Vol.2014-ARC-212 No.17
2014/10/7

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

倉田 成己[1,a)]　塩谷 亮太[2]　五島 正裕[3]　坂井 修一[1]

概要：
本項では，高コストな CAM を用いず，RAM によって構成されたブルーム・フィルタ (**BF**) を用いるメモリ・アクセス順序違反/フォワーディング・ミス検出手法を提案する．BF は，複数のハッシュ関数を用いることによる，極めて低い偽陽性率に特徴がある．フィルタを用いる順序違反/フォワーディング・ミス検出手法はいくつか提案されており，そのうちのいくつかは BF を用いているとしているが，複数のハッシュ関数に言及，および，評価した研究はない．提案では更に，ロード/ストア命令のサイズの違いに対する対策なども行っている．評価の結果，BF の低い偽陽性率などにより，1888b 程度のフィルタで 99.6%の IPC を維持できることが分かった．

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と，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，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．

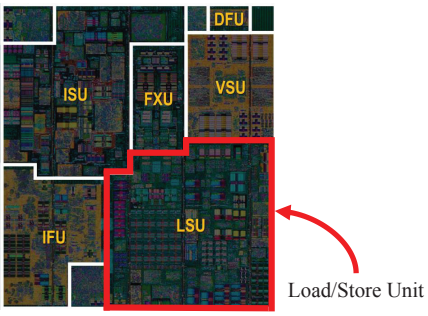

図 1　POWER8 のチップ写真 [3]

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** に，POWER8 プロセッサのチップ写真を示す [3]．LSU がコアの 1/4 程度を占めている．L1D アレイの占める割合は高くなく，LSQ の面積と消費電力の削減が極めて重要であることが分かる．

フィルタを用いた順序違反検出

そのため，CAM の電力を削減したり，CAM を省略できる LSQ が提案されている [4], [5], [6]．これらは，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．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

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
[2] 名古屋大学大学院工学研究科
Graduate School of Engineering, Nagoya University
[3] 国立情報学研究所
National Institute of Informatics
[a)] kurata@mtl.t.u-tokyo.ac.jp

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．

本稿では，このフィルタとして，**パラレル・カウンティング・ブルーム・フィルタ**を用いる手法を提案する．**ブルーム・フィルタ** [7] 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．過去にはブルーム・フィルタを用いたとする手法はいくつか提案されている [4], [5], [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，サイズの割に高い偽陽性率に苦しんでいる．我々の知る限り，既存研究で複数のハッシュ関数に言及したものはなく，本稿は順序違反/フォワーディング・ミス検出において複数のハッシュ関数を評価した初めての論文となるであろう．

本稿の構成

本稿の構成は以下のとおりである．まず 2 章で，本稿提案のポイントであるブルーム・フィルタについて説明し，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．ついで，3 章で，既存手法と提案手法を含む，フィルタを用いた順序違反/フォワーディング・ミス検出についてまとめる．その上で，既存手法の問題点を明らかにする．その後，4 章で提案手法の詳細について説明し，5 章で評価を行う．

2. ブルーム・フィルタ

提案手法の第一の特長は，フィルタとして**ブルーム・フィルタ** (**BF**) [7] を用いることにある．本章では BF について説明する．まず次節で，BF の基本について説明する．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．その後，2.3，2.4 節で，実際に提案で用いる**パラレル・カウンティング・ブルーム・フィルタ**を紹介する．

2.1 ブルーム・フィルタ

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 [7]．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．

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** 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 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．

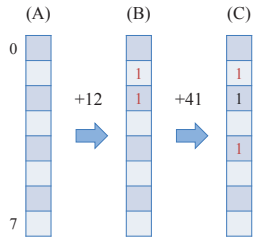

図 2 ブルーム・フィルタの例

(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 (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，以下でのべるように，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，

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

となる．この式からでも，m を増加させるより，k をわず

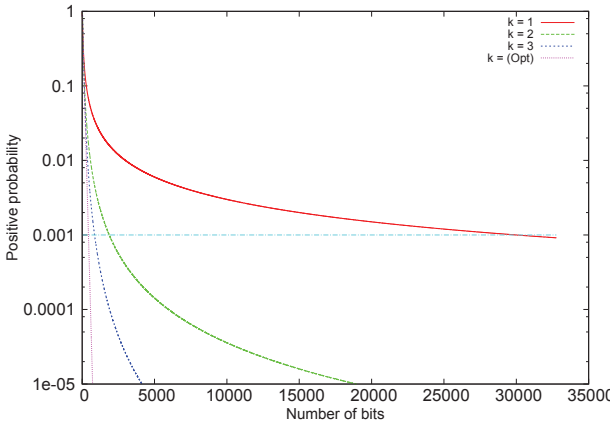

図 3 BF のエントリ数と 陽性率の関係

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 3** 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 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 [8]．PBF では，エントリ数 m の配列を，エントリ数 $m' = m/k$ の k 個の部分配列に分割し，各部分配列は，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（A.1.1 参照）．すると，各部分配列を構成する k 個の RAM のポート数はそれぞれ 1 で済む．

PBF の陽性率 P_{para} は，オリジナルの BF に比べると若干大きいが，実用領域ではその差はほとんどない（A.1.2 参照）．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 [9]．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・フィルタを用いる．

3. フィルタを用いた検出手法

本章では，既存手法と提案手法を含む，フィルタを用いてメモリ・アクセス順序違反/フォワーディング・ミス検出を行う手法についてまとめる．

以下，3.1 節で，ベースとなるプロセッサのモデルについて述べ，順序違反検出とフォワーディング・ミス検出が双子の問題であることを明らかにする．3.2 節で手法の分類について述べた後，3.4 節からは，既存手法として，SVW，DMDC などについて個別に説明する．

3.1 順序違反検出とフォワーディング・ミス検出

1 章で述べたように，out-of-order スーパスカラ・プロセッサにおいて，**ロード・キュー** (**LQ**) と**ストア・キュー** (**SQ**) は，ロード/ストア命令の依存関係を守りつつ，out-of-order に実行する役割を果たす．ロード/ストア命令は曖昧であり，依存関係を守るためには，動的なターゲット・アドレスの比較が必須である．

LSQ の CAM

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 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 [10], [11] 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．

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．次節以降で述べる手法では，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 の問題の半分以下しか解決していないことになる．

投機的フォワーディング

SQ の CAM を省略するためには，更に**投機的フォワーディング**を組み合わせることになる [6], [12], [13]．予測には，依存予測器の結果をそのまま用いることができる．先行するストア命令に依存すると予測されたロード命令は，当該ストア命令の発行後に発行されると同時に，当該ストア命令から直接ストア・データを受け取ればよい．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．

次節以降で述べる SVW と提案手法では，これら 2 つの問題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．

3.2 フィルタを用いた手法の分類

RAM によって作られたフィルタを用いる手法の基本は，

(1) 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テーブル (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．

表 1 各手法の比較

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

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，テーブル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出を行うのである．

表 1 に，本章で紹介する各手法をまとめる．

各手法は，① テーブルの構成 と，② テーブルへのアクセス 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 テーブルの構成

いずれの手法においてもテーブルのキーにはロード/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が，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 テーブルへのアクセス

テーブルへのライト → 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，前述のテーブルのバリューが b. ビット の場合には，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．

以下，提案手法が最もシンプルなので，それを用いて順序違反検出とフォワーディング・ミス検出の基本的な動作を説明し，その後，3.4 から既存手法の動作について説明する．

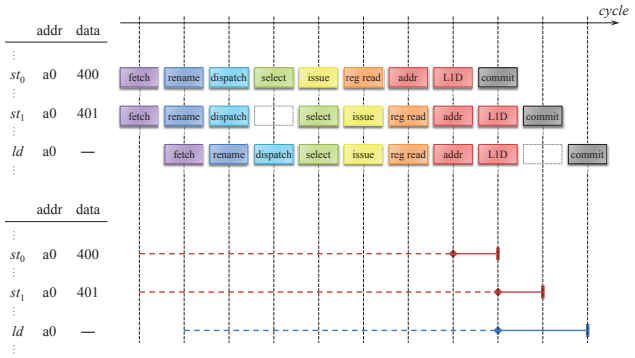

図 4 パイプラインの表記

3.3 フィルタを用いた検出手法

本節では，提案手法を例に，フィルタによる順序違反/フォワーディング・ミス検出の基本的な方法ついて説明する．

パイプライン図の表記法

各手法の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・ステージの前後関係のみである．したがって，**図 4**（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 4（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．

順序違反検出

この簡略化されたパイプライン図を用いて，**図 5** に提案手法の順序違反検出の様子を示す．

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．

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，実行時にテーブルのビットをセットし，ストア命令は，コミット時にテーブルをリードしてビットを検査する．

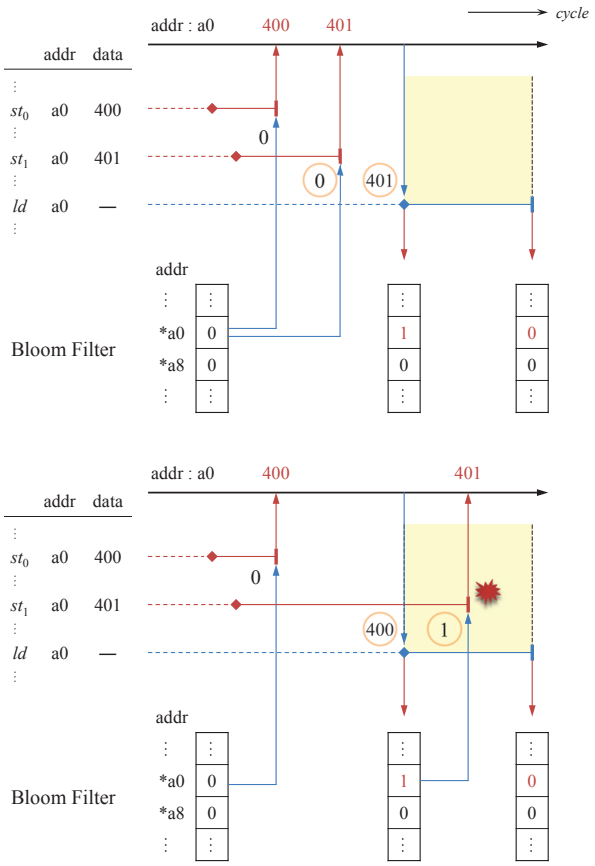

図 5 提案手法の順序違反検出
順序違反がない場合（上）とある場合（下）

フォワーディング・ミス検出

図 6 に提案手法のフォワーディング・ミス検出の様子を示す．同図では，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．

提案手法では，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，フォワーディングを行った場合には，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テーブル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 がテーブルから 1 をリードし，フォワーディング・

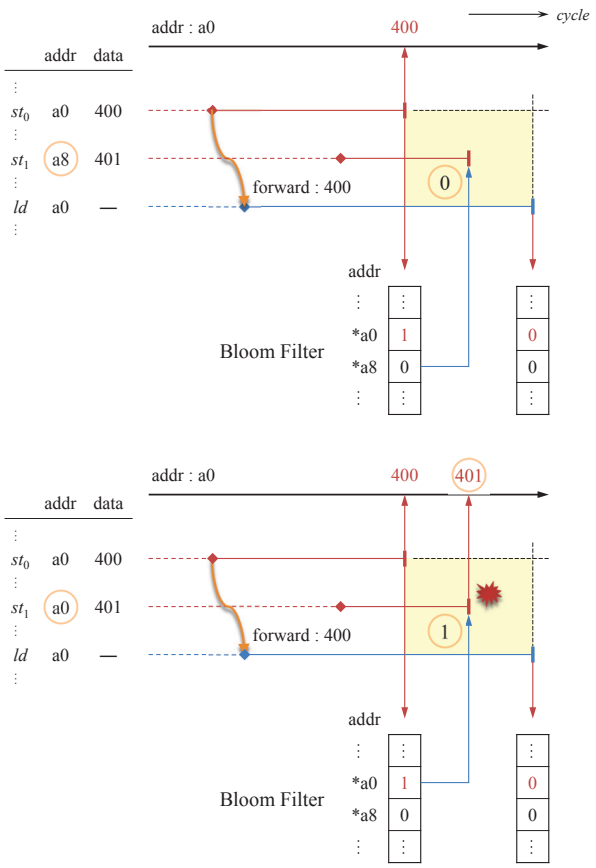

図 6 提案手法のフォワーディング・ミス検出
フォワーディング・ミスがない場合（上）とある場合（下）

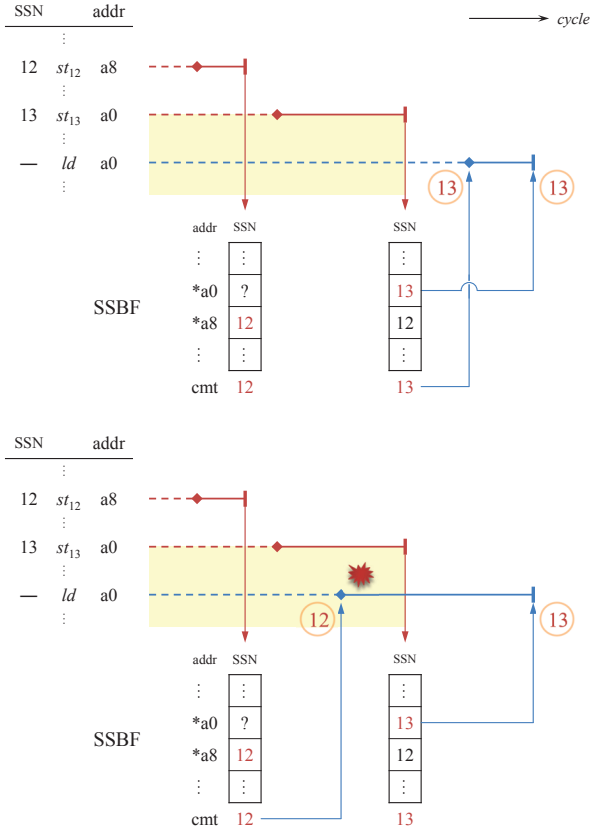

図 7 SVW の順序違反検出
順序違反がない場合（上）とある場合（下）

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テーブルはハッシュ・テーブルであり，ハッシュ値の偶然の**衝突**による偽陽性 (false positive, **偽陽性**) 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．

3.4 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．

3.2 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

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

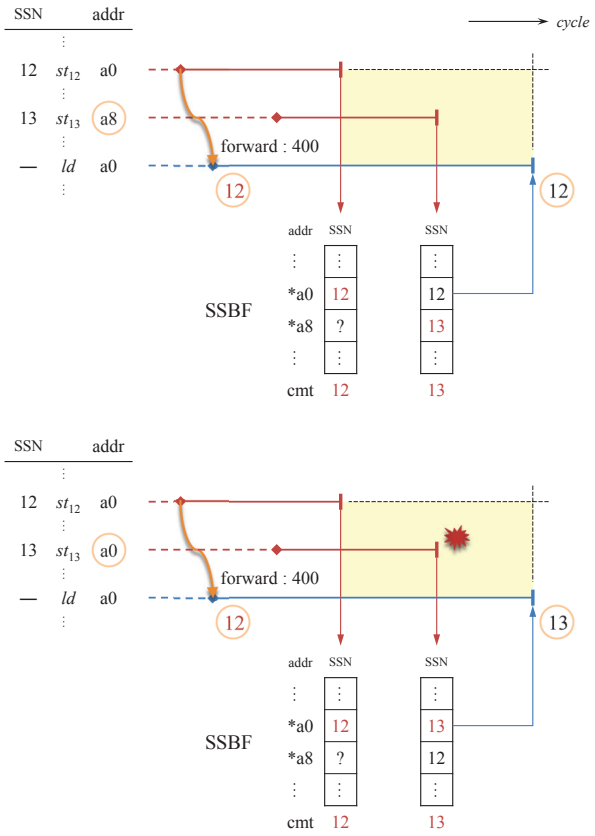

図 8 SVW のフォワーディング・ミス検出
フォワーディング・ミスがない場合（上）とある場合（下）

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 SSN_{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，SSN_{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

3.2 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 7** 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 SSN_{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 SSN_{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 7（下）は，順序違反がある場合の動作である．ld は，実行時に SSN_{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，SSN_{cmt}12 より大きいため，順序違反が発生したことが分かる．

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，SSN_{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，SSN_{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 8（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，5 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．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 [10], [11].

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 → 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．

3.5 Delayed Memory Dependence Checking

Delayed Memory Dependence Checking (**DMDC**) [14] 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・テーブルを用いて，同一アドレスに対するものであるか検証する．

前半

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

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．

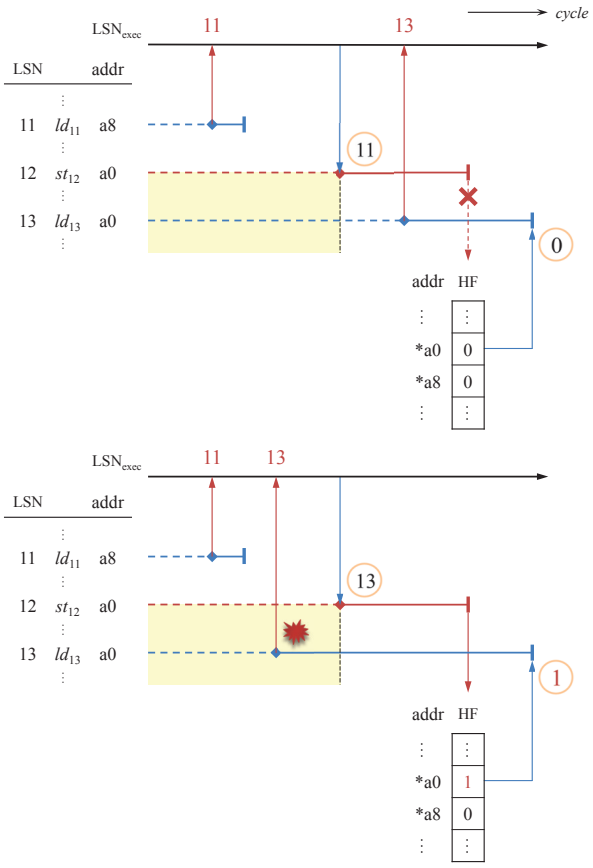

図 9 DMDC の順序違反検出
順序違反がない場合（上）とある場合（下）

順序違反検出

図 9（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 9（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 LSN_{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．

[*2] LSN と LSN_{exec} は，文献 [14] 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．

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 9（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 9（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れたとき，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 [14] 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 [14] 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 [6], [10], [11].

3.6 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．3.2 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．5 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．

3.7 本章のまとめ

本章でとりあげた手法を，表 1 にまとめる．本章の議論において問題があると指摘した項目を網掛けで示した．既存手法と提案手法との違い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．

4. 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．本手法では，CAM や，フィルタを用いた従来の手法と比較して，回路面積と消費電力を大幅に削減しながら，CAM に匹敵する性能を達成できる．

提案手法の動作は，基本的には 3.3 で述べたとおりである．ただし，ブルーム・フィルタではなく，カウンティング・ブルーム・フィルタを用いるため，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．

本章では，提案手法の詳細について述べる．以下，まず陽性やオーバーフローが発生した際の動作について述べる．その後，4.3 節以降で，提案手法のフィルタの詳細な構成について説明する．

4.1 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 の低い偽陽性率により，コストの高い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．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，特に何もせず，実行を再開すればよい．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 [10], [11] 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 の検索には，LQ の発行ポートを用いることができる．LQ の発行ポート数 1〜2 程度で，LQ 内のロードの 32〜64 命令程度を検索するとすると，バックエンドがストールするサイクル数は平均で数十サイクル程度にもなる．しかし，5 章で述べ

```
for (i = 0; i < (1 << reg->hashw); i++)
        reg->hash[i] = 0;
```

図 10 同一アドレスに対するロード命令が頻繁に実行される例

るように，PCBF の低い陽性率ならば，この程度のペナルティにも耐えることができる．

4.2 オーバーフローへの対処

陽性が k 個のハッシュ値すべてが一致しないと起こらないのに対して，オーバーフローは，k 個のハッシュのうち 1 個でもフルになっていると起こるので，より問題である．

オーバーフローに際して，単に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．まずバックエンド・パイプライ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．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．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．5 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．

4.3 同一アドレスに対するロード命令の頻発

オーバーフローが発生する原因には，ハッシュ値の偶然の衝突により発生する場合と，同一のアドレスに対するロード命令が短期間のうちに頻繁に実行される場合が挙げられる．前者は，確率的な事象であるため，各カウンタのビット数を十分に増やす以外には解決方法はない．本項では，特に後者について考える．

同一アドレスに対するロードの頻発

同一アドレスへのロードは，主に，プログラミングの拙さやポインタ解析の失敗により発生するものと考えられる．**図 10** に，SPEC 2006 の libquantum の一部を示す．同図中の for 文の脱出条件にある reg−>hashw は，ループ不変であるにも関わらず，毎回ロードされている．

このような問題の多くはコードの修正で治りはするが，a. シーケンス・ナンバ を用いる手法では特に問題なく動作するため，b. ビット を用いる SHF や本提案でも対処する必要があろう．なお，3.6 節で紹介した Single Hash Filter を用いた手法 [4] では，この問題につての言及はなく，カウンタのビット数を一律に増加させることによってこの問題に対処している．

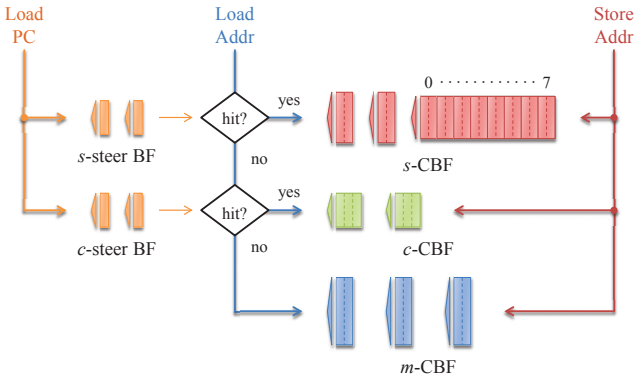

図 11 提案フィルタの構成

c-CBF

本提案では，エントリ数 m は少ないがカウンタのビット数 c は大きい CBF を別途用意することで解決する．ここで，メインの m が大きい PCBF を ***m*-CBF**，c が大きい PCBF を ***c*-CBF** と呼ぶ．

***c*-steer BF**

評価の結果，オーバーフローを頻繁に引き起こすロード命令は静的に同一の命令であることが多いことが分かった．そこで，m-CBF と c-CBF どちらに追加するかの選択は，オーバーフローを起こしたロード命令の PC を用いておこなう．

振り分けには，ロード命令の PC を要素とする BF，***c*-steer BF** を用いる．ロードが PCBF を更新する際には，まず c-steer BF にアクセスし，陽性であれば c-CBF を更新し，陰性であれば m-CBF を更新する．c-steer BF は m-CBF や c-CBF と違い，カウンティングではない通常の BF である．ロードがオーバーフローを起こした場合，その命令の PC を c-steer BF に追加する．c-steer BF のリセットは，一定のサイクル —— 5 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．

図 11 に示すように，3 種の PCBF と 2 種の BF を組み合わせた構成をとる．

ロード命令は，c-steer BF を用いて，m-CBF，c-CBF のいずれかのエントリをインクリメント/デクリメントする．一方ストア 命令は，双方に対してリードを行う．リードを行った結果，いずれか一方でもヒットすれば陽性とする．

4.4 ロード/ストア命令のサイズへの対応

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．5 章の評価に用いた命令セットは，Alpha であるが，byte-word extension を採用しており，1，2 B のロード/ストア命令を持つ．

例えば，アドレス 0x154〜0x157 の 4 B にアクセスするストア命令の後に，0x156〜0x157 の 2 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，3.4 節で述べた 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 0x150〜157 までの 8 B ワードへのアクセスとみなし，アドレスを 0x150 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．

（C 言語で言えば）long の代わりに short や char 型を用いることは，それらが巨大配列の要素であった場合，極めて重要な最適化となり得る．そのため，1〜2B ワードに対するのロード/ストア命令は今後も重要な役割を果たすと考えられる．評価に用いた SPEC CPU2006 ベンチマーク [15] では，h264ref，astar，xalancbmk などがこのようなプログラムにあたる．5 章では，10 個のプログラムをこのようなクラスに分類して評価した．

***s*-CBF**

本項の提案では，サイズを考慮した PCBF である ***s*-CBF** を用意する．*s*-CBF は，*m*-CBF，*c*-CBF と同様，それぞれ 8 B ワードを単位としてアクセスする k 個の CBF からなる．ただし，k 個のうち 1 つのみのエントリが，*m*-CBF，*c*-CBF とは異なっている．$k-1$ 個は，*m*-CBF，*c*-CBF と同じ通常の CBF で，各エントリは c ビットのカウンタである．残り 1 つは，それらとは異なり，各エントリが c ビットのカウンタ 8 個からなっている．8 個のカウンタうち，実際にロード/ストアの対象となったバイトに対応するカウンタだけが，インクリメント/デクリメント，そして参照されることになる．

PCBF のため，テーブルが k 個あることを利用して，そのうち 1 個のみをバイトごとに拡張することで，全体の利用効率を上げる訳である．

***s*-steer BF**

s-CBF に振り分けるかどうかは，*c*-CBF と同様に，ロード命令の PC を要素とする BF，***s*-steer BF** を用いる．*s*-steer BF は，追加される PC が異なる他は，*c*-CBF と同じものである．また，ストア命令による検索は，3 種すべての CBF に対して行われる．

5. 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．以下，5.1 節で評価環境についてまとめた後，5.2 節で提案手法の評価を行い，5.3 節で既存手法との比較を行う．

5.1 評価環境

ベンチマーク

ベンチマークは SPEC CPU 2006 [15] の全 29 プログラムで，データ・セットは *ref* を使用した．各プログラムは gcc 4.6.1 の -O3 オプションでコンパイルした．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．

主に提案手法の工夫を評価するために，**表 2** に示すように，プログラムを 3 種のクラスに分類した．class *c*/class *s* は，それぞれ，4.3/4.4 節で述べた 2 つの問題が顕著に現れるプログラムで，主に *c*-CBF/*s*-CBF の効果を測るために用いる．class *m* は，これらの問題が現れないプログラムである．一部のプログラムのみを用いるのは，「お行儀のよい」プログラムと平均化することで悪影響が過少に評価されることを防ぐためで，最悪ケースを評価しているのだと理解されたい．

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 [16] を用いた．ベースラインとなるプロセッサの構成は**表 3** 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．

また，依存予測器としては，Store Set [10] を用いた．

5.2 提案手法の評価

本節ではまず，以下に示す，提案手法の工夫の効果を評価する：

(1) k の効果
(2) c の効果
(3) *c*-CBF，*s*-CBF の効果
(4) オーバーフロー時のストールの効果

表 4 に，提案手法のデフォルトのパラメタを示す．特に断りのない場合には，これらの値を用いた．

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

表 2 SPEC CPU 2006 ベンチマークのプログラムの分類

クラス	プログラム
m	bwaves, zeusmp, soplex, povray, calculix, GemsFDTD, lbm, omnetpp, wrf, sphinx3
c	perlbench, gamess, mcf, milc, cactusADM, leslie3d, namd, dealII, libquantum, xalancbmk
s	bzip2, gcc, gromacs, gobmk, hmmer, sjeng, h264ref, tonto, astar

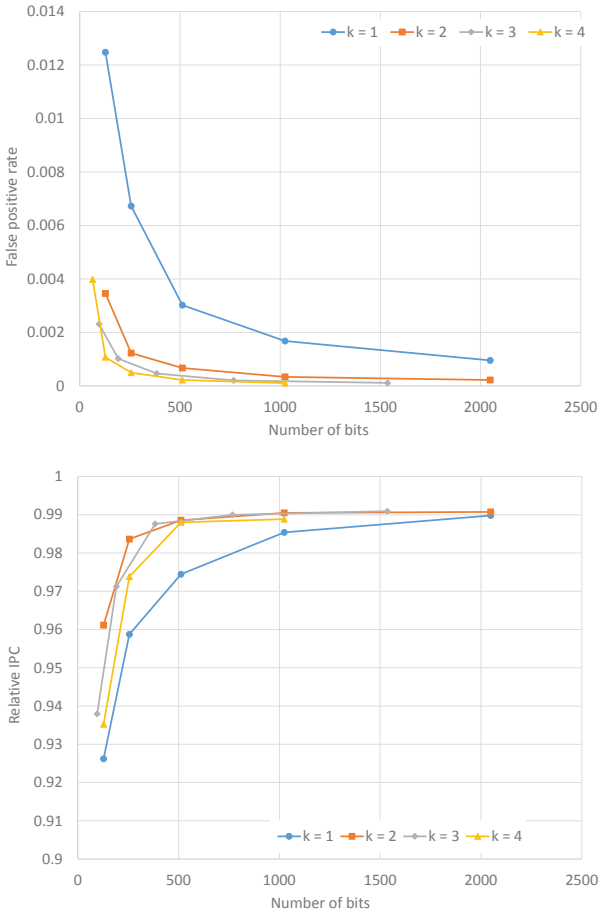

図 12 k の効果 偽陽性率（上）と相対 IPC（下）

陽性による性能低下はない．いくつかのグラフを示すが，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．

***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，シミュレーションによって，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．

表 3 プロセッサの構成

パラメタ	値
ISA	Alpha 21264A w/ byte-word ext.
fetch/cmt	4/4 inst./cycle
inst window	int : 32, fp : 16, mem : 16
ROB	128 entries
LQ/SQ	48/48 entries
branch pred	8KB g-share
miss penalty	10 cycles
BTB	2K-entry, 4-way
L1D	32KB, 4-way, 64B/line, 3 cycles
L2C	4MB, 8-way, 64B/line, 15 cycles
main memory	200 cycles

表 4 提案手法の各 BF のパラメタ

BF	$m' \times k \times c \times \# =$ total
m-CBF	$128 \times 3 \times 2 \times 1 =$ 768
c-CBF	$32 \times 4 \times 4 \times 1 =$ 512
s-CBF	$16 \times 2 \times 3 \times 1 =$ 96
	$16 \times 1 \times 2 \times 8 =$ 256
c-steer BF	$64 \times 2 \times 1 \times 1 =$ 128
s-steer BF	$64 \times 2 \times 1 \times 1 =$ 128
total	1888

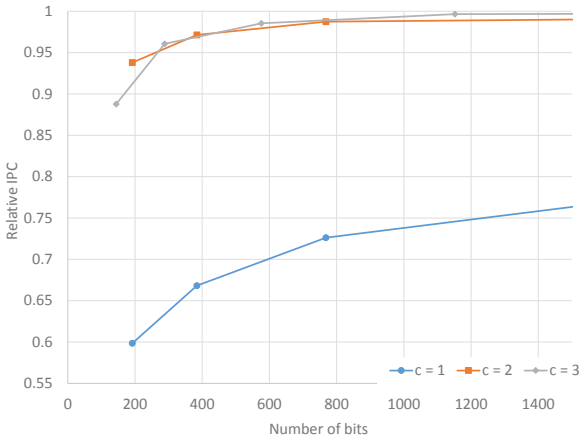

図 13 c の効果 相対 IPC

図 12 に，結果を示す．フィルタは，m-CBF のみ，ベンチマークは class m のみを用いた．曲線は，$k = 1$，2，3，4 ごとに m を変化させたものである．なお，$c = 2$ に固定した．

$k = 1$ 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．$k = 4$ では，偽陽性率は減少しているものの，相対 IPC は逆に悪化している．これは，ハッシュ関数 1 つあたりに占めるエントリ数が減少し，オーバーフローが増加しているためと考えられる．したがって，以降では，$k = 3$ に固定する．またこの場合の m は，たかだか $m = 128 \times 3$ 程度でよい．

***c* の効果**

CBF でオーバーフローが発生する原因は，ハッシュ値の偶然の衝突と，4.3 で述べた，同一アドレスに対するロード命令の頻発がある．後者については次の項で評価することとし，この項では前者について考える．

前者に対しては，m を無闇に増加させるよりも c をわずかに増加させた方が効率がよいことを示す．そのため，フィルタは m-CBF のみ，ベンチマークは class m のみを用いた．

図 13 に結果を示す．曲線は，$c = 1$，2，3 ごとに m を変化させたものである．

$c = 1$ の場合と比較して $c \geq 2$ が小エントリで IPC を維持できており，特に $c = 2$ の場合の効率がよいことがわかる．したがって，以降 $c = 2$ に固定する．

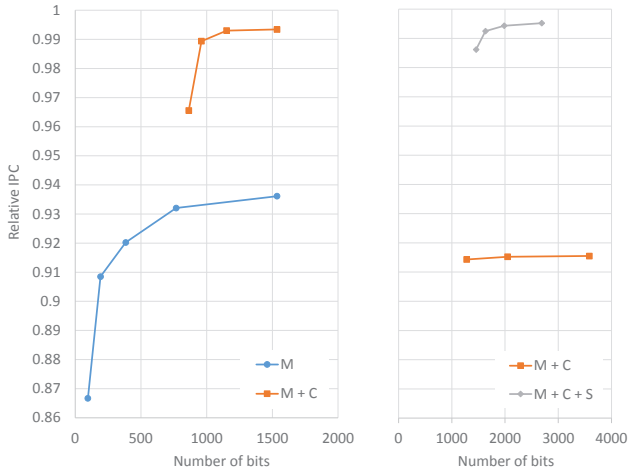

図 14 *c*-CBF の効果（左）と，*s*-CBF の効果（右） 相対 IPC

c-CBF，*s*-CBF の効果

上述した同一アドレスに対するロード命令の頻発の問題と，4.4 節で述べたロード/ストア命令のサイズによる問題のそれぞれに対して，*c*-CBF と *s*-CBF を組み合わせることで総ビット数が少なくて済むことを示す．

評価モデルは，*m*-CBF のみを用いるものと，*c*-CBF を組み合わせる場合，更に *s*-CBF を組み合わせ場合の 3 通りとした．ベンチマークは，それぞれ，class *c*，class *s* を用いた．

図 14 に，結果を示す．*m*-CBF は，$k = 3$，$c = 2$ に固定した．事前の評価により，*c*-CBF，*s*-CBF の *c* は，それぞれ，4，2 で十分であることが分かったので，その値に固定した．各図において，M は *m*-CBF のみ，$M + C$ は *m*-CBF と *c*-CBF を，$M + C + S$ は全ての CBF を組み合わせたモデルを表す．図 14（左）については，M モデルでは *m*-CBF の *m* を変化させ，$M + C$ モデルは *c*-CBF の *m* を変化させた．また図 14（右）については，$M + C$ モデルでは *m*-CBF の *m* を変化させ，$M + C + S$ モデルは *s*-CBF の *m* を変化させた．

m-CBF のみを用いた場合と比較して，class *c* のベンチマークでは *c*-CBF を，class *s* のベンチマークでは *s*-CBF を組み合わせることで，それぞれ 99%以上の IPC を達成できていることがわかる．

オーバーフロー時のストールによる効果

4.2 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，少し待ってからフラッシュした方がよい．

図 15 に，各ベンチマークにおいてオーバーフローが発生したときにフラッシュされる命令の割合と，オーバーフローが発生したときの平均ストール・サイクルを示す．フラッシュされる命令の割合はベンチマークによってまちまちであるが，平均約 51.8%とおよそ半分がフラッシュの必要なく実行が再開されていることがわかる．また，平均ストール・サイクルは namd を除くいずれのベンチマークに関しても数サイクル以内であり，少し待つだけで実際にフラッシュが必要か判断し，実行再開もしくはフラッシュされていることがわかる．namd に関してのみ，平均ストール・サイクルが 7.55 サイクルと長くなっているが，namd はフラッシュが必要な命令の割合がほとんどなく，ストールが有効に働いていると考えられる．

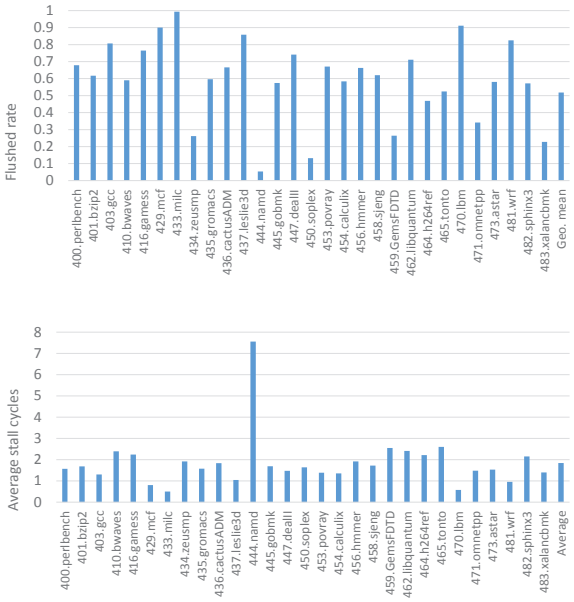

図 15 オーバーフロー時のストールによる効果
オーバーフロー発生数に対するフラッシュ率（上）と
平均ストール・サイクル（下）

表 5 評価モデル

	Filter	Forwarding	Confirmatory Test	Predictor Learning	FP Stall (cycles)
SVW	SVW	Speculative	Load Re-Exec	SPCT	1
DMDC	DMDC	SQ-CAM	N/A		Flush
SHF	CBF ($k = 1$)		LQ-CAM		1
CBF	CBF ($k \geq 2$)	Speculative	LQ Sequential Search		10s

5.3 既存手法との比較

最後に，各手法のフィルタのサイズと IPC との関係を評価する．

4.4 節で述べたロード/ストア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 [6], [14] 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，評価には class *s* を用いないこととする．ただし，提案手法の総ビット数には，*s*-CBF を含んでいる．

評価モデル

表 5 に，評価したモデルをに示す．表中，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ので注意され

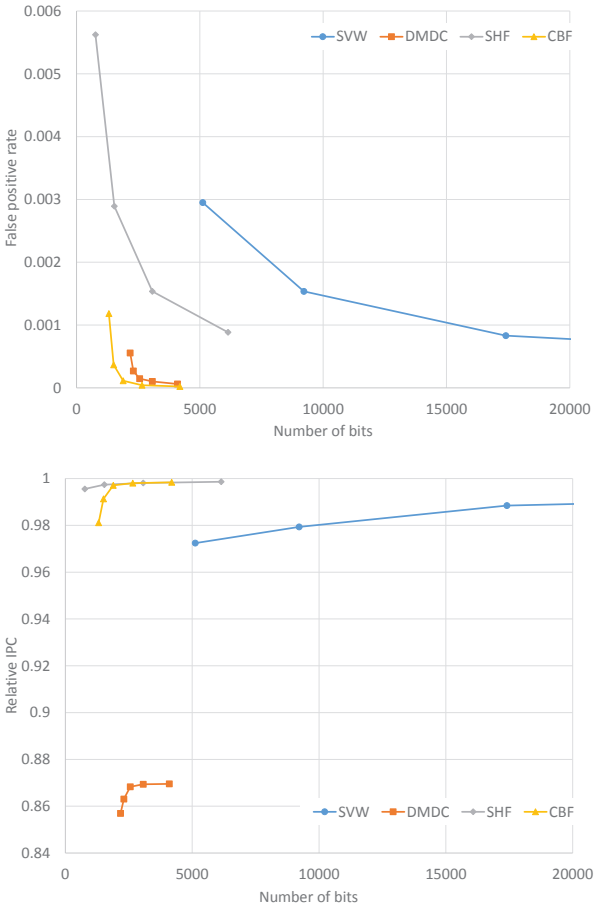

図 16 各モデルの比較 偽陽性率（上）相対 IPC（下）

たい：

- 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．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とした．
- 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とした．

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，3.4 節参照）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．

評価結果

図 16 に，結果を示す．グラフからは，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フィルタ容量が必要となる．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 のフィルタとしての性能は，SVW や SHF と比べると良いが，提案手法よりは若干悪い．また，偽陽性率の割に IPC が非常に低くなっている．これは，依存予測器を用いないため，偽陽性のみならず真陽性も頻発しているためである．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 [10], [11].
- 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．ただし SHF は，LQ-CAM 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. おわりに

フィルタを用いた順序違反/フォワーディング・ミス検出はいくつか提案されているが，それらは以下のような問題を抱えている：

- 複数のハッシュ関数を用いておらず，偽陽性率が高い．
- ロード命令のコミット時に検出する手法は，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 とは別の手立て必要とし，そのコストはフィルタ自体より高い．
- 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．

本稿で提案した BF を用いた順序違反/フォワーディング・ミス検出手法は，これらの問題をすべて解決している．特に，我々が知る限り $k \geq 2$ の BF を用いた提案はなく，本稿は，評価は順序違反/フォワーディング・ミス検出において $k \geq 2$ の BF を評価した最初の論文となるであろう．評価の結果，BF の低い偽陽性率などにより，1888bit 程度のフィルタで 99.6%の IPC を維持できることが分かった．

謝辞 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 による．

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, *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, Washington, DC, USA, IEEE Computer Society, pp. 399– (online), available from ⟨http://dl.acm.org/citation.cfm?id=956417.956553⟩ (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, *2005 International Conference on Computer Design*, 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] 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).

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

[9] 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).

[10] Chrysos, G. Z. and Emer, J. S.: Memory dependence prediction using store sets, *ACM SIGARCH Computer Architecture News*, Vol. 26, No. 3, pp. 142–153 (online), DOI: 10.1145/279361.279378 (1998).

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

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

[13] 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 (online), DOI: 10.1109/MICRO.2006.39 (2006).

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

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

[16] 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).

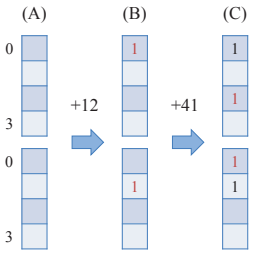

図 A·1 パラレル・ブルーム・フィルタの例

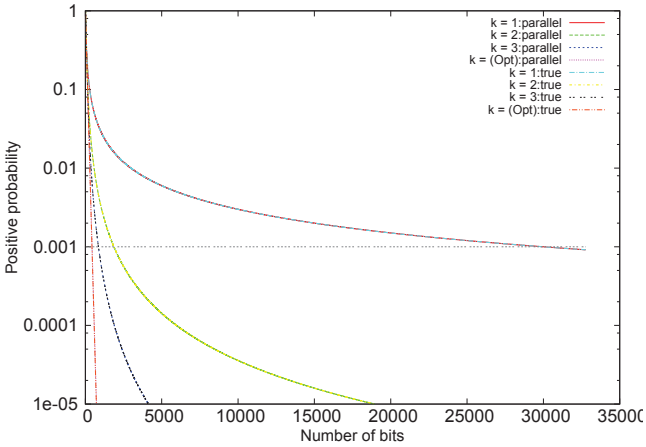

図 A·2 Parallel および True BF のエントリ数と positive 率

付　　録

A.1 パラレル・ブルーム・フィルタの詳細

A.1.1 パラレル・ブルーム・フィルタの動作

図 A·1 の例では，エントリ数 $8/2 = 4$ の部分配列を 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，上の部分配列に対してはエントリ $h_0(1) = 0$ にアクセスし，下の部分配列に対してはエントリ $h_1(2) = 1$ にアクセスする．

A.1.2 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：

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

これは，2.2 節で述べた P_{true} (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 3 と同じ条件下での P_{para} と P_{true} を，**図 A·2** 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．