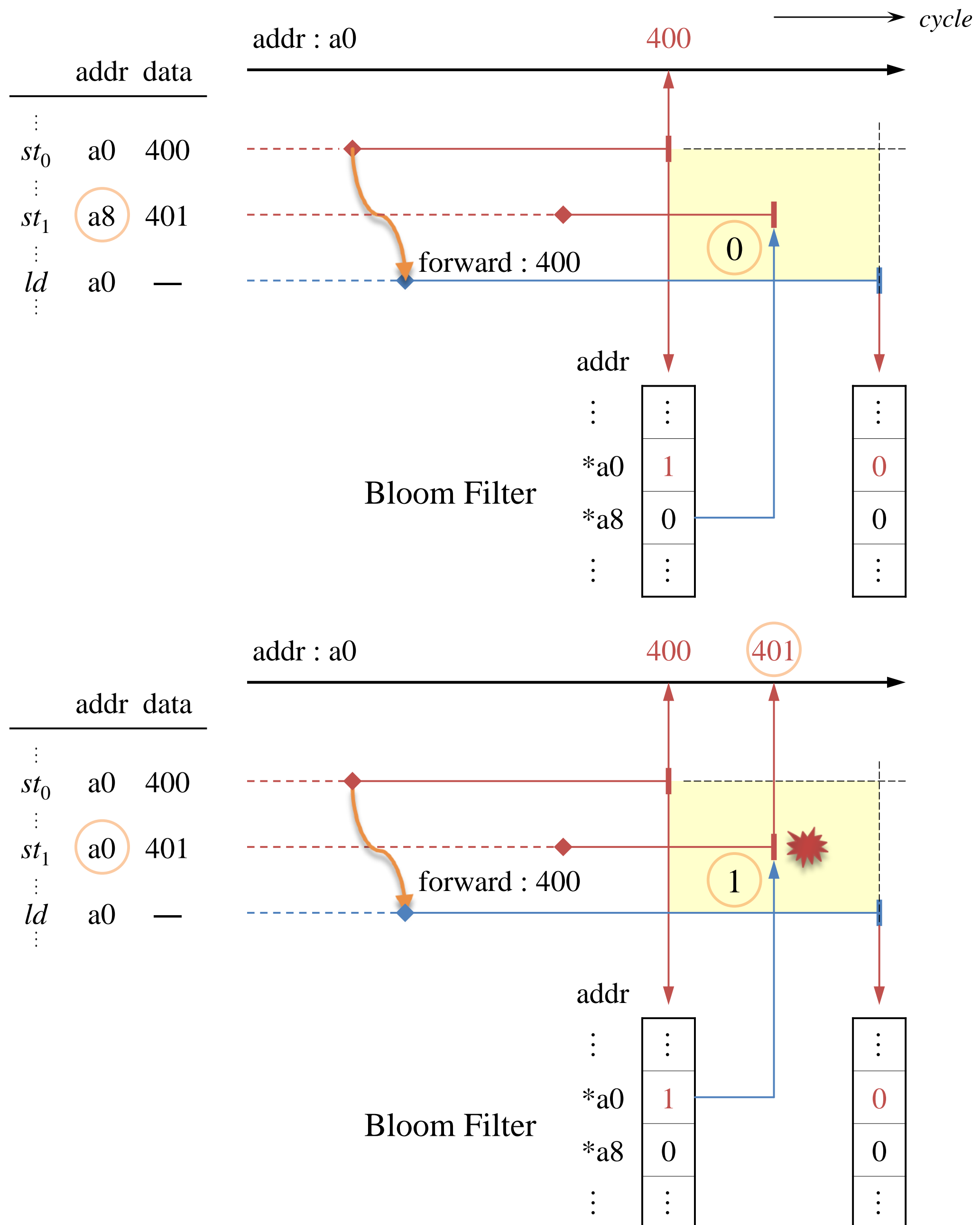

図 3.4: PCBF におけるフォワーディングの処理：ミスがない場合 (上) とミスがある場合 (下)[2]

3.2.2　フォワーディングの処理

フォワーディングとは，ロード命令がメモリを介さずに，ストア命令からストア・データを受け取ることである．ストアのコミットに先んじて，ロードが実行

されることにより，データが授受されることを意味する．そのため，フォワーディングするストア命令のコミットと，フォワーディングされるロード命令の実行の順序は当然，逆転する．フィルタによる順序違反検出手法においては，フォワーディングによるストアのコミットとロードの実行の逆転を検出しない制御を行う必要がある．

PCBF では，フィルタへの要素の登録タイミングをずらすことによって，フォワーディングによる逆転を検出しないように制御している．

図 3.2 は BF におけるフォワーディングの制御を表した図である．同図では，実際の依存関係どおりにフォワーディングが行われている様子 (同図上) と，際の依存関係とは異なるストア命令からフォワーディングが行われたときに順序違反として検出される様子 (同図下) を示している．

ここでは，上図においては st_0 と ld，下図においては st_1 と ld が依存関係にある命令である．同図においては，依存関係にあると判断された st_1 と ld の間でフォワーディングが行われているが，下図では依存関係に無いため，フォワーディング・ミスが生じている．

BF を用いる手法では，フォワーディングされたロード命令は，実行時にフィルタへの要素の登録を行わない．そのかわりに，フォワーディングを行ったストア命令がコミット時にフィルタへ要素を登録するように制御を行う．

図 3.2 上図においてフォワーディングを行った st_0 が要素をフィルタに登録 (インクリメント) し，フォワーディングされた ld がこの要素をデクリメントする．これにより，ストア命令はフォワーディングを行ったロード命令を検出しないようにしている．下図では，フォワーディングを行った st_0 と ld の間に存在する st_1 がコミット時の参照において，st_0 が登録した要素を参照する．この参照によって，st_0 は自分に先行された ld 命令が自分と異なるストア命令からフォワーディングされた (もしくは普通の実行順序違反) として検出し，コレをフォワーディング・ミスとして検出する．

3.3 Delayed Memory Dependence Checking

Delayed Memory Dependence Checking(DMDC) では 2 段のフィルタに要素を登録することで，メモリ・アクセス順序違反を検出する手法である [6].

2 段のフィルタの役割が異なり，1 段目のフィルタでは，後続のロード命令の実

行と実行の順序が入れ替わってしまったストア命令を検出をしており，2 段目のフィルタでは，1 段目で検出されたストア命令とターゲット・アドレスが一致するロード命令を検出している．ここで，1 段目のフィルタで，ストア命令と実行と，後続ロード命令の実行の順序の入れ替わりを検出していることから分かるように，DMDC では，SQ-CAM によるフォワーディングを前提としている．

1 段目のフィルタでは登録/参照される要素は，ロード/ストア命令にフェッチ順に割り振られた Load Store Number(LSN) と呼ばれるシーケンス・ナンバである．また，2 段目のフィルタで登録/参照/削除される要素は，ビットである．

DMDC におけるフィルタへの登録/参照/削除の手順は
(1) ロード命令が実行時に，1 段目のフィルタの該当エントリへ自身の LSN を登録.
(2) ストア命令が実行時に，1 段目のフィルタから該当エントリの要素を参照.
(3) 1 段目のフィルタにより検出されたストア命令のみが実行時に，2 段目のフィルタの該当エントリへ要素を登録
(4) ロード命令のコミット時に，2 段目のフィルタから該当エントリの要素を参照.
(5) 1 段目のフィルタで検出されたストア命令を追い越し得るロード命令が，すべてリタイアした時，2 段目フィルタの全ての要素を削除.
の 5 つの手順からなる．1 段目フィルタの (1) と (2) とで読み書きされた値の大小関係を比較することで，後続のロード命令の実行に遅れて実行されたストア命令を検出する．そして，1 段目フィルタで検出されたストア命令のみが 2 段目のフィルタに書込 (登録) を行い，その書込先を読み込んだロード命令がメモリ・アクセス順序違反を起こし得る命令として検出される．そして，2 段目のフィルタの要素の削除は，(5) のように，ストア命令と実行順序違反し得るロード命令がすべてリタイアした時に行われる．

また，DMDC ではメモリ・アクセス順序違反が実際に生じていたのかの検査は，従来の LQ-CAM による検査によって行われてる．これは，DMDC では他のフィルタによる手法とは異なり，LQ-CAM による優先順位付き検索の頻度を下げることを大きな目的としているためである．

また，冒頭で述べたように，DMDC は SQ-CAM による非投機的なフォワーディングを前提としているため，フォワーディングを行ったロード/ストア命令を検出しないための特別な処理は，投機フォワーディングを行わない限り必要が無い．

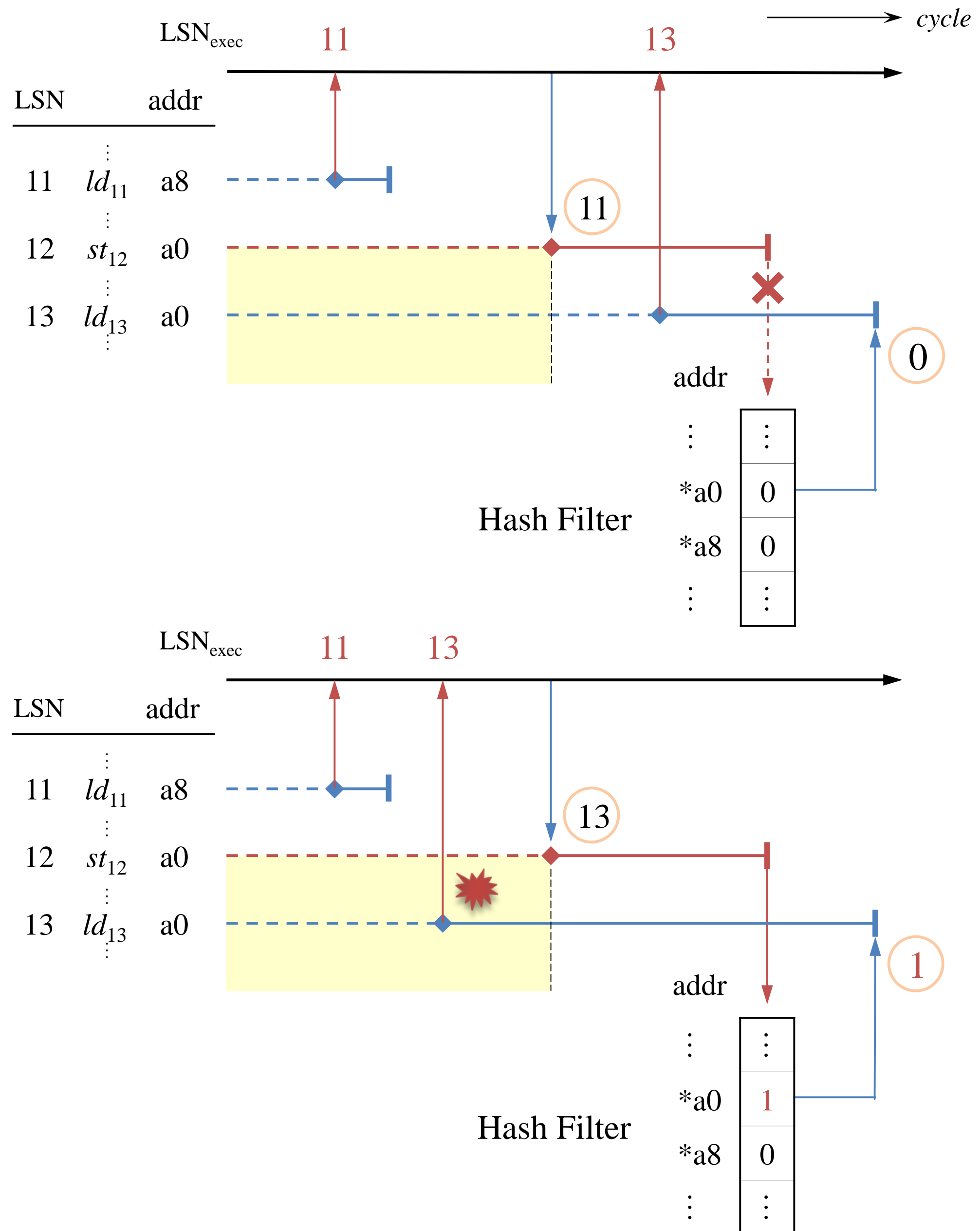

図 3.5: DMDC における順序違反検出:順序違反がない場合（上）とある場合（下）

3.3.1 DMDC の順序違反検出

図 3.5 は DMDC においてメモリ・アクセスが正しく行われている様子 (同図上) と，同一アドレスへ読み書きを行う命令のアクセス順序が入れ替わり違反が検出されている様子 (同図下) を表した図である．また同図上部の右矢印は 1 段目のフィ

ルタに書き込まれている値の変化を表してる．

この図においてはストア命令は st とロード命令は ld としている．また ld_{11}, st_{12}, ld_{13} の順にフェッチされており，メモリ命令に割り当てられているシーケンス・ナンバは上から順に，11, 12, 13 となっている．

図 3.1 上図では，ld_{13} がメモリから読み込む値は，st_{12} がアドレス $a0$ に書き込むデータであるので，メモリ・アクセス順序違反は生じていない．同図では，ld_{11}, ld_{13} は実行時にメモリからデータを読み込むと同時に，1 段目のフィルタの の該当エントリに，自身の LSN である，11，13 を書き込む．

st_{12} は実行時に 1 段目のフィルタに書かれている 11 を読込，自分の LSN と読み込んだ値を比較し，自分の LSN のほうが読み込んだ値より大きければ，ストア命令の実行とロード命令の実行が入れ替わっていないとする．そのため，同上図では，ストア命令は 2 段目のフィルタへの書込は行わない．そしてロード命令は，コミット時に 2 段目のフィルタから 0 を読み込むことで，実行順序違反が発生しなかったと判断する．

一方，同下図では，ld_{13} の実行が早まることによって st_{12} と ld_{13} とでメモリ・アクセス順序違反が生じている図である．

同図において，st_{12} が実行時に 1 段目のフィルタから読み込む値は，13 となるため，1 段目のフィルタで，このストア命令はあるロード命令と実行順序違反を引き起こし得る命令であるとして検出する．そして，1 段目のフィルタで検出された st_{12} は，2 段目のフィルタの自身の該当エントリに，ビットをたてる．ld_{13} はコミット時に 2 段目のフィルタから st_{12} が書き込んだ値を読込，これを実行順序違反として検出する．

3.3.2 DMDC の問題

DMDC では，ストア命令の実行とロード命令のコミットの入れ替わりを，1 段目のフィルタで検出している．しかし，この検出は SQ の CAM を用いたフォワーディングを前提とする方法であり．SQ-CAM の排除には至っていない．もし，DMDC で SQ-CAM の排除を試みようとするならば，ストア命令のコミットとロード命令の実行の順序が入れ替わりを検出するフィルタに 1 段目のフィルタを変更することになる．

しかし，その一方で，このようにフィルタへと変更を加えた時，フォワーディングが行われたロード命令は先行するストアのコミットを追い越すため，1 段目の

フィルタでフォワーディングを行ったストア命令が必ず検出されてしまう．また，フォワーディングを行ったストア命令と，フォワーディングが行われたロード命令とでは，ターゲット・アドレスが必ず一致するため，2 段目フィルタでもまた実行順序違反を引き起こしうる命令だとして検出されてしまう．つまり，フォワーディングが行われたロード命令は，正しくストア・データを得ているのにも関わらず検出されてしまう．

このように，投機フォワーディングを適用しようと思うと，フォワーディングに関わった命令は全て実行順序違反を引き起こしうると判定されてしまい，フィルタの偽陽性発生率が大きくなってしまう．DMDC において，フォワーディングを検出しないようなフィルタの制御については未だに知られていない．

第4章　ブルーム・フィルタ

ブルーム・フィルタ (BF) とは，**与えられたキーが集合の要素として含まれるかどうか**を効率よく判定するデータ構造の 1 つである [7]．BF は k 個のハッシュ関数を用いることに大きな特徴が有り，ハッシュ値に対応するビットをセット/チェックすることで，キーが集合の要素であるかどうかの判定を行う．この判定には，偽陽性はあるが，偽陰性はない，

また複数のハッシュ関数を使うため，1 つのハッシュ関数のみを用いるハッシュ・フィルタ (HF) と比較して，偽陽性発生率が小さいことに大きな特徴がある．

ここでは，まず HF と BF の 2 つのフィルタの動作についての説明を行う．そして，BF の陽性率の解析解を示し，低い偽陽性率を実現していることを説明する．

4.1　ハッシュ・フィルタの動作例

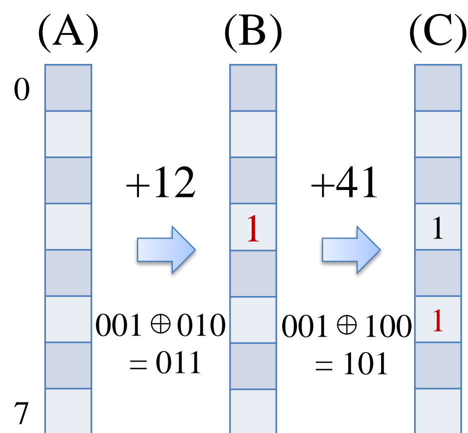

図 4.1: ハッシュ・フィルタ の例

図 4.1 は HF において，キー値を登録する様子を表した図である．同図 (A) では，HF は何も書き込まれていない初期状態を表している．図 4.1 の HF において，キー値は八進数で 00 から 77 までの 64 通り，ハッシュ関数は，キー値の 8 の位と，1 の

位を XOR したもの 1 種 (k = 1) とし，そしてエントリ数 m は m=8 とした．また，以下の例において，ハッシュ値の計算は 2 進数表示で行っていることに注意していただきたい．

いま，HF に要素 12 を登録すると，12 のハッシュ値は左側の桁である 001 と右側の桁である 010 を XOR した 011 であるので，該当要素 011 にビットがセットされる (同図 (B))．このとき，ある要素がフィルタに登録されているかどうかを考えよう．要素 12 に対して登録判定を行うと，該当エントリにビットがセットされているため，HF は陽性を示す．これは図 4.1 (B) にて，登録した要素 12 を示しているため，真陽性となる．要素 41 に関してはそのハッシュ値は，101 となり，該当エントリにはビットがセットされていないので，HF は陰性を返す．一方で要素 56 に対して判定するとき，ハッシュ値は 5 = 101 と 6 = 110 の XOR を取った 011 になる．このとき，図 4.1 (B) で追加した要素 12 とハッシュが衝突しているため，HF は陽性を返す．しかし，要素 56 は実際には登録していないため，HF の判定は偽陽性となる．

次に，要素 41 をフィルタに登録すると，HF の状態は，図 4.1 (C) のようになる．図 4.1 (C) の状態で，ある要素の集合に含まれるかどうかの判定が真陽性，偽陽性，陰性を返すそれぞれの場合の数について考えてみる．

(1) 真陽性について

これは，図 4.1 (B),(C) で登録した 12，41 の 2 通りである．

(2) 偽陽性について

いま，ハッシュ関数として一様なハッシュ値をとる XOR を使っているので，ある要素のハッシュ値は，8 エントリに対して均等に分配される．図 4.1 (C) では，フィルタのエントリのうち，2 ビットが立っている．64 通りのキー値が均等に 8 箇所にバラけるとき，ビットが立っている 2 エントリにヒットする確率は，2/8 であるので，この場合の数は $64 \times 2/8 = 16$ 通り．偽陽性となるのは，これから真陽性である 2 通りを除いた，14 通りである．

(3) 陰性について

全ての場合の数である 64 通りから (1)，(2) の場合の数を除いて 48 通りである．

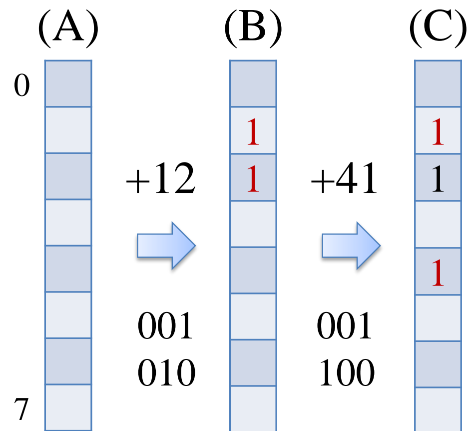

図 4.2: ブルーム・フィルタの例

4.2　ブルーム・フィルタの動作例

BF は本章の冒頭でも述べたが，複数のハッシュ関数を用いることに大きな特徴がある．**図 4.2** は BF にキー値を登録する様子を示したものである．同図 (A) はフィルタへの登録が行われていない初期状態を示す．図 4.1 とこの図においては，キー値およびエントリ数は同一であるが，用いるハッシュ関数が，キー値の八の位 (上位の桁) と，一の位 (下位の桁) の k = 2 種である点において違いがある．

HF の例と同様にして，BF に要素 12 を登録する様子が図 4.2 (B) である．BF では，キー値の上位の桁と下位の桁の二種であるため，001 と 010 の二箇所にビットがセットされる．図 4.2 (B) の状態で，ある要素が登録されているかどうかの判定を行うことを考えてみる．HF の例と登録されている要素 12 については陽性を，登録されていない要素 14 に対しては陰性を示す．その一方で，要素 56 は BF では，101 と 110 になる．このとき，図 4.2 (B) では，要素 56 に該当するエントリにビットはたっていない．このように，HF では偽陽性を示していた要素 56 が BF では陰性を示す．

続いて，要素 14 をフィルタに登録した様子が，図 4.2 (C) にあたる．ここでは，要素 12 と要素 14 とで，ハッシュ値が衝突を引き起こしている BF ではこれを暗黙的に上書きを行っている．図 4.2 (C) の状態で BF が，ある要素の集合に含まれるかどうかの判定が真陽性，偽陽性，陰性を返すそれぞれの場合の数について考えてみる．

(1) 真陽性について

図 4.2 (A)，(B) で登録された 12，41 の 2 通りである．

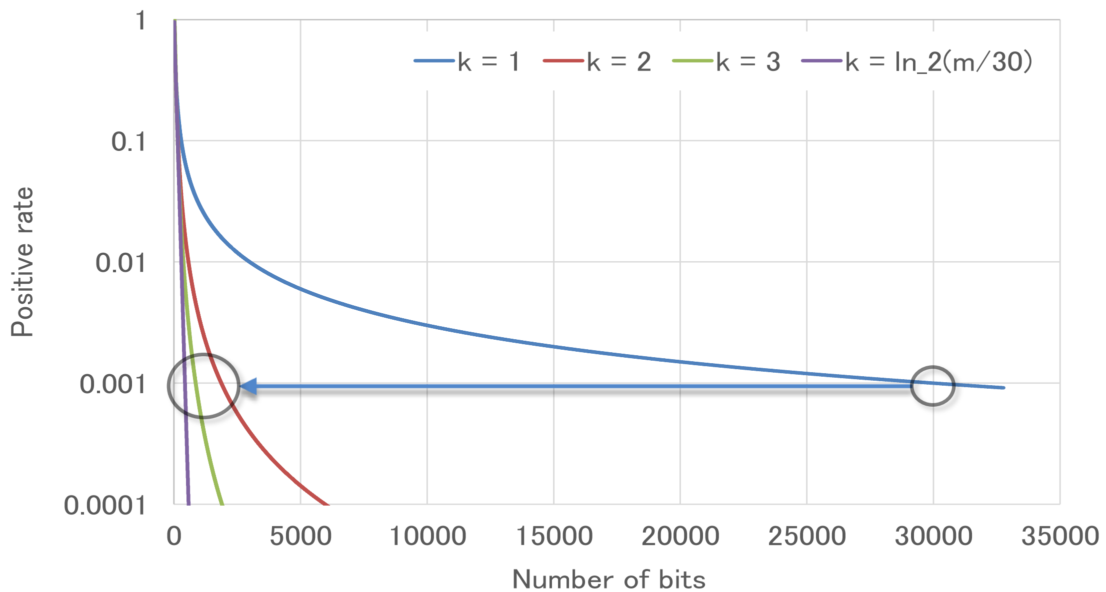

図 4.3: ブルーム・フィルタの解析解

(2) 偽陽性について
登録される 2 箇所をセットするキー値は［124］［124］の順列組み合わせから，9 通り．これから真陽性を除いた，7 通りとなる．
(3) 陰性について
全ての場合の数から，(1)，(2) の場合の数を除いた，55 通りである．

先ほどの，HF の例と偽陽性の数について比較してみると，BF のほうが僅かに小さくなっていることが分かる．このようにハッシュ関数の数が増えることで，BF では偽陽性率を下げることができる．

4.3　ブルーム・フィルタの解析解

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，

$$P = \left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^{k} \tag{4.1}$$

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 が劇的に減少することが分かるであろう．実用上は，P 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 4.3** に，エントリ数 m に対する陽性率 P 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，$P=(1/2)^k \approx 0.6185^{m/n}$ となる．すなわち，P 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 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 [5, 11].

4.4 BF の問題とその解決策

BF では複数のハッシュ関数を用いることで偽陽性発生率を低く抑えることが出来るが，登録した要素を削除することは出来ないことに問題がある．x 節で説明したように，ハッシュ値が偶然の衝突を引き起こしたときに暗黙的な上書きを許しているからである．例えば，図 4.2(C) で要素 14 を削除すれば，要素 12 の上位

の桁の情報も一緒に削除されてしまう．このように，要素が削除できない問題は，BF の各エントリをカウンタにすることで解決される．

また，BF をハードウェアで構成すると，複数のハッシュ関数で単一の BF にアクセスすることになる．これは各エントリに読み書きを行うポートの増加を意味する．一般的に RAM の面積はそのセルに読み書きを行うポート数の 2 乗に比例した面積となる．そのため BF ではハッシュ関数の数を増やせば増やすほど面積が大きくなる．この問題は PCBF と呼ばれる BF をサブ・アレイに分割することで解決される．

第5章　Bloom-like SVW

SVW では，フィルタに読み書きを行う要素がシーケンス・ナンバであるため，複数のハッシュ関数を適用する方法は知られていなかった．そのため，SVW では，3 章で紹介したフィルタと比べて，フィルタの容量あたりの偽陽性発生率が高いことが示されている [2]．本論文では，BF をより一般的に拡張し，SVW でも複数のハッシュ関数を適用し偽陽性発生率を抑える手法として Bloom-like SVW を提案する．本章では，まずブルーム・フィルタの原理とその一般化について説明を行い，次に Bloom-like SVW について説明を行う．

5.1　ブルーム・フィルタの原理とその一般化

BF は複数のハッシュ関数を用いてフィルタのエントリにビットを登録/参照することで，ある要素がある集合に含まれるかどうかを判定するフィルタである．フィルタであるので，この判定には偽陰性はないが，偽陽性はある．

いま BF を，要素となる，異なるハッシュ関数を用いる複数のハッシュ表の集合だと考えると，その要素の集合への帰属判定は次のようになる．
(A) 要素となる，異なるハッシュ関数を用いるハッシュ表の出力が，少なくとも 1 つでも陰性ならば全体で陰性を返す．
(B) 要素となる，異なるハッシュ関数を用いるハッシュ表の出力が，全て陽性ならば全体で陽性を返す．
要素であるハッシュ表もまたフィルタであるので，偽陰性はないが偽陽性はある．BF はハッシュ表全体がフィルタであると同時に，要素であるハッシュ表もまたフィルタとなっている．そのため，(A) 要素となるハッシュ表に偽陰性がないため，少なくとも 1 つでもハッシュ表の出力が陰性ならば，その他のフィルタでの検出は偽陽性であることが原理的に分かる．また (B) より，フィルタ全体としての陽性確率は，個々のハッシュ表の陽性確率の積で表わせ，ハッシュ表の数が増えるだけで陽性率は劇的に下がる．

これは何もハッシュ表にかぎらず，一般のフィルタに拡張が可能である．それは一般のフィルタ自体が，偽陰性はないが，偽陽性はあるような判定を行うからである．

そのため BF をより一般的に考えると (A)(B) の記述は次のように書き換えられる．
(A') 要素となる，異なるハッシュ関数を用いる複数の一般のフィルタの出力が，1つでも陰性ならば全体で陰性を返す.
(B') 要素となる，異なるハッシュ関数を用いる複数の一般のフィルタの出力が，全て陽性ならば全体で陽性を返す.

これは要素となる異なるハッシュ関数を用いる一般のフィルタからなる全体のフィルタは，要素となる一般のフィルタが増えれば増えるほど偽陽性率を劇的に下げることができることを示している．

5.2 Bloom-like SVW

SVW がシーケンス・ナンバの読み書きを行うテーブル (SSNT) はビット表ではないので，複数のハッシュ関数を用いることは出来ないとされていた [2, 12]．提案手法では，SVW の読み書きが行われるテーブルをフィルタとして考えるのではなく，SVW 自体を一般のフィルタと考える．そして，SVW 自体を要素フィルタとし，異なるハッシュ関数を用いる複数の SVW を Bloom-like に用いることで，全体のフィルタを構成する．この全体としてのフィルタ Bloom-like SVW は
(A') 要素となる，異なるハッシュ関数を用いる SVW の出力が少なくとも 1 つでも陰性ならば全体として陰性を返す.
(B') 要素となる，異なるハッシュ関数を用いる SVW の出力が全て陽性ならば全体として陽性を返す.
のような動作をする．また個々の要素となるフィルタの登録/参照手順は，3.1 節で紹介した SVW と同じである．つまり，
(1) ストア命令がコミット時に，フィルタの該当エントリへ自身の SSN を登録する．
(2) ロード命令が実行時に，フィルタに書き込まれている最大の要素を参照する．
(3) ロード命令がコミット時に，フィルタの該当エントリから要素を読込む.
の 3 つの手順である．SVW で検出されたロード命令について，実際にメモリ・ア

クセス順序違反を起こしたかどうかの確認検査も，SVW と同じロード再実行にて行われる．

5.2.1 Bloom-like SVW の順序違反検出

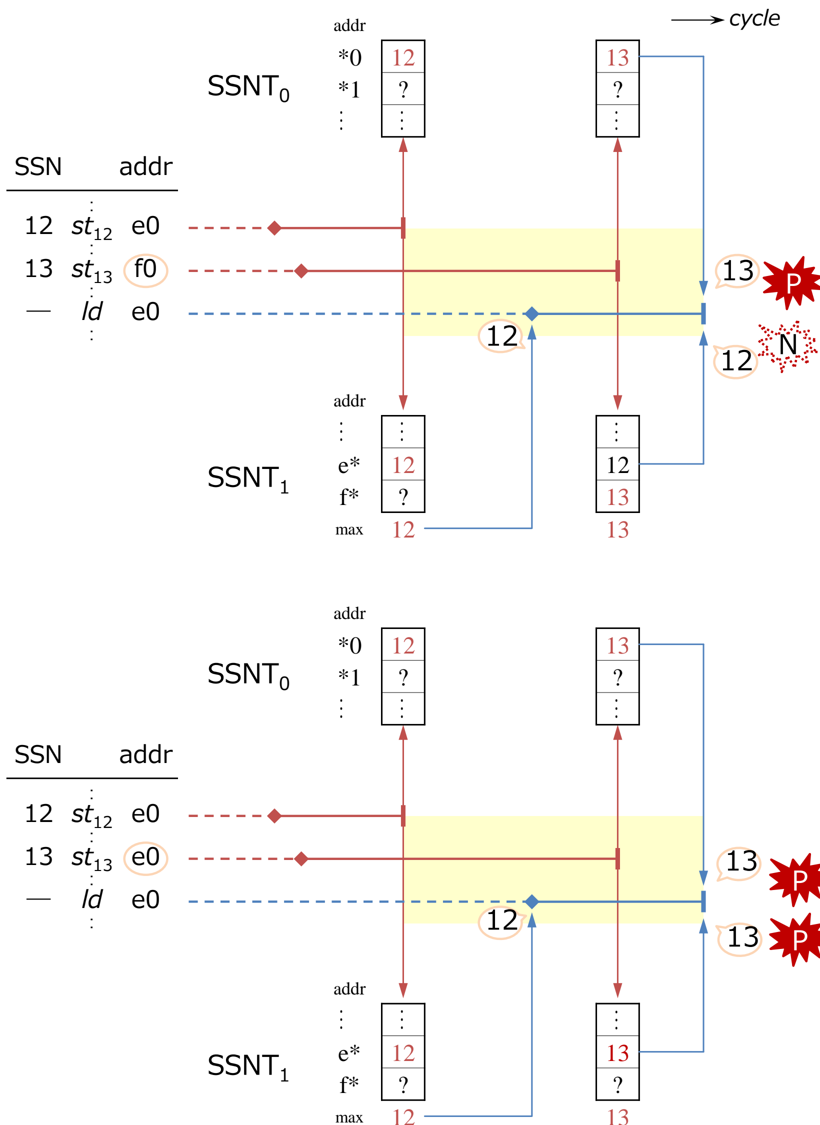

図 5.1: Bloom-like SVW の順序違反検出:順序違反がない場合（上）とある場合（下）

図 5.1 は，提案手法における順序違反検出例である．この図では，3.1.1 節の図 3.1 と概ね同じ構成となっている．図 3.1 と異なる点は，図 3.1 ではアドレスをハッシュ

としていた SSNT が，図 5.1 では，アドレスの右側の桁をハッシュとする $SSNT_0$ とアドレスの左側の桁をハッシュとする $SSNT_1$ の 2 つある点である．図 5.1 の上図は，順序違反が生じていない例である．このとき，アドレスの左側の桁では，st_{13} と ld において，右側の桁が等しいことによるハッシュ値の衝突が起きている．これにより，ロードが実行時とコミット時に $SSNT_0$ から読み込む値には，大小関係が生じる．そのため，実際には順序違反が起きていないのに，$SSNT_0$ では陽性を，つまり偽陽性を返す．一方で，アドレスの左側の桁は異なっているため，ロードが実行時とコミット時に $SSNT_1$ から読み込む値は等しくなる．そのため，$SSNT_1$ は陰性を返す．このとき，Bloom-like SVW は，$SSNT_1$ で陰性であるため，全体として陰性を返す．

図 5.1 の下図は，st_{13} と ld のアドレスが一致しているため，順序違反を起こしている例である．このとき，アドレスの右側の桁もアドレスの左側の桁も一致しているため，st_{13} が実行時に書き込むエントリと，ld がコミット時に読み込むエントリは，$SSNT_0$，$SSNT_1$ でともに一致する．そのため，$SSNT_0$ と $SSNT_1$ の両方で陽性を示すため，Bloom-like SVW は全体として陽性を返す．

5.2.2 Bloom-like SVW におけるフォワーディングの処理

個々のフィルタは 3.1.2 節と同様の処理を行う．全体としてのフィルタの動作は，5.1 節で説明したのとほぼ同様である．

第6章　評価

本章では，まず提案手法について，フィルタの容量を変化させてフィルタとしての性能を評価する．続いて，提案手法に投機フォワーディングを適用した評価を行う．

6.1　評価環境

ベンチマーク　ベンチマークは SPEC CPU 2006 [13] 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 [14] を用いた．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．

プロセッサの構成　ベースラインとなるプロセッサの構成を**表 6.1** に示す．表 6.1 の各パラメタは，IBM POWER7 [1] や Intel Haswell [3] など，最近のハイエンド・プロセッサを参考にしている．また，提案手法におけるロード再実行は，2 サイクルのバックエンド・ストールにより実現可能であるとして理想化した．また SVW においてフィルタに用いるシーケンス・ナンバは 16 ビットであるとし，シーケンス・ナンバのオーバーフロー時の処理についても理想化し，考慮に入れていない．

6.2　フィルタの容量に対する性能評価

本節では，フィルタの容量に対する相対 IPC と偽陽性率の評価を行った．ベースラインは，CAM を用いて順序違反検出を行うモデルで，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．そして，ベースラインと評価モデルに適用したメモリ依存予測器は Store Set 依存予測器である．また，変化させたパラメタは，フィルタのエントリ数 (正確にはインデックスの大きさ) と，フィルタの数 (ハッシュ関数の数) を変化させた．

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**図 6.1** に示す．同上図の偽陽性のグラフでは，左下にプロットされている曲線ほど，同下図の相対 IPC のグラフでは左上にプロットされている曲線ほど，性能が良い．同図中の 5 つの系統は，それぞれフィルタの数 k = 1,2,···,5 に相当する．このグラフでは，フィルタの容量が増えることで，偽陽性率が下がることで相対 IPC が向上していく様子が分かる．また，フィルタの容量が一定の時には，フィルタの数を増やすことが偽陽性率を低下させることが分かる．例えば，偽陽性率を 2% 程度に抑える (このときの相対 IPC の低下率は 2% 程度) には，k = 1 では，20,000 ビット以上の容量が必要なのに対し，k = 2 では，8,000 ビット程度となる．このように k = 1 と比べて，k ≧ 2 と僅かにハッシュ関数の数を増やすだけで，偽陽性率は劇的に現象し，相対 IPC の低下も低く抑えられている．また 5,000 ビット程度のフィルタを用意すれば，1% 以内の IPC 低下に留めることができる．

6.3 ベンチマーク毎の偽陽性率と相対 IPC

図 6.2 は，図 6.1 において，相対 IPC99% を実現するフィルタのパラメタでの，各ベンチマーク毎の偽陽性率 と相対 IPC を表したグラフである．このときの SVW のパラメタは，k = 4 かつ，フィルタの総容量は，5,120 ビットである．

表 6.1: プロセッサの構成

Parameter	Value
ISA	Alpha 21264A w/ byte-word ext.
fetch/issue/cmt	8/8/8 inst./cycle
inst window	64 entries unified
ROB	192 entries
LQ/SQ	72/42 entries
branch pred	16KB:g-share/8K:local hybrid
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
Store Set	
SSID Table	4K entries/1way
LFST Table	512 entries/ 1way

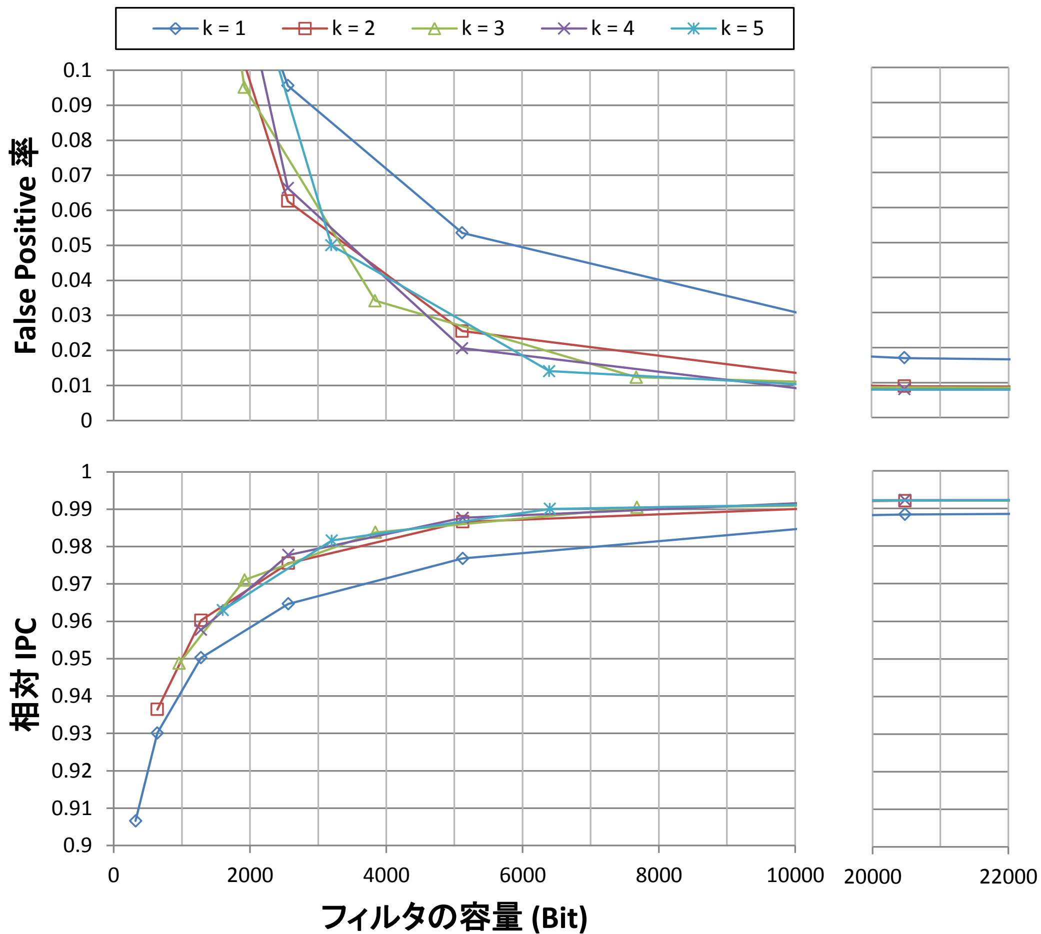

図 6.1: フィルタの容量に対する FalsePositive 発生率 (上) と相対 IPC(下)

図 6.2 をみて，偽陽性率がベンチマークごとにばらついていることが分かる．特に，zeusmp では，8% の偽陽性率によって相対 IPC が 6% と大きく低下している．また，astar に於いては，相対 IPC が 1 を上回る性能を示してるが，これは 6.5 節で後述する LQ-CAM による偽陽性がベースラインのモデルで発生しているのが原因である．

続いて，図 6.2 において，高い偽陽性率を示したいくつかのベンチマークを抽出したグラフが**図 6.3** である．このグラフでは，フィルタの数，つまりハッシュ関数の数を k = 4 と固定した時の，偽陽性率の変化を表している．このグラフでは，フィルタの総容量が増えるに従い，理想的には偽陽性率は 0 に収束するはずである．先ほど高い偽陽性率を示した zeusmp もフィルタの容量が上がることで，偽陽性率が 0 に近づいていると言える．一方で，soplex などの一部ベンチマークは，

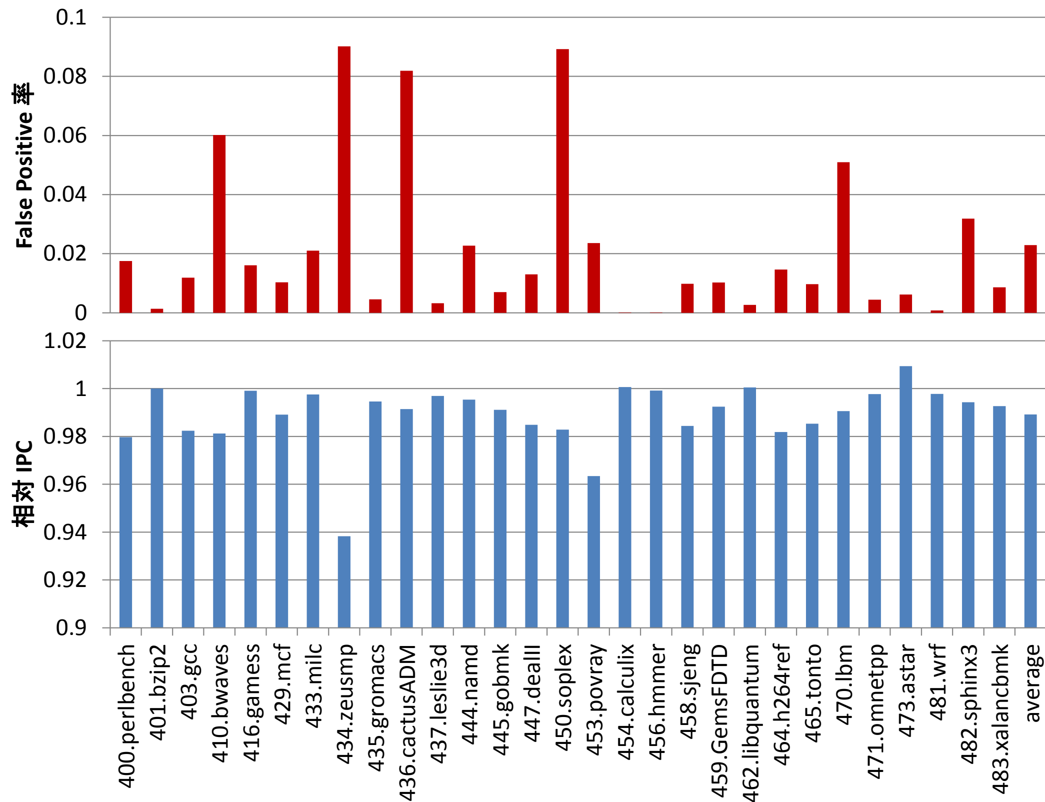

図 6.2: ベンチマーク毎の偽陽性発生率 (上) と相対 IPC(下)

偽陽性率が 10% と高い位置に収束しているように見える．他のベンチマークにおいても，0 ではない値に偽陽性率が収束しているものがある．このことから，原理的に避け得ない偽陽性の存在が示唆される．この事については，6.5 で詳しく述べる．

6.4 提案手法への投機フォワーディング適用時の評価

6.4 では，提案手法に投機フォワーディングを適用したときの効果を評価する．また，本評価での投機フォワーディングは，SQIP[11] をベースとしており，アドレスが一致しないかぎり投機フォワーディングを行わない．

6.4.1 評価モデル

評価したモデルは以下のとおりである．

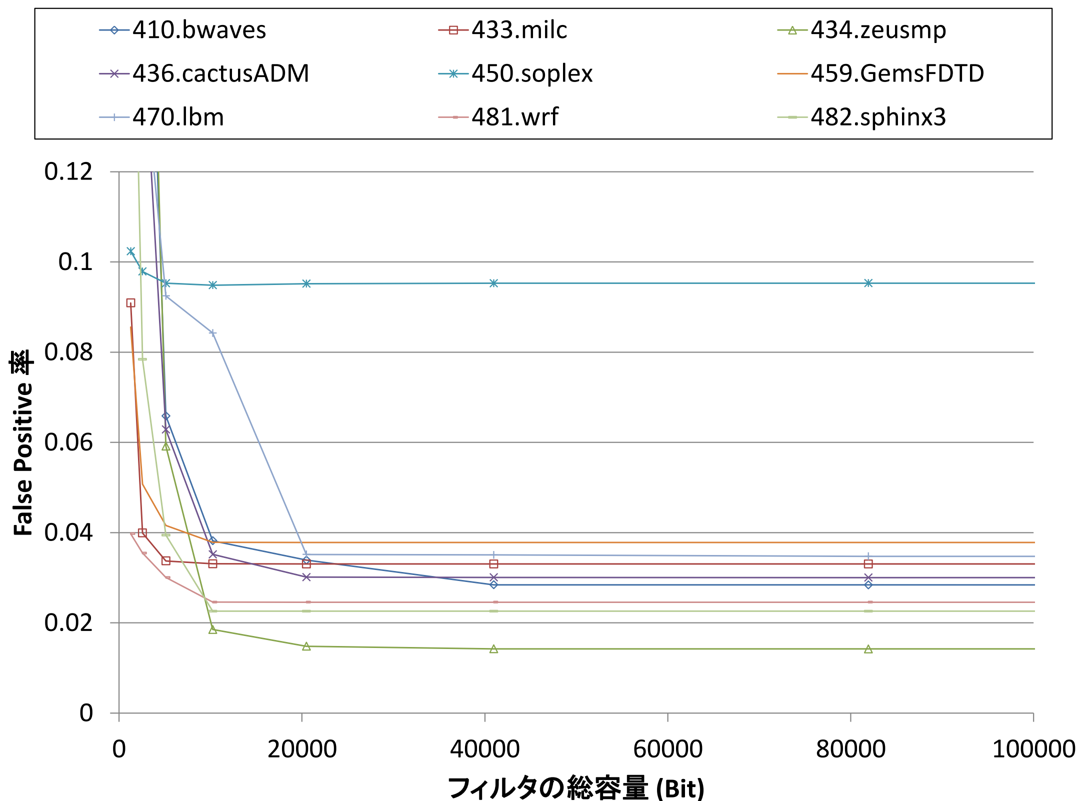

図 6.3: ハッシュ関数の数 : k = 4 における 偽陽性率とフィルタの総容量

- 距離ベースの依存予測器を用いた Bloom-like SVW の投機フォワーディング．但し，予測に用いる分岐履歴長を 0〜6 ビットと変化させ，それぞれに応じた 7 つのモデルを用意した．

- Store Set 依存予測器を用いた Bloom-like SVW の投機フォワーディング

ここで，用いる Bloom-like SVW は，LQ-CAM を用いて順序違反検出を行うモデルと比べて，性能低下が全ベンチマーク平均で 0.5% 程度のごく僅かなパラメタを用いた．但し，距離ベースの依存予測器を用いたモデルでは，予測テーブルの数を分岐履歴毎に持つようにした．例えば，分岐履歴長を 2 ビット用いるならば，予測テーブルの数は 4 つにするといった具合である．また，Store Set 依存予測器を用いたモデルでは，予めパラメタ・チューニングを行い最良な結果を示したものを用いている．

二つの依存予測器の細かなパラメタを**表 6.2** にのせる．

また本手法におけるベースラインは，CAM による実行順序違反検出とフォワーディングを行うモデルであり，このモデルは表 6.1 の構成に従っている．

6.4.2 評価

評価した項目は以下のとおりである．

- ベースラインに対する相対 IPC
- 相対 IPC と予測テーブルの容量の関係
- 依存予測の結果

本節では，それぞれの評価について述べる．

ベースラインに対する相対 IPC

結果を**図 6.4** に示す．同図の凡例において，GHTx が距離ベースの依存予測器のモデルである．このモデルでは，GHTx の x が分岐履歴長にあたる．また SS が Store Set 依存予測器を用いたモデルである．同図横軸は，SPEC CPU 2006 の各々のベンチマークにあたり，縦軸が相対 IPC である．

同図からは，投機フォワーディングの適用による性能 (相対 IPC) 低下は全ベンチマークを平均して 2%〜5% に抑えらている．また，距離ベースの依存予測器では，分岐履歴長を長くすればするほど，相対 IPC が高くなる傾向が有ることがわかる．

また，gcc, zeusmp, namd, soplex において，Store Set を依存予測器として用いるモデルが良い性能を示している．

相対 IPC と予測テーブルの容量の関係

図 6.5 は，図 6.4 における各モデルの平均の相対 IPC と，予測テーブルの容量の関係を表した図である．同図の横軸が予測テーブルの総容量 (Bit)，縦軸が相対 IPC であり，左上にプロットされているものほど性能が高いことを示している．また距

表 6.2: 依存予測器 の構成

Parameter	Value
Store Set	
SSID Table	4K entries/ 8way
LFST Table	512 entries/ 1way
距離ベースの依存予測器	512 entries/ 8way

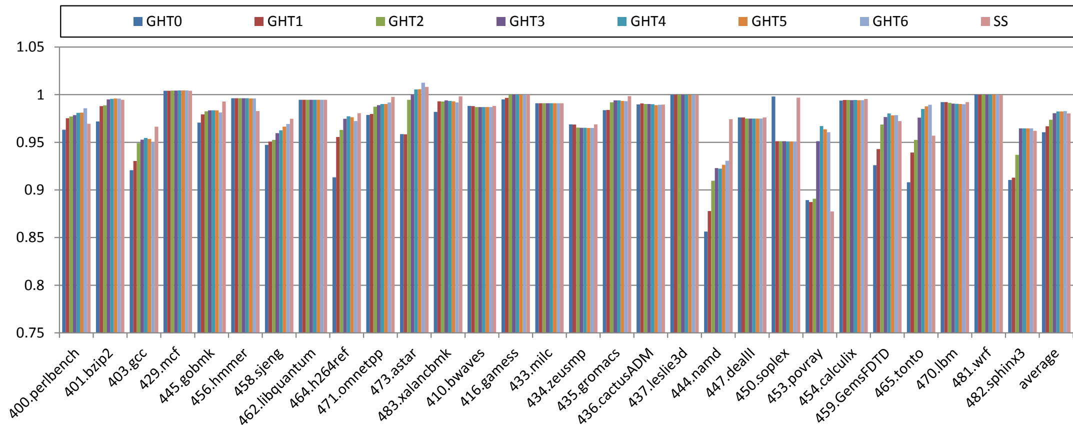

図 6.4: ベースラインに対する相対 IPC

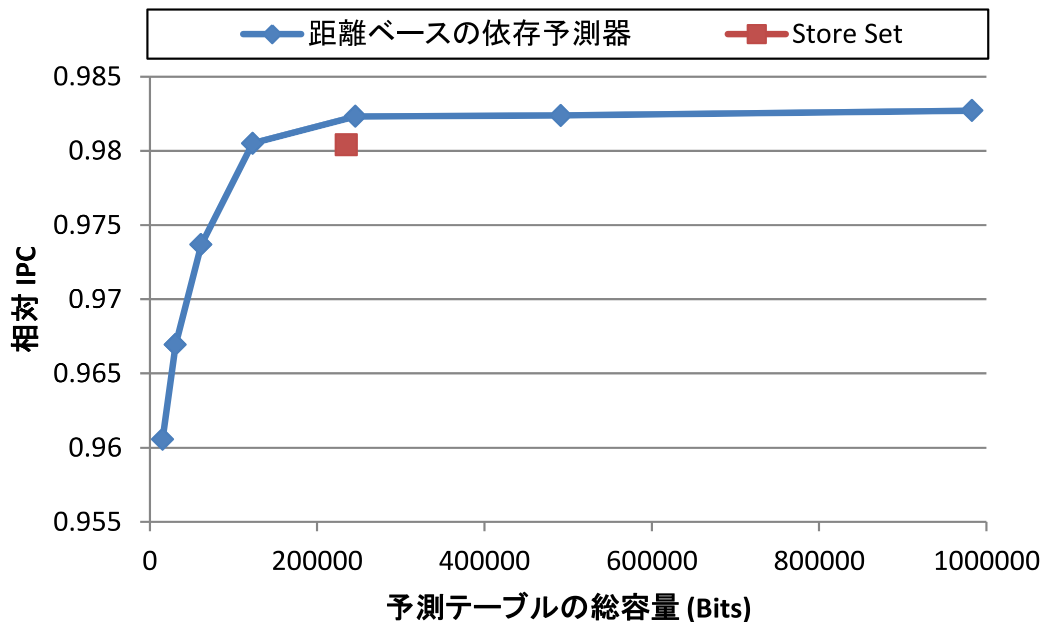

図 6.5: 予測テーブルの容量と相対 IPC

離ベースの依存予測器としてプロットされてる 7 つの点は，左から順に，GHT0, GHT1，…，GHT6 に該当している．

このグラフからは，相対 IPC を 2% 程度低下に保つために必要な予測テーブルの容量は，Store Set では 245,760 Bit であるのに対し，距離ベースの依存予測器では，122,880 Bit である．距離ベースの依存予測器は，Store Set と比べて，半分程度予測テーブルの容量を削減できることを示す結果となった．

依存予測の結果

次に，依存予測の結果を分類して，2% の IPC 低下の要因を探った．SQIP ベースの投機フォワーディングにおいて，依存予測されたロード命令の依存予測の結果は，まず実際に依存があったロード命令と実際に依存はなかったロード命令の 2 つに大別される．またそれぞれの場合における細かな状況を考えると下記のように 8 つに分類される．

実際に依存はあった ：

- **(1)** ■ ：依存ありと予測し，実際に予測した依存元に依存していてフォワーディングを受けた．
- **(2)** ■ ：依存ありと予測し，予測した依存元とアドレスが一致していたのでフォワーディングされた．しかし実際には，予測と異なるストアに依存していたので実行順序違反が起きた．
- **(3)** ■ ：依存ありと予測したが，依存元が既にリタイアしていたため，メモリからデータを得た．しかし，実際には予測と異なるストアに依存があったため，実行順序違反が起きた．
- **(4)** ■ ：依存ありと予測したが，予測した依存元とアドレスが一致しなかったので，メモリからデータを得た．しかし，実際には予測と異なるストアに依存していたので実行順序違反が起きた．
- **(5)** ■ ：依存なしと予測したが，実際には依存があったので実行順序違反が起きた．

実際に依存はなかった ：

- **(6)** ■ ：依存なしと予測し，実際に依存はなかった．
- **(7)** ■ ：依存ありと予測したが，依存元が既にリタイアしていたため，メモリからロードデータを得る．実際には依存がなく，また依存していないストア命令を待つこともなかった．
- **(8)** ■ ：依存ありと予測したが，予測した依存元とアドレスが一致しなかったので，メモリからデータを得た．実際には依存がないストア命令を待ってしまった．

ここでは，左側のシンボルの色は，後に説明する**図 6.6**，**図 6.7** の凡例の色に対応している．

この 8 つの分類のうち，(1)，(6) は，予測と実際の結果が正しく，性能低下を引き起こし得ない．また，(7) に関しても，予測の結果は間違っていたが，実行に遅延を生じること無く正しいデータを得ることができるので，性能低下を引き起こさない．

性能低下を引き起こし得るのは，予測が間違っていて，実行順序違反からの回復処理のロールバックを生じる (2)，(3)，(4)，(5) の 4 つと，予測が間違うことで本来ならば依存しない命令を待ち実行に遅延が生じる (8) の計 5 つである．

図 6.6，図 6.7 は依存予測の結果を表した積立棒グラフで，図 6.6 は INT 系のみを，図 6.7 は FP 系のみを抽出している．このグラフは，軸のプラス側では実際に依存があった (上記分類 (1)-(5) に該当) ロード命令の割合，軸のマイナス側では実際に依存がなかった (上記分類 (6)-(8) に該当) ロード命令の割合を示している．そのため，プロットされている棒グラフの正の値と負の値の絶対値を足せば 1 となる．また同図では，8 本の棒グラフで 1 つのベンチマークの依存予測の結果を表しており，対応するベンチマーク名は横軸の下側に書いてある．またベンチマーク毎の 8 本の棒グラフが表すモデルは，左から順に GHT0，GHT1，···，GHT6，SS に対応する．この評価モデルは，グラフの横軸上部に書いてある．

図 6.6，図 6.7 からは，すべての系列のおいてマイナス側のグラフが長く，ロード命令の大半が依存関係に無いことがわかる．また予測が正しい (1)，(6) がグラフの大部分を占めている．また (1)，(6) のように性能低下を引き起こし得ない (7) も，ベンチマークによってははっきりと棒グラフに現れる程度には多い．

次に性能低下を引き起こしうるベンチマークについて見てみる．実行順序違反を引き起こしうる (2)-(5) については，全ベンチマークで平均しても，ロード命令中で 0.1% 程度のごく僅かなものであった．そのため，(2)-(5) のようなごくわずかしか無いロード命令が，2% 程度の性能低下を引き起こすとは考え難い．

一方で，(8) の実行に遅延が生じるロード命令について見てみると，図 6.6，図 6.7 をみてわかるように，ベンチマークによっては 7% ほどの割合を占めている．

図 6.8 は，性能低下を引き起こしうる要因である，実行に遅延が生じるロード命令 (以下 (8) のロード命令と記す) の割合を図 6.6，図 6.7 から抽出したグラフである．

この図 6.8 と図 6.4 を合わせてみると，相対 IPC が大きく下がるベンチマークほど，(8) のロード命令が多い傾向がある．その一方で，zeusmp などのベンチマー

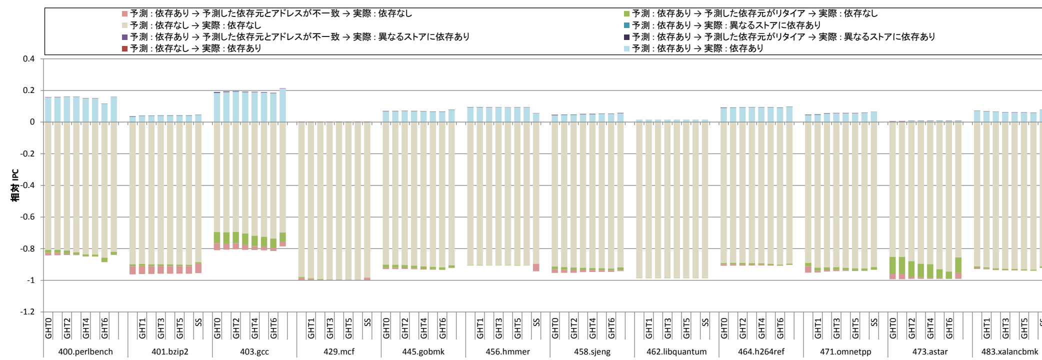

図 6.6: 依存予測の結果 (INT 系)

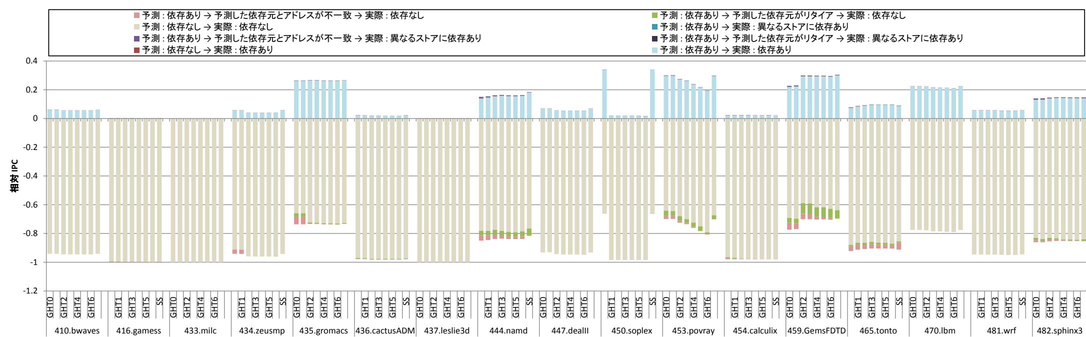

図 6.7: 依存予測の結果 (FP 系)

クでは，履歴長の増加とともに，(8) のロード命令が減少しているのにもかかわらず，相対 IPC は増加していないといった例外も存在している．

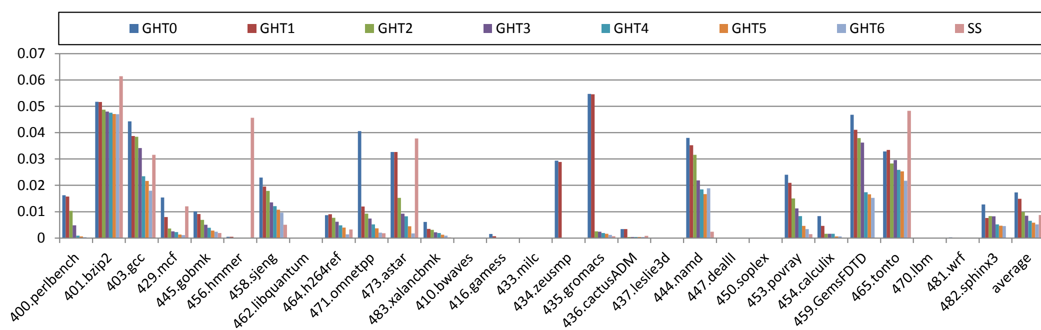

図 6.8: 実際に依存はないが依存があると予測されるロードの割合

6.5 考察

本節では，提案手法で高い IPC を示した原因と，高い偽陽性率を示した原因について考察する．主に，LQ-CAM の偽陽性と，原理的に避けることのできない偽陽性である，サイズの違いによる偽陽性，サイレント・ストアによる偽陽性について述べる．

LQ-CAM による順序違反検出の偽陽性

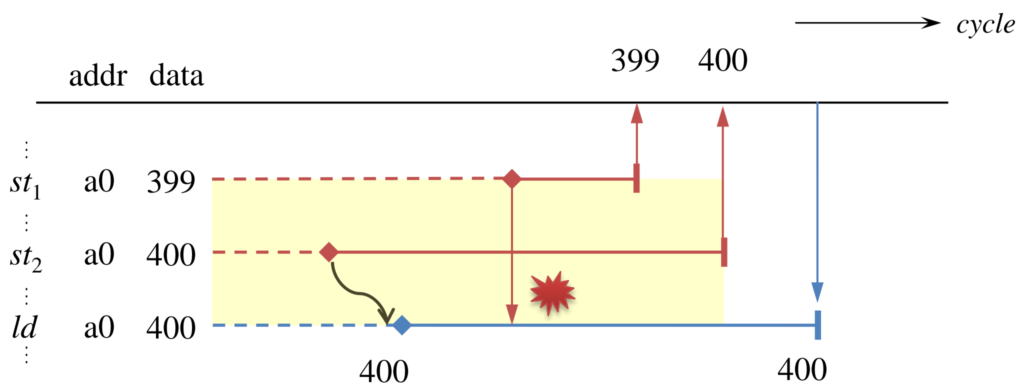

図 6.9: LQ-CAM によるフォワーディング時の偽陽性

図 6.9 は，LQ-CAM によるメモリ・アクセス順序違反検出の偽陽性を表した図である．LQ-CAM はフィルタによる手法とは異なり，ストア命令の実行とロード命令の実行との入れ替わりを検出しさえすれば，実行順序違反の検出が可能である．図 6.9 では，st_1,st_2,ld は，ターゲット・アドレスが等しい．プログラム順で依存関係に有るのは，st_2 と ld であるため，フォワーディングが行われている．しかし，st_1 の実行が ld の実行より遅れてしまうと，フォワーディングされた ld を st_1 が実行順序違反として検出してしまう．このように LQ-CAM では，正しくフォワーディングされたロード命令を，誤って検出してしまう恐れがある．つまり LQ-CAM には偽陽性がある．このフォワーディングに伴う偽陽性は提案手法には存在しない．そのため，図 6.2 のように，ベースラインを上回る IPC がでるベンチマークが存在する．

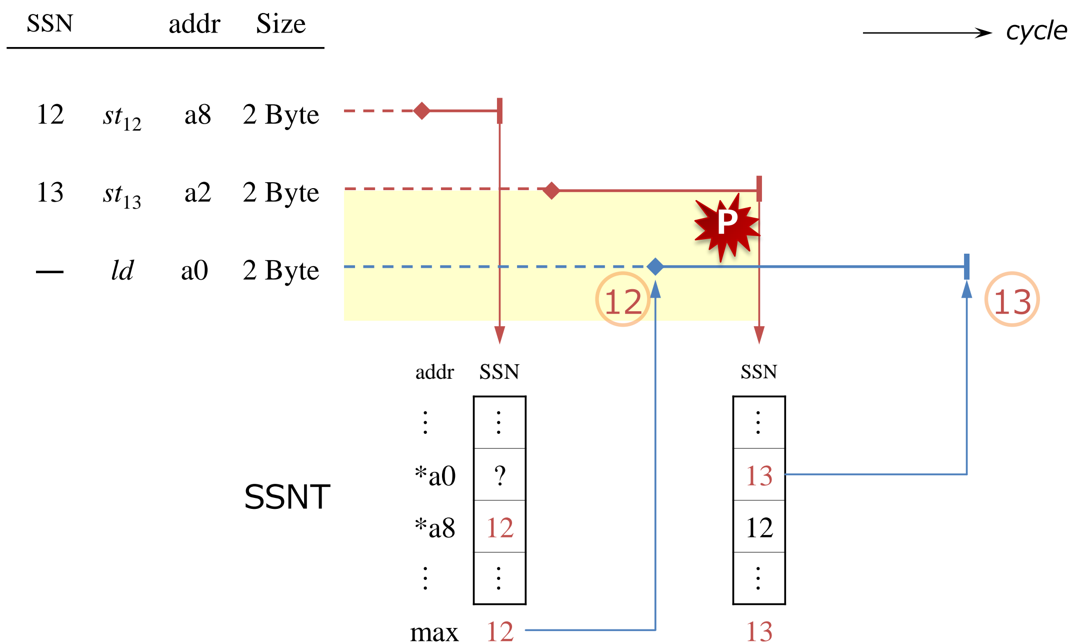

図 6.10: サイズの違いによる偽陽性

サイズの違いによる偽陽性

今回，SVW および提案手法では，テーブルに読み書きを行う際，アドレスの下位 2 ビットを切り捨てていた．そのため，ある異なるアドレスにおいて，下位 2 ビットが異なろうが，上位ビットが等しければ，異なるアドレスにおいて，ハッシュ値の衝突が発生してしまう．**図 6.10** にある全てのロード/ストア命令のアクセス・サイズは 2Byte である．このとき，st_{13} と ld はそれぞれターゲット・アドレスが $a2$ と $a0$ であるため，依存関係にない．しかし，下位 2 ビットを切り捨てる今のモデルでは，アドレス $a0$ と $a2$ はハッシュ値が衝突してしまう．そのため，ハッシュ値は衝突するが依存関係にない st_{13} と ld で，コミットと実行の順序が入れ替われば，提案手法では偽陽性を示す．このように，ロード/ストア命令のアクセス・サイズによっては，原理的に避けることの出来ない偽陽性が存在する．

サイレント・ストアによる偽陽性

図 6.11 では，st_{12}，st_{13} と ld では，ターゲット・アドレスが $a0$ で一致している．そして，st_{12}，st_{13} で，メモリに書き込むデータも 400 と一致している．このとき，

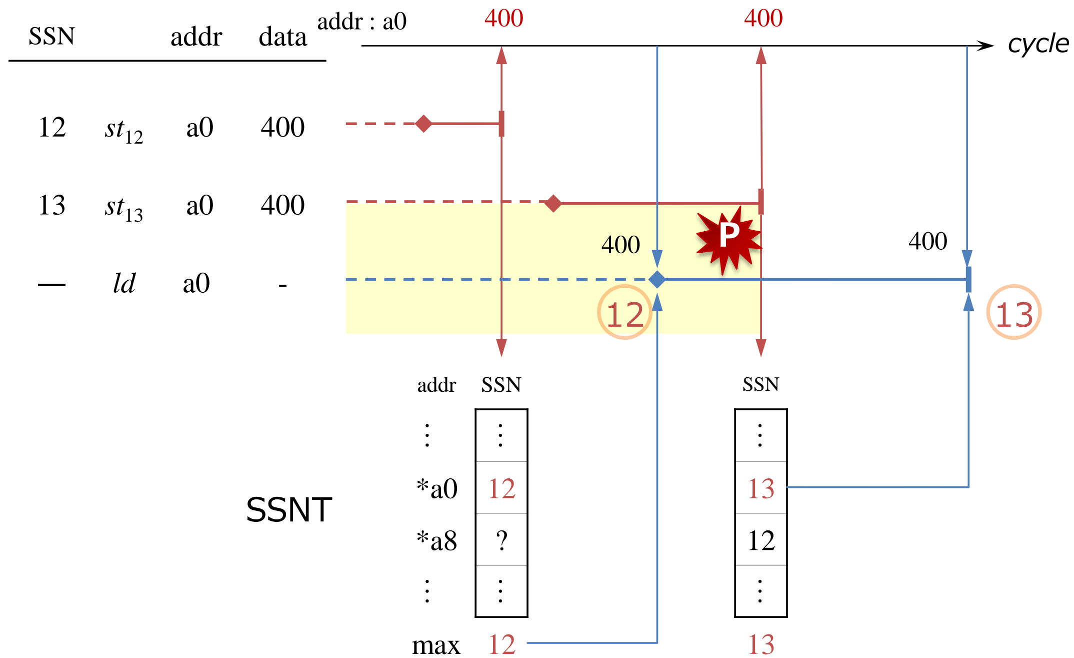

図 6.11: サイレント・ストアによる偽陽性

st_{13} は ld と依存関係にある．図 6.11 では，st_{13} のコミットと ld の実行が入れ替わっているので，実行順序違反として ld が検出される．検出された ld は，偽陽性か否かを確かめるために，ロード再実行を行う．ロード再実行では，ロード命令の実行時と再実行時で読み込んだ値が等しいかどうかで偽陽性を判断する．しかし，st_{13} は書き込まれる前と同じ値を書き込んでしまうので，確かに ld は実行順序違反を引き起こしているが，ld は正しい値を得ているため偽陽性と判定される．このような現象を引き起こす，書き込まれる前と同じ値を書き込んでしまうストア命令はサイレント・ストアと呼ばれている．サイレント・ストアによる偽陽性もまた，原理的に避けることが出来ない．

第7章　おわりに

7.1　本論文のまとめ

本論文では，LQ を簡略化する手法として，Bloom like SVW を提案し，その評価を行った．本手法では，ブルーム・フィルタの概念をより一般的なフィルタへと拡張したことで，一般的なフィルタでも複数のハッシュ関数を適用しさえすれば，小容量でかつ低い偽陽性率のフィルタを構成できることを示した．提案手法では，5000 Bit 程度のフィルタを用意すれば，1% 程度の性能低下に抑えられることが示された．

また，提案手法 SQIP をベースとした投機フォワーディングを適用した評価を行った．この評価では，投機フォワーディングの適用による提案手法の IPC 低下は，2% 程度と小さく抑えられること，また 2% 程度の性能低下を目標とするには，距離ベースの依存予測器は Store Set の半分程度の予測テーブルの容量で十分であることが示された．

また，依存予測の結果について細かく分析した結果，IPC 低下の要因は，実行順序違反による影響よりも，依存予測が間違うことで実行に遅延が生じてしまうロード命令の影響の方が大きいことが示唆された．

7.2　今後の課題

本論文では，3 章で紹介した手法の実装が間に合わず，提案手法の評価しか行えなかった．今後は，フィルタによる実行順序違反を行う手法として，PCBF を用いた手法，DMDC を用いた手法のそれぞれに投機フォワーディングを用いた評価を行いたい．

また，本論文における評価では，フィルタの容量に対する IPC の評価は行ったが，面積評価については行っていない．今後は，提案手法，および 3 章で紹介した手法を含めて，投機的なロード/ストア命令の実行を行う手法の全体としての面

積評価を行いたい.

そして，6.5 節で，言及した原理的に避けることのできない偽陽性について，どれほど IPC 低下に影響を与えているのかの調査も合わせて行いたい.

関連図書

[1] Sinharoy, B., Van Norstrand, J., Eickemeyer, R., Le, H., Leenstra, J., Nguyen, D., Konigsburg, B., Ward, K., Brown, M., Moreira, J., Levitan, D., Tung, S., Hrusecky, D., Bishop, J., Gschwind, M., Boersma, M., Kroener, M., Kaltenbach, M., Karkhanis, T. and Fernsler, K.: IBM POWER8 processor core microarchitecture, *IBM Journal of Research and Development*, Vol. 59, No. 3, pp. 2:1–2:21 (2015).

[2] Kurata, N., Shioya, R., Goshima, M. and Sakai, S.: Address Order Violation Detection with Parallel Counting Bloom Filters, *IEICE Trans. on Information and Systems* (2015).

[3] Hammarlund, P., Martinez, A. J., Bajwa, A. A., Hill, D. L., Jiang, E. H. H., Dixon, M., Derr, M., Hunsaker, M., Kumar, R., Osborne, R. B., Rajwar, R., Singhal, R., ReynoldD ’Sa, Chappell, R., Kaushik, S., Chennupaty, S., Jourdan, S., Gunther, S., Piazza, T., Burton, T.: Haswell: The Fourth-Generation Intel Core Processor, *Micro, IEEE*, Vol. 34 (2014).

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

[5] Roth, A.: Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA ’05, Washington, DC, USA, IEEE Computer Society, pp. 458–468 (2005).

[6] Castro, F., Pinuel, L., Chaver, D., Prieto, M., Huang, M. and Tirado, F.: DMDC: Delayed Memory Dependence Checking Through Age-Based Filtering, *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitec-*

ture, MICRO 39, Washington, DC, USA, IEEE Computer Society, pp. 297–308 (2006).

[7] Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors, *Commun. ACM*, Vol. 13, No. 7, pp. 422–426 (1970).

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

[9] Yoaz, A., Erez, M., Ronen, R. and Jourdan, S.: Speculation Techniques for Improving Load Related Instruction Scheduling, *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, Washington, DC, USA, IEEE Computer Society, pp. 42–53 (1999).

[10] Sha, T., Martin, M. M. K. and Roth, A.: NoSQ: Store-Load Communication Without a Store Queue, *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, Washington, DC, USA, IEEE Computer Society, pp. 285–296 (2006).

[11] Sha, T., Martin, M. M. K. and Roth, A.: Scalable Store-Load Forwarding via Store Queue Index Prediction, *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, Washington, DC, USA, IEEE Computer Society, pp. 159–170 (2005).

[12] 倉田成己: 高効率なメモリ順序違反検出機構に関する研究, 博士論文, 東京大学大学院情報理工学研究科 (2014).

[13] The Standard Performance Evaluation Corporation: SPEC CPU2006 suite.

[14] 塩谷亮太, 五島正裕, 坂井修一: プロセッサ・シミュレータ「鬼斬弐」の設計と実装, 先進的計算基盤システムシンポジウム SACSIS, pp. 120–121 (2009).

研究業績

口頭発表（査読なし）

1. 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 2013-ARC-206, No. 4, pp. 1-7 (2013).
2. 西川 卓, 塩谷 亮太, 入江 英嗣，五島 正裕, 坂井 修一: メモリ・アクセス順序違反検出手法の評価, 情報処理学会研究報告 2015-ARC-216 , No. 4, pp. 1-9 (2015).
3. 西川 卓，塩谷 亮太, 入江 英嗣，五島 正裕, 坂井 修一: Bloom-like SVW の評価, 情報処理学会研究報告 2015-ARC-217 , No. 9, pp. 1-8 (2015).
4. 西川 卓，塩谷 亮太, 入江 英嗣，五島 正裕, 坂井 修一: フィルタを用いたメモリ・アクセス順序違反検出手法の評価情報処理学会研究報告 2015-ARC-219 (発表予定)

謝辞

本研究を進めるにあたり，本当にたくさんの方々にお世話になりました．指導教員である，坂井修一教授には，相談会等でさまざまなご指摘や叱咤激励をいただきました．本年度から配属の入江英嗣准教授には，コンピュータ・アーキテクチャに関する議論の相手をして頂きました．論文共著者である名古屋大学大学院工学研究科の塩谷亮太助教には，研究のアイディアやシミュレータの使い方などでアドバイスを頂きました．同じく共著者である国立情報学研究所の五島正裕教授には研究指導，物の考え方そして事務処理に至るまで様々なご助言・ご助力を頂きました．五島先生のおかげで今の私がある，といっても過言ではありません．

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

その他にも，気軽に話せた同期の方々のおかげで，楽しい研究室生活をおくることが出来ました．その他，多くの研究室メンバーにも様々な面で助力をいただくことが出来ました．みなさま本当にありがとうございました．