Vol.2015-ARC-217 No.9
2015/10/8

Bloom-like SVW の評価

西川 卓[1]　塩谷 亮太[2]　入江 英嗣[1]　五島 正裕[3]　坂井 修一[1]

概要：ロード・ストア・キュー (LSQ) の CAM を排除するため，RAM で構成されたハッシュ・フィルタを用いてメモリ・アクセス順序違反を検出する手法が提案されてきた．しかし，我々の以前の提案であるパラレル・カウンティング・ブルーム・フィルタを用いた手法を除いて，複数のハッシュ関数を利用する手法はこれまでに知られていなかった．そこで，既存手法の 1 つである Store Vulnerability Window について，複数のハッシュ関数を利用する手法である Bloom-like SVW を提案している．本稿では，近年のハイエンド・プロセッサをモデルとして，より詳細な評価を行い，複数のハッシュ関数を利用することで低い偽陽性率を実現できることを確かめた．

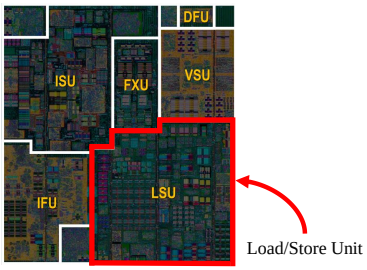

図 1　POWER8 のチップ写真 [1]

1. はじめに

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** を用いて LSQ を構成することによって行われて来た．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], [2]．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** が，IBM POWER8 プロセッサのチップ写真であるが [1]，この図から見て分かるように 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 2** は Intell Haswell プロセッサにおける L1D の面積と LSQ の CAM の面積を CACTI [3] によって算出したものである．このグラフから見て分かるように，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分かる．

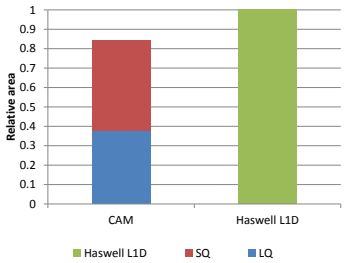

図 2　L1D と LSQ の CAM の面積 [1]

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

1　東京大学大学院情報理工学系研究科
2　名古屋大学大学院工学研究科
3　国立情報学研究所

つか提案されている [4], [5], [6], [7]．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，偽陽性が低い BF を用いる，**ブルーム・フィルタ (BF)**[8] を用いた手法 [7] と，SVW [5] を Bloom-like に用いる手法を提案している．

投機フォワ-ディング

SQ CAM の排除は，投機的にフォワーディングを行うことで，ロード命令による依存元ストアの検索を無くす手法がいくつか提案されている．この投機フォワーディングの性能は，ロード命令とストア命令の依存関係を予測する依存予測器の精度の高さによって決まる．この依存予測器には，[9] や [10] で提案されている，ロード命令とストア命令の動的な命令間距離を学習するものがある．

本稿では，ロード命令とストア命令の動的な命令間距離を学習する依存予測器を実装した Bloom-like SVW のための評価を行う．

本稿の構成

本稿の構成は以下のようになる．まず 2 章 でメモリアクセス順序違反を SVW を用いて説明する．続く 3 章 で，提案の要となる BF が低い偽陽性を実現するメカニズムを説明する．そして?? 章 で提案手法である Bloom-like SVW を説明する．5 章 で簡単な IPC の評価を行い，6 章で本稿をまとめる．

2. SVW による メモリ順序違反検出

本章では，メモリ・アクセス順序違反とフォワーディング・ミス，およびそれぞれの検出について，SVW を用いて説明する．

2.1 メモリ命令の実行とコミット

out-of-order 実行において，命令の実行は out-of-order に行われるが，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事である．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

実行時にアドレス計算と，L1D に値を読み込みにいく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計算のみを行い，コミット時に L1D に書込み，つまりステートの更新を行う．

ここでは，ロード命令とストア命令においてメモリアクセスのタイミングが違うことに注意をしていただきたい．

メモリ・アクセス順序違反とは，依存関係に有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．

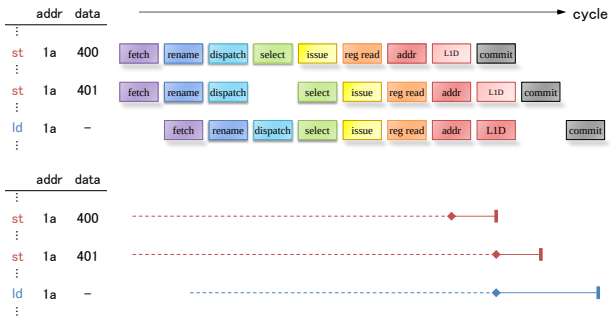

図 3 一般的なパイプライン・チャート (上) と
本稿で用いるパイプライン・チャート (下)

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ので，本稿では**図 3** 上図の一般的なパイプライン・チャートではなく，図 3 下図のようなパイプライン・チャートを用いる．このチャートでは，実行ステージを菱型，コミットステージは縦棒，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表している．

2.2 Store Vulnerability Window によるメモリ・アクセス順序違反検出

Store Vulnerability Window(SVW) とは，Roth らが提案しているフィルタによる LQ の CAM の排除手法である [5]．この手法では，フィルタにシーケンス・ナンバを登録・参照することで，メモリ・アクセス順序違反違反を検出する手法である．本稿では，ストアに動的に割り当てられるシーケンス・ナンバを StoreSequence Number(SSN)，SSN を登録するフィルタを SSN Table(SSNT) と呼ぶことにする．

図 4 を用いて，メモリアクセス順序違反，及び SVW におけるメモリ順序違反検出を説明する．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 st12，st13，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 4 上図は，ld が st13 の書き込むデータ 401 を読み込むメモリ順序違反が生じていない例である．SVW では，ストア命令は実行時にメモリにストア・データを書き込むと同時に，SSNT の該当エントリに，自身の SSN を書き込む．それに対して，ロード命令は実行時には SSNT の最大値を，コミット時には SSNT の該当エントリに格納され

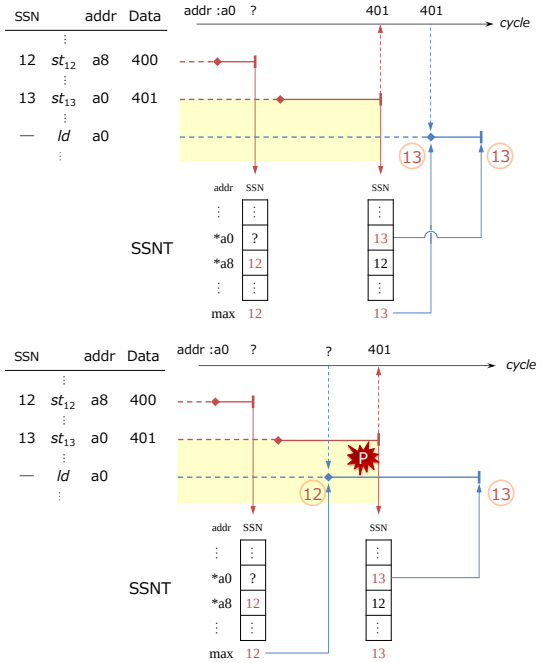

図 4 SVW の順序違反検出：
順序違反がない場合（上）とある場合（下）

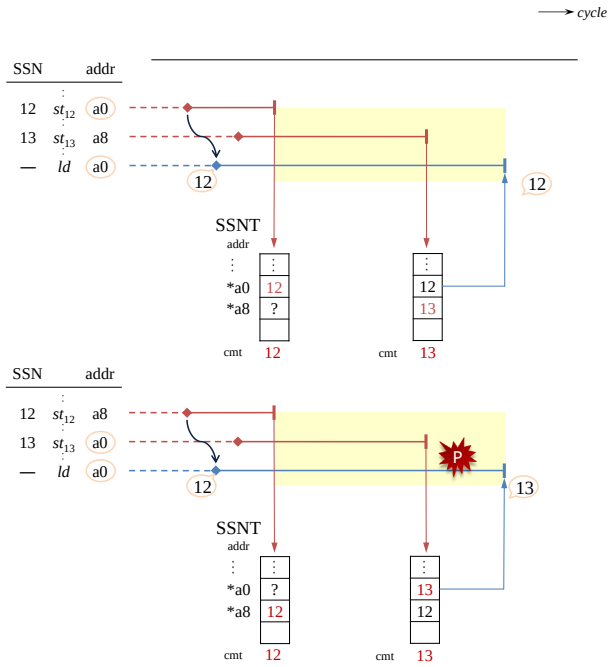

図 5 svw におけるフォワーディング・ミス 検出
ミスがない場合 (上) ミスがある場合 (下)

ている SSNT を読み込む．図 4 上図では，st12 はコミット時に SSNT の該当エントリである*a0 に 12 を，st13 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*a8 から 13 を読み込む．この時，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，st13 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 4 の下図は，ld の実行が早まりデータ 401 を読み込めない，つまりメモリ順序違反が生じている例である．ここでは，ld が実行時に読み込む SSNT の最大値が 12 となるところに上図との差異がある．これにより，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 ではメモリ順序違反を検出する．

2.3 SVW による フォワーディング・ミス検出

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．投機フォワーディングは，ロード命令とストア命令の依存関係を予測し，フォワーディングする手法である．この予測は完璧には出来ないため，予測ミス，つまりフォワーディング・ミスを検出する必要がある．

しかし，SVW では，フォワーディング・ミスは順序違反検出と同じようには検出できない．フォワーディングはストア命令の実行時に行われるため，ロード命令の実行がフォワーディングを行うストア命令のコミットよりも先行してしまうからだ．これは，フォワーディングされたロード命令の全てが，メモリ順序違反として検出されることを意味する．

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 5 を用いて，フォワーディングとフォワーディング・ミス，及び SVW におけるフォワーディング・ミス検出を説明する．ここでは，上図においては st12 と ld，下図においては st13 と ld が依存関係にある命令である．両図共に，依存予測に従って，st12 が ld にフォワーディングを行っているが，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．

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 st12 から 12 を読み込んでいる．上図ではコミット時に 12 を読み込むことで，フォワーディングを行った st12 と ld において，アドレスが一致していることを保証する．下図ではコミット時に 13 を読み込むことで，st12 と ld のアドレスが一致していないことを判別し，これをフォワ-ディング・ミスとして検出する．

2.4 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有る．

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非常に大きな問題であったが，後に述べるようにこれは我々の提案により緩和される．

アクセス手順の問題

アクセス手順の問題は，ストアセット [11] のような依存関係に有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．こうした予測器では過去に起きた，メモリ・アクセス順序違反違反をもとに学習を行うが，違反の検出時には順序違反されたストア命令がコミットされているため，依存関係に有るメモリ命令の特定ができないからだ．そのため，SVW では，メモリ依存予測のために，コミットしたストア命令を特定するためのロジック (SPCT) が別途必要である．

3. 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 [8]．BF は k 個のハッシュ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，包含判定を行う．この判定には，偽陽性はあるが，偽陰性はない，

またハッシュ関数を複数使うことで，1 つのハッシュのみを用いるハッシュ・フィルタ (HF) と比較して，偽陽性発生率が劇的に低い．

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．

3.1 ハッシュ・フィルタの動作例

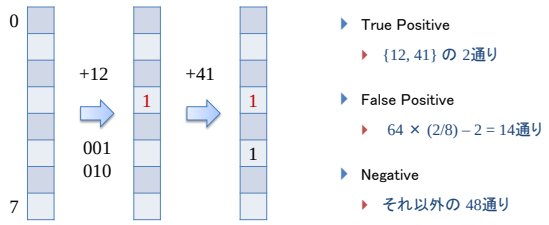

図 6 ハッシュ・フィルタ の例

本節では，ハッシュ・フィルタ (HF) の動作例について，**図 6** を用いて説明を行う．図 6 の HF において，キー値は八進数で 00-77 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 k = 1 種とし，そしてエントリ数 m は m=8 とした．このとき，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．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．要素 41 に関してはそのハッシュ値は，101 となり，該当エントリはセットされていないので，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．これは要素 12 とハッシュが衝突を起こすことで，登録されていない要素に対して陽性を返すため偽陽性となる．

次に，要素 41 をフィルタに登録したとする．このとき，ある要素の帰属判定の結果は次のようにばらける．(1) 真陽性についてこれは，12，41 の 2 通りである．(2) 偽陽性について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．それゆえ，二つの要素がフィルタに登録されている場合に，ある要素が参照したエントリにビットがセットされている確率は，64 × 2/8 = 16 とおりである．偽陽性となる場合の数は, このうち真陽性である場合の数を除いて，14 通りである．(3) 陰性について全ての場合の数から (1)，(2) の場合の数を除いて 48 通りである．

3.2 ブルーム・フィルタの動作例

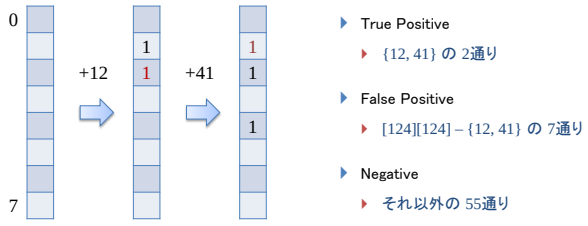

図 7 ブルーム・フィルタの例

BF は 3 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有る．**図 7** を用いて簡単に BF の動作を説明する．図 6 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，キー値の八の位と，一の位の k = 2 種であるのに，大きな違いが有る．

HF の例と同様にして，BF に要素 12 を登録すると，ハッシュ関数が 2 種であるため，001 と 010 の二箇所にビットがセットされる．先程の例のように，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 と変わらず要素 12 については陽性を，素 14 に対しては陰性を

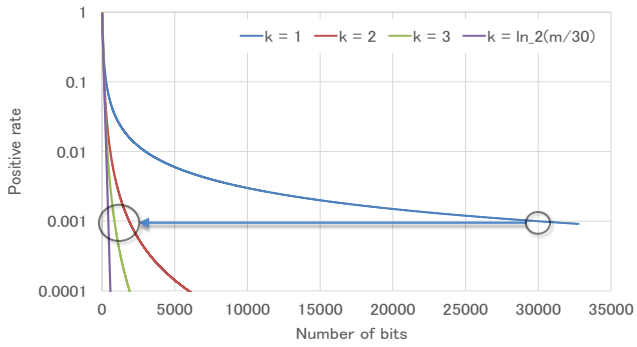

図 8 ブルーム・フィルタの解析解

示す．その一方で，要素 56 に関しては二つのハッシュ値が，101 と 110 となるため，HF の例と打って変わって陰性を示す．

続いて，要素 14 をフィルタに登録した時，フィルタの返す判定結果ついては次のようになる．(1) 真陽性について 12，41 の 2 通りである．(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．これはハッシュ関数の数が増えたことで細かな判定ができるようなったことを示している．このハッシュ関数の数の違いが大幅に BF の偽陽性率を下げている．

3.3 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，

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

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 8** 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 [4], [5], **?**.

4. 提案手法:Bloom-like SVW

SVW のようなシーケンス・ナンバを用いた順序違反検出手法においては，複数のハッシュ関数を用いる手法がこれまでに知られていなかった．

本章では，まずブルーム・フィルタの動作原理を説明してから，BF の一般化を行う．それから，提案手法の原理について説明を行い，それから提案手法の動作，そして詳細について説明を行う．

4.1 ブルーム・フィルタの原理と一般化

BF は複数のハッシュ関数を用いてフィルタの該当エントリをセット/リード することによって，要素と集合の帰属関係を判定するフィルタである．これは，複数の HF を集め統合したフィルタで，それぞれのフィルタで異なるハッシュ関数を用いるフィルタと言い換えることができる．このように言い換えた時，BF の動作は次のように表せる．

■要素となる HF のうち，少なくともひとつでも陰性ならば，BF は全体として陰性

■要素となる HF のうち，全てが陽性ならば，BF は全体として陽性

このことから，BF の陽性率は，それぞれの HF の陽性率の積で表すことができる．よって，BF は，HF の数を増やすことで，陽性率，つまり偽陽性率を大幅に削減できる．この動作は HF に偽陽性はあるが，偽陰性がないという性質により，成立している．フィルタ一般には，偽陽性はあるが，偽陰性はないという性質があるため，BF の動作は次のように一般化できる．

■要素となる一般のフィルタのうち，少なくともひとつでも陰性ならば，フィルタは全体として陰性
■要素となる一般のフィルタのうち，全てが陽性ならば，フィルタは全体として陽性

これはつまり，異なるハッシュでアクセスされるフィルタを複数集めたフィルタは，劇的に偽陽性率を削減できることを示している．

4.2 Bloom-like SVW の動作

SVW を BF のように用いることはできない．それは，SVW において順序違反検出を行うフィルタである，SSN 表 (SSNT) の構成要素はビットではなく，シーケンシャル番号であるからである．それゆえ提案手法では，SVW を要素フィルタとして扱うことにする．つまり，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 に用いて全体としてフィルタを構成している．Bloom-like SVW では，要素フィルタである SVW の少なくともひとつで陰性であるならば全体として陰性を返し，要素フィルタである SVW の全てで陽性であるならば全体として陽性を返す．

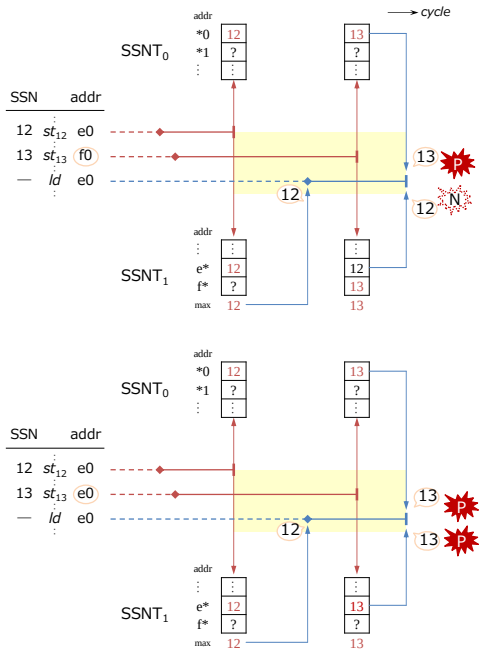

図 9 Bloom-like SVW の順序違反検出
順序違反がない場合（上）とある場合（下）

図 9 は，提案手法における順序違反検出例である．この図では，2.2 章の図 4 と概ね同じ構成となっている．図 4 と異なる点は，図 4 ではアドレスをハッシュとしていた SSNT が，図 9 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 つ有る点である．図 9 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，st13 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 9 の下図は，st13 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，st13 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．

4.3 Bloom-like SVW の詳細

サイズの問題への対応

メモリ命令がアクセスするデータのサイズは，1B,2B,4B,8B と複数ある．それに対し SVW では，8B 単位で順序違反検出を行っており，SSNT には下位 3bit の情報が切られてアクセスされる．そのため，SVW では，下位 3bit は異なるのに，上位のビットが一致するために，順序違反として検出されてしまうことがある．

このサイズの問題による偽陽性を回避するために Bloom-like SVW では，4B 単位での SSNT への読み書きを行うことで順序違反を検出するようにしている．

SQ の CAM 省略手法

Bloom-like SVW では，投機フォワ-ディングを行うために，依存予測器をもちいる．この依存予測器には，NoSQ で用いられている動的な命令間距離を用いる依存予測器を利用する．この予測器では，ロード命令のみで予測ができるため，Store Set のように，ストア命令の情報を保持するテーブルは必要が無い．

5. 評価

本章では，Bloom-like SVW における ハッシュ関数の数と IPC および偽陽性率について評価を行った．また我々がかつて提案した BF による順序違反検出手法と，比較評価を行った．

5.1 評価環境

ベンチマーク

ベンチマークは SPEC CPU 2006 [12] 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 [13] を用いた．ベースラインとなるプロセッサの構成は**表 1** に示す通りである．

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．

依存予測器としては，Store Set [11] を用い，評価においては，実装が間に合わなかったため，SQ-CAM の簡略化手法は適用していない．

5.2 IPC と偽陽性率の評価

本節では，提案手法のハッシュ関数の数 k の効果を評価する．

また評価において提案手法特有のパラメタは，SSN は 16 ビットのシーケンス・ナンバ，SPCT のエントリ数は我々の事前評価において最も良い性能を示した 16 エントリとした．

またロード再実行による確認検査は，陽性が検出された直後の 1 サイクルで可能であるとし，バックエンドを 2 サイクル，ストールすることによって完了できると理想化した．

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幾何平均を示す．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，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．

図 10 の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

また，グラフ中の曲線はそれぞれ $k = 1, 2, \ldots, 5$ の場合で，$SSNT_k$ のエントリ数をそれぞれ，$8, 16, 32, \ldots$ と変化させたものである．

このグラフを見てもらえば分かるように，フィルタの容量が増えれば，偽陽性率が下がることが確認される．また，k の数が増えるに従って，フィルタの偽陽性率は下がることが確認された．これにより，フィルタの相対 IPC はハッシュ関数の数に従って上がることも同時に確認された．また，相対 IPC の低下率を 1% 以内としたとき，k = 1 では，およそ 65,000 Bit が必要となるのに対し，k = 2 では 16,000 Bit であり，およそ 5 分の 1 程度の差があった．

このように $k = 1$ 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ことが分かる．

表 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

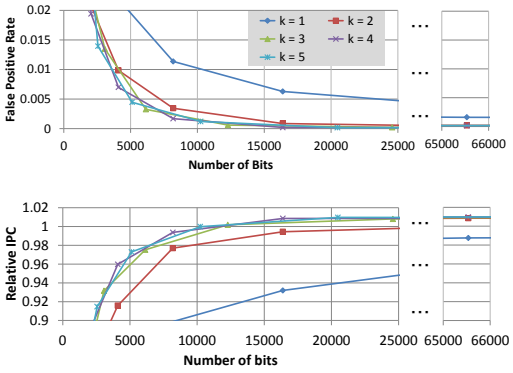

図 10 k の効果:偽陽性率 (上) と相対 IPC(下)

5.3 PCBF との比較

本節では，我々がかつて提案した PCBF による順序違反検出手法と比較する．詳しい説明に関しては文献 [7] を参照していただきたい．

PCBF の評価モデル

PCBF は 4B ワード単位で順序違反検出が可能なモデルを用いた．コレは，提案手法の評価に条件を合わせたものである．また手法に特有のパラメタや，具体的な実現方法については文献 [7] に記載されている値に従った．また Bloom-like SVW ではハッシュ関数の数 k=1, 4 を，PCBF による手法では k = 4 として，比較評価を行った．

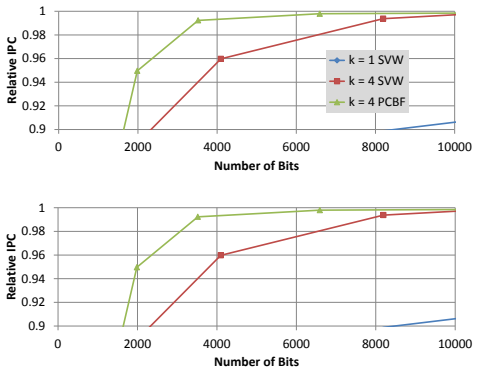

図 11 それぞれの手法の偽陽性率 (上)，相対 IPC(下)

この評価では，Bloom-like SVW は k の数を増やしたことによって，劇的な性能低下は得られているが，PCBF と比べて性能が下がっている事がわかる．例えば，偽陽性発

生率を 1%以内を目標にした場合，PCBF では，2000 Bit，Bloom-like SVW では，4000 Bit 程度必要である．

この性能差は，SVW の要素フィルタの構成要素がビットであるのに対して，PCBF の要素フィルタの構成要素がカウンタであることから生じている．

6. おわりに

本稿では，最新のプロセッサにおける LSQ がコストの高いロジックの一つであることを説明し LQ を簡略化する手法である，Bloom-like SVW を提案した．また，Bloom-like SVW の簡単な評価を行い，ハッシュ関数の数を増やすことで劇的に偽陽性発生率を下がることを確認した．今後は，SQ-CAM の簡略化手法である投機フォワーディングを実装し，評価を行う予定である．

参考文献

[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] 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).

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

[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] Roth, A.: Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, *32nd International Symposium on Computer Architecture (ISCA'05)*, 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, pp. 297–308 (2006).

[7] 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).

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

[9] 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 (2006).

[10] 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).

[11] 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).

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

[13] 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).