ウェブクローラ向けの効率的な重複 URL 検出手法

久保田 展行† 上田 高徳‡ 山名 早人‡† ‡‡

† 早稲田大学理工学部コンピュータ・ネットワーク工学科 〒169-8555 東京都新宿区大久保 3-4-1

‡ 早稲田大学大学院基幹理工学研究科 〒169-8555 東京都新宿区大久保 3-4-1

‡† 早稲田大学理工学術院 〒169-8555 東京都新宿区大久保 3-4-1

‡‡ 国立情報学研究所 〒101-8430 東京都千代田区一ツ橋 2-1-2

E-mail: {nobu.k, ueda, yamana}@yama.info.waseda.ac.jp

あらまし ウェブクローラにおいて，冗長なクローリングを避けるための重複 URL 検出は必須の処理である．重複 URL 検出処理には，URL のハッシュ値を求め，その集合を元に重複検出を行う手法が一般的に用いられている．しかし，インターネット上には数百億の URL が存在するため，最終的には 100GB 以上のハッシュ値を管理する必要があり，メモリのみで処理することは困難となる．本稿では，新たな乱択データ構造である Stable Bloom Filter Light を提案し，ストレージ上でのハッシュ値集合操作回数を削減するための近似的なキャッシング手法を実現する．実験の結果，既存手法と比較して，同等のキャッシュヒット率のときにメモリ使用量を約 63%削減できた．

キーワード ウェブクローラ，重複検出，キャッシュ，Bloom Filter

Efficient Duplicated URL Detection for Web Crawlers

Nobuyuki KUBOTA† Takanori UEDA‡ and Hayato YAMANA‡† ‡‡

† Faculty of Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan

‡ Graduate School of Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan

‡† Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555 Japan

‡‡ National Institute of Informatics 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan

E-mail: {nobu.k, ueda, yamana}@yama.info.waseda.ac.jp

Abstract It is necessary for Web crawlers to avoid redundant crawling by detecting duplicated URLs. Web crawlers usually use the set of hash values generated from URLs to detect duplicates. However, the size of hash values will be over 100GB because the Web has billions of URLs. This large amount of hash values disables Web crawlers to detect duplicated URLs without storage accesses. In this paper, we present a novel randomized data structure called Stable Bloom Filter Light and develop a method using Stable Bloom Filter Light to reduce storage accesses. Experimental results show that our method cuts 63% of the memory size compared to existing methods for the same cache hit rate.

Keyword Web Crawler, Duplication detection, Cache, Bloom Filter

1 はじめに

ウェブクローラにとって，クロール済みのウェブページを再クロールするような，冗長なクローリングは致命的である．同一ウェブページに複数回アクセスすることは，クローリング対象のウェブサーバに不必要な負荷をかけるだけでなく，ウェブクローラ自体の性能も低下させる．同一ウェブページに対して短期間で複数回クロールしないようにするために，ウェブクローラはクローリング対象のウェブページがクロール済みであるかどうかを判断する必要がある．ウェブクローラでは，クロール済みのウェブページの URL を集合として持ち，クローリング対象のウェブページにアクセスする前に，URL 集合に対してアクセスしようとしている URL の包含判定を行うことで，そのウェブページへアクセスすべきか判断する．本稿では，URL 集合への包含判定処理を，重複 URL 検出と呼ぶ．

通常，重複 URL 検出をするにあたり，まず，URL は数バイトのハッシュ値へと変換される．我々の調査によると URL の平均長は約 70 バイトであるため，小さなハッシュ値へ変換することで，空間使用量を削減することが可能となる．次に，収集が完了した URL のハッシュ値は重複検出用の集合に入れられる．この集合に，アクセスする予定である URL のハッシュ値を含んでいるかどうかを問い合わせることで，重複 URL 検出が行われる．

URL のハッシュ値集合を操作するにあたり問題となるのは，URL の数が膨大なことである．ウェブ上には，数百億の URL が存在することがわかっている[10].

そのため，64 ビットのハッシュ値を用いた場合でも，最終的なハッシュ値集合の容量は 100GB を超えることとなる．100GB を超えるようなハッシュ値集合をメモリ上で扱うには，ウェブクローラを複数台のマシンで並列実行し，マシン毎にクロールするドメインを限定することで，マシン 1 台あたりのメモリ使用量を抑えるという手段が考えられる．しかし，クローリングに使用可能なマシンの台数が限られている場合は，メモリ上のみでハッシュ値操作を行うのは困難である．また，マシンの台数にゆとりがある場合も，重複 URL 検出を少ないメモリ使用量で処理することができれば，100GB 以上のメモリ空間の節約となるため，得られる恩恵は大きい．そのため，本稿では使用可能なメモリが限られている場合でも効率的に動作する重複 URL 検出処理を取り扱う．

少ないメモリ使用量で重複 URL 検出を行うためには，ストレージ上でハッシュ値集合を管理する必要がある．ただし，ストレージ上でのハッシュ値集合操作は，一般的にメモリ上での操作よりも処理コストが大きい．よって，ストレージ上でのハッシュ値集合操作，すなわちストレージアクセスを減らすことが重要となる．

URL のハッシュ値集合操作によって発生するストレージアクセス回数を削減するための解決策として，URL をキャッシュする方法がある[2]．クローリング中に出現する URL をキャッシュしておくと，高い確率でキャッシュヒットすることが知られている．www.yahoo.com などの頻出 URL がキャッシュに載りやすいことや，相互リンクなどの，共起関係にある URL の組み合わせが多いことが，高いキャッシュヒット率の理由とされている．例えば，LRU などの一般的なキャッシング手法を使用することで，約 80%のキャッシュヒット率を実現可能である．そのため，キャッシュをメモリ上に確保することで，ストレージ上にある URL のハッシュ値集合に対する問い合わせ回数と，ハッシュ値集合の更新回数を大幅に減らすことが可能となる．

URL をキャッシュするときに問題となるのは，キャッシュの更新と，要素の包含判定の操作の計算量である．例えば LRU キャッシュの場合は O(log N)の計算量が必要である．そのため，キャッシュサイズを大きくすると，ヒット率は向上するが，キャッシュの操作にかかる計算時間が増加する．その結果，一定のキャッシュサイズを超えると，ストレージアクセス以上の計算時間がかかり，キャッシュを利用する利点がなくなる[6]．しかし，キャッシュの操作コストを O(1)にすることができれば，計算時間をほとんど変えることなくキャッシュサイズを増加させることが可能となり，キャッシュヒット率を向上させることができる．O(1)の計算量でキャッシュ操作を実現するためには，Bloom Filter[3]を利用する近似的な手法が考えられる．Bloom Filter とは，集合の包含判定を行う際に，含まれていない要素を含まれていると誤判定する false positive の発生を許容する代わりに空間効率を改善した乱択(ランダマイズド)データ構造である．

本稿では，新しい乱択データ構造である Stable Bloom Filter Light を提案し，従来から提案されている URL をキャッシュする手法に加えて Stable Bloom Filter Light を用いることで，近似的な重複 URL 検出を行う手法を実現する．提案手法を用いることにより，重複 URL 検出処理に LRU キャッシュのみを用いた場合と比較して，ストレージ上の重複 URL 検出用ハッシュ値へのアクセス回数を削減することができる．

以下，2 節では用語定義を行う．3 節では，キャッシュ，Bloom Filter を利用した重複 URL 検出の関連研究について述べる．4 節では，Stable Bloom Filter Light を用いた重複 URL 検出手法を提案する．5 節では，ウェブグラフを用いたクローリングシミュレーションにより，提案手法を評価する．

2 用語定義

本稿では，false positive と false negative を以下のように定義する．

- false positive (FP)
  未クロールのウェブページをクロール済みと誤って判定した場合
- false negative (FN)
  クロール済みのウェブページを未クロールと誤って判定した場合

3 関連研究

本章では，キャッシュを利用した重複 URL 検出，Bloom Filter を利用した重複 URL 検出の関連研究について述べる．

3.1 キャッシュを利用した重複 URL 検出

2003 年に Broder らによって行われた実験により，LRU や CLOCK をウェブクローラにおける重複 URL 検出に利用することで，高いキャッシュヒット率を実現できることが示された[2]．実験では，サイズが 2^{18}程度のキャッシュを用いることで，80%程度のキャッシュヒット率を実現した．

ストレージを利用した重複 URL 検出を行う前に，キャッシュを用いて重複 URL 検出を行うことで，ストレージアクセスを減らすことが可能となる．ただし，キ

ャッシュの操作にかかる計算量は一般的にキャッシュサイズに依存するため，キャッシュサイズが大きすぎると，ストレージのみを利用した重複 URL 検出よりも遅くなるという実験結果が得られている[6].

3.2 Bloom Filter を利用した重複 URL 検出

Bloom Filter とは，1970 年に Bloom によって提案された，FP を許容する代わりに空間効率を改善した，集合に対する要素の包含判定を近似的に実現する乱択（ランダマイズド）データ構造である[3]. Bloom Filter は長さ m のビット列である．最初に，Bloom Filter のビットはすべて 0 で初期化される. Bloom Filter に要素を挿入するには，k 個のハッシュ関数を用いる．挿入する要素のハッシュ値(mod m)を k 個求め，ハッシュ値に対応する Bloom Filter のビットをすべて 1 にすることで，挿入処理が完了する．ある要素が Bloom Filter に挿入されているかどうかを判定するためには，まず，挿入時と同様に判定する要素の k 個のハッシュ値を計算する．次に，そのハッシュ値に対応するビットを調べ，対応するすべてのビットが 1 であったときのみ，要素が挿入済みであると判断される．図 3.1 は，要素 x, y が挿入された Bloom Filter における要素 z の包含判定である．図では，要素 z のハッシュ値に対応するビットの 1 つが 0 であるため，要素 z は Bloom Filter には含まれないと判断される．

Bloom Filter では FP が発生する．m ビットの Bloom Filter に k 個のハッシュ関数を用いて n 個の要素が既に挿入されているとき，新たな要素の挿入による FP 発生率は

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \quad \cdots (3.1)$$

によって求まる．また，m ビットの Bloom Filter に n 個の要素を挿入することがわかっている場合，最適なハッシュ関数の数 k は

$$k=\frac{m}{n}ln2 \quad \cdots (3.2)$$

で求めることが可能である[3].

3.2.1 Counting Bloom Filter (CBF)

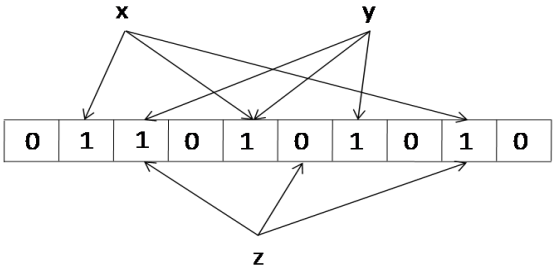

図 3.1 要素 x, y の挿入と z の包含判定

Bloom Filter の特徴として，挿入された要素の削除ができないという点が挙げられる. 2000 年に Fan らによって提案された Counting Bloom Filter (CBF)は，Bloom Filter が使用している 1 ビットのフラグの代わりに，数ビットのカウンタを用いることで，削除機能を実現したデータ構造である[7]. CBF では，要素を追加する際に，k 個のハッシュ値に対応するカウンタをインクリメントする．要素の包含判定は，k 個のハッシュ値に対応するカウンタがすべて 0 でないかどうかを調べることで行われる．そして，要素の削除は，k 個のハッシュ値に対応するカウンタをデクリメントすることで実現可能である．CBF では要素の削除が可能であるが，カウンタに使用するビット数を d とすると，通常の Bloom Filter の d 倍の空間を使用することになる．

3.2.2 ストリームにおける重複検出

2003 年に Metwally らによって，ランドマークウィンドウモデル(landmark window model)とスライドウィンドウモデル(sliding window model)，ジャンピングウィンドウモデル(jumping window model)におけるウィンドウに含まれる要素の包含判定に Bloom Filter を利用する研究が行われた[1].

スライドウィンドウモデルとジャンピングウィンドウモデルでは，古い情報をウィンドウから削除するために CBF を利用している. スライドウィンドウモデルでは要素を削除するための情報を持つ追加領域が必要となる．ジャンピングウィンドウモデルでは，複数個の CBF を合成して利用することで，スライドウィンドウモデルと比較して少ないメモリ使用量でウィンドウサイズを拡大することを可能にした.

ウェブクローラにおいても，クローリングの過程で得られるクローリング対象の URL はストリームであると考えると，Metwally らの手法を用いて重複 URL 検出を行うこともできる.

3.2.3 Stable Bloom Filter (SBF)

Bloom Filter の FP 発生率を低くする別の手法として，2006 年に Deng らが Stable Bloom Filter (SBF)を提案した[5]. SBF は CBF をベースにした手法である．SBF では，FP 発生率を減らすため，古い要素を削除するのではなく，要素の挿入時にランダムにいくつかのカウンタをデクリメントする．この処理により，FP，FN 発生率がある一定の値で安定することが数学的に示されている.

SBF において，要素挿入時にデクリメントするカウンタの最適な個数 p は

$$p = \left\{ \left(1 - FPR^{\frac{1}{k}}\right)^{-\frac{1}{Max}} - 1 \right\}^{-1} \left(\frac{1}{k} - \frac{1}{m}\right)^{-1} \quad \cdots (3.3)$$

として求めることができる．m は SBF で使用するカウンタの個数，k は使用するハッシュ関数の数，FPR は許容する FP 発生率，Max はカウンタの最大値である．式(3.3)で求まる p を，デクリメントするカウンタの個数として使用することで，FN 発生率を最小化することが可能となる．

SBF では，Metwally らの手法と違い，SBF に含まれている要素を管理するための追加領域や，複数の CBF を使用する必要がない．また，SBF に必要なカウンタの数 m は，挿入する予定の要素数に依存しない．そのため，SBF の空間使用量は最悪でも CBF と同等となる．その代償として，要素挿入時のデクリメントに乱数生成が必要なため，計算量は O(1)だが，Metwally らの手法と比較して計算時間が増加する．

4　提案手法

本節では，まず，既存手法の問題点とその改善案を示す．次に，SBF の近似的な実装である Stable Bloom Filter Light (SBFL)を提案する．最後に，SBFL を用いたストレージアクセスの少ない重複 URL 検出手法を提案する．

4.1　既存手法の問題点

以下では，URL のキャッシング手法，Bloom Filter の既存手法の問題点と，それに対する改善案を示す．

4.1.1　キャッシュを利用した重複 URL 検出

URL をキャッシュすることで，重複 URL 検出処理を行うときに 80%のキャッシュヒット率を実現できる[2]．しかし，これは十分大きなキャッシュを用いた場合であり，一般的にヒット率の高いキャッシュでは，要素の挿入・包含判定の処理を O(1)の計算量で実装することは困難である．従って，キャッシュサイズが増加すると計算時間が増加する．また，キャッシュサイズを小さくすれば計算時間を削減することは可能であるが，キャッシュヒット率が低下する．既存手法では，キャッシュヒット率の向上と計算時間の削減を同時に実現することは不可能である．

キャッシュヒット率の向上と計算時間の削減を同時に実現するためには，要素の挿入と包含判定を O(1)の計算量で行う近似的なキャッシュを用いれば良い．これにより，キャッシュサイズを増やしても計算時間が増加しなくなるため，既存手法と比較して少ない計算時間で大きなサイズのキャッシュを操作することが可能となる．同時に，キャッシュサイズを増やすことにより，キャッシュヒット率が向上する．

4.1.2　Bloom Filter を利用した重複 URL 検出

Bloom Filter には，挿入された要素数が増加するほど FP 発生率が大きくなるという問題がある[3]．この問題は，あらかじめ Bloom Filter に挿入される予定の要素数がわかっていれば，挿入予定の要素数に対して十分長いビット列を用いることで回避できる．しかし，ウェブクローリングにおいて重複検出処理の対象となる URL 数は未知である．

挿入される要素数が未知である場合でも一定の FP 発生率を実現するために，CBF を利用する手法が Metwally らによって提案されている[1]. Metwally らの手法では CBF に挿入される要素数に最大値を設けることで，入力される URL の数に依存せず FP 発生率を固定することができる．Metwally らの手法では，一定のルールに従って，CBF へ挿入された要素を削除していく．しかし，削除の操作により FN が発生するという問題が起こる．

ウェブクローラにおいては，FN が発生する問題に対する改善策として，Metwally らの手法の前段に LRU キャッシュを置く手法が考えられる．LRU キャッシュによって頻出 URL の重複検出を使うことにより，頻出 URL に対する FN の発生率を抑えることが可能となる．

4.1.3　Stable Bloom Filter (SBF)

Metwally らの手法では，要素の挿入と包含判定は O(1)で行うことが可能であるが，挿入されている要素を管理するための追加領域が必要となる．SBF は，追加領域を使用せずに，O(1)の計算量で Metwally らの手法と同等の処理を行うことができる[5].

SBF では要素の挿入と同時に，ランダムにカウンタをデクリメントする．しかし，デクリメントするカウンタをランダムに選択するために，多くの乱数を生成する必要がある．乱数の生成には計算時間がかかるため，同じ計算量の Metwally らの手法と比較し，計算時間が増加する．例えば，SBF で用いるカウンタの数を 2^{30}，ハッシュ関数の数を 8，許容する FP 発生率を 0.001%，カウンタの最大値を 3 としたときには，要素を挿入するたびに約 84 個の乱数を生成する必要がある．

要素挿入時の乱数生成によって計算時間が増加する問題を解決するためには，乱数を生成しないという手段が考えられる．具体的には，SBF への要素挿入時にデクリメントされるカウンタを規則的に選択することで，乱数生成の必要性をなくし，高速化することが可能となる．

SBF も，FN が発生する問題に関しては，Metwally

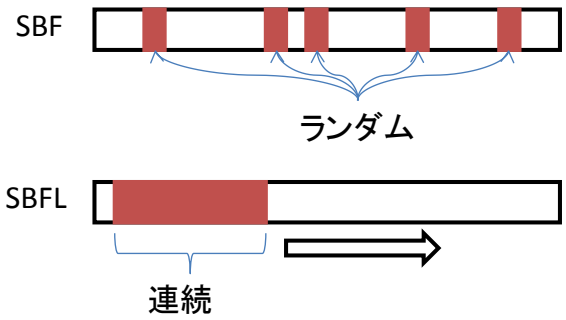

図 4.1 SBF と SBFL におけるデクリメント位置

らの手法と同じである.

4.2 Stable Bloom Filter Light (SBFL)

SBF[5]の近似的な実装である，Stable Bloom Filter Light(SBFL)を提案する．SBFL によって，SBF を利用することにより発生する，乱数生成による速度低下の問題を解決することが可能となる.

以下では，まず，SBFL のアルゴリズムについて説明し，次に SBFL が SBF とほぼ同等の特性を持つことを証明する.

4.2.1 アルゴリズム

SBFL では，SBF において，要素挿入時に発生する乱数生成をなくすことで高速化を図っている．SBF と SBFL の主な違いは，要素挿入時に必要なカウンタのデクリメントに乱数を用いるかどうかである.

SBF では，要素挿入時にランダムに p 個のカウンタを選択しデクリメントすることで，FP 発生率を抑えている．一方，SBFL では，要素挿入時に，シーケンシャルかつ循環的に p 個のカウンタをデクリメントすることで，FP 発生率を抑える(図 4.1)．SBFL では，デクリメントするカウンタは規則的に選択されるため，乱数生成が不要である．そのため，乱数生成にかかる時間の分だけ，SBF よりも高速であると言える.

4.2.2 同等性の証明

SBFL が SBF とほぼ同等の特性を有することを証明する．この事実を証明するためには，挿入された要素のハッシュ値に対応するカウンタがデクリメントされる確率が，SBF と SBFL でほぼ等しいことを示せば良い.

まず，デクリメントされるカウンタを重複ありでランダムに選択した場合と，重複なしでランダムに選択した場合において，個々のカウンタがデクリメントされる確率がほぼ等しいことを示す.

SBF では，要素挿入時にランダムにいくつかのカウンタを選択しデクリメントすることで，FP 発生率を抑えている．デクリメントされるカウンタは重複して選択される可能性がある．このとき，SBF で使用しているカウンタの個数を m，1 度の試行でデクリメントするカウンタの数を p とすると，1 度の試行で個々のカウンタが少なくとも 1 回選択される確率は

$$1-\left(\frac{m-1}{m}\right)^{p} \quad \cdots (4.1)$$

となる.

SBFL では，デクリメントされるカウンタは重複して選択されない．よって，デクリメントされるカウンタがランダムに選択されたとすると，1 度の試行で個々のカウンタがデクリメントされる確率は

$$1-\prod_{k=0}^{p-1}\frac{m-k-1}{m-k} \quad \cdots (4.2)$$

となる．しかし，$p \ll m$ のとき，式(4.2)は式(4.1)とほぼ等しくなる．よって，m が p と比較して十分に大きいときは，デクリメントするカウンタが重複ありで選択される確率と，重複なしで選択される確率はほぼ等しいと言える.

次に，デクリメントされる重複のない p 個のカウンタをどのように選んでも，特定の要素のハッシュ値に対応するカウンタがデクリメントされる確率が変わらないことを示す．このとき，ハッシュ関数によって生成されるハッシュ値は一様に分布していると仮定する.

SBF では，要素の挿入は k 個のハッシュ値によって行われる．そのため，1 つの要素に対応するカウンタは，SBF 上に k 個存在する．ハッシュ値は一様に分布しているため，k 個のカウンタがデクリメントされる確率はそれぞれ等しい．よって簡単のため，1 つのカウンタがデクリメントされる確率を考える.このとき，ある 1 つのハッシュ値に対応するカウンタがデクリメントされる確率は$\frac{p}{m}$である．ハッシュ値が一様に分布しているため，この確率は，p 個のカウンタをランダムに選択しても規則的に選択しても不変である.

以上より，ハッシュ関数によって生成されるハッシュ値は一様に分布しているという仮定の下では，SBFL は SBF とほぼ同等の特性を有することが示された.

4.3 重複 URL 検出手法

次に，LRU キャッシュと SBFL，ストレージ上の 64 ビットハッシュ値集合を用いた，ストレージアクセスの少ない近似的な重複 URL 検出手法を提案する.提案手法により，キャッシュサイズを増加させることによってキャッシュヒット率を向上させること，また，SBFL における FN 発生率を低下させることが可能となる．結果として，冗長なストレージアクセスを削減す

ることができる．

提案手法は，(1)重複 URL 検出，(2)LRU キャッシュと SBFL の更新 2 つの過程にわけて実行される．以下では，それぞれの過程について説明する．

4.3.1 重複 URL 検出

重複 URL 検出では，まず URL が入力される．入力された URL は

1. LRU キャッシュ
2. Stable Bloom Filter Light (SBFL)
3. ストレージ上の 64 ビットハッシュ値集合

の順に重複 URL 検出処理にかけられる(図 4.2)．1 から 3 のいずれかで重複 URL であると判定された場合は，その URL に対してクロールを行わない．

重複 URL ではないと判定された場合は，その URL の 64 ビットハッシュ値が，ストレージ上のハッシュ値集合へ挿入される．

4.3.2 LRU キャッシュと SBFL の更新

提案手法では，重複 URL 検出を行う際に，同時に LRU キャッシュ内の要素の並べ替えと，SBFL への要素の挿入を行う(図 4.3)．

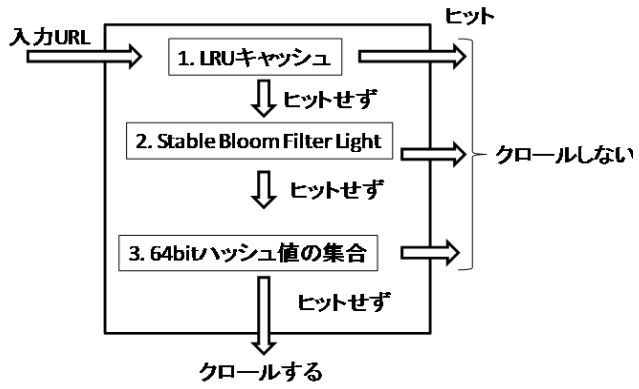

図 4.2 提案手法の構成

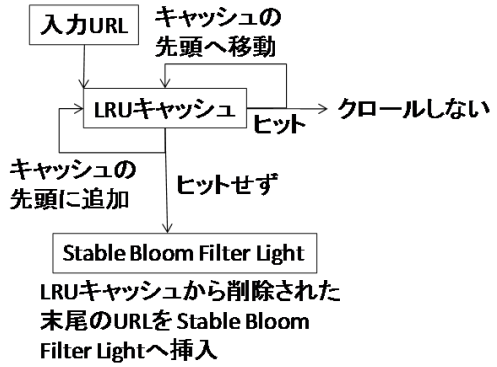

図 4.3 キャッシュの更新

LRU キャッシュで URL がヒットした場合，ヒットした URL を LRU キャッシュの先頭へ移動する．

LRU キャッシュで URL がヒットしなかった場合は，URL をキャッシュの先頭へ追加する．その際に，LRU キャッシュに含まれる要素数が設定値を超えた場合には，LRU キャッシュから末尾の要素が削除され SBFL へ挿入される．

5 実験と評価

本章では，実験結果の評価指標，実験に使用したデータセット，実験方法，実験結果について説明する．

5.1 評価指標

本稿では，ストレージ上のハッシュ値集合へのアクセス回数（キャッシュヒット率）と，メモリ使用量によって，提案手法を評価する．

5.2 データセット

538,137,220 ホストから構成される，ホスト単位のウェブグラフを用いた．ホスト単位のウェブグラフとは，ウェブページ間のリンク関係を，ホスト単位でまとめたものである．

実験に用いたウェブグラフは e-Society プロジェクト[4]によって収集されたウェブデータを元に作成したものである．このウェブグラフに含まれる，アウトリンクを持つユニークなホストの数は 34,848,026 であり，アウトリンク（辺）の総数は 4,155,528,923 である．アウトリンクを持たないホストは，リンクはされているが存在しないホスト，もしくは，未クロールのホストである．

5.3 実験方法

5.2 のウェブグラフを用いて，www.soumu.go.jp を起点として，幅優先型のウェブクローリングシミュレーションを実行し，FP，FN の発生回数，キャッシュアクセス回数，キャッシュヒット回数を計測した．

提案手法のシミュレーションは，LRU キャッシュのサイズを 2^{20} に固定して行った．SBFL のカウンタの個数 m は 2 の n 乗(n = 24,25,…,30)で変化させた．SBFL で使用するハッシュ関数の数は，4 個，もしくは 8 個とした．LRU キャッシュが必要とするメモリ使用量は実装により異なるが，本稿では，LRU キャッシュが必要とするメモリ使用量は，1 要素あたり 20 バイトとする．また，SBFL のカウンタは 1 個あたり 2 ビット使用する．

比較対象として，LRU キャッシュを単体で用いて実験を行った．LRU のサイズは 2 の n 乗(n = 18,19…,24)で変化させた．

5.4　LRU キャッシュの効果

LRU キャッシュ単体を用いてシミュレーションを行った際のキャッシュヒット率を図 5.1 に示す．LRU キャッシュを単体で用いた際には，FP は発生しなかった．

5.5　SBF と SBFL（提案手法）の比較

SBF と SBFL の実行速度と FP，FN の発生回数を比較する．

まず，SBF と SBFL の実行速度を比較する．SBF と SBFL が実装上異なる部分は，要素挿入時のカウンタデクリメントである．その他の部分は全く同じ実装をしているため，カウンタをデクリメントする速度を計測し，比較する．SBF では，乱数生成に Mersenne Twister[8]を利用した．SBFL では，2 ビットのカウンタを個別にデクリメントする方法と，2 ビットのカウンタ 32 個を並列に分岐命令無し飽和デクリメントする方法[9]を使用した．また，実行速度の計測は，Intel Xeon 2.66GHz，16GB のメモリを搭載したマシンで行った．2^{30}個のカウンタ（メモリ使用量 256MB）を 20 億回デクリメントしたときの計測結果を表 5.1 に示す．

計測結果より，SBF のカウンタデクリメントに要する時間は，SBFL の約 34 倍であることがわかる．また，SBFL において，32 並列のデクリメントを行った場合，SBF の約 332 倍の速度で動作することがわかる．

次に，$k = 4$，$FPR = 1\%$としたときの SBF と SBFL における，FP，FN 発生回数の比較結果を図 5.2 と図 5.3 に示す．今回のシミュレーションでは SBFL の方が SBF よりも FP，FN の発生率を低く抑えることができた．これは，デクリメントするカウンタの選択方法の関係上，SBFL が SBF と比較して，LRU や CLOCK に近い特性を備えているためだと考えられる．

5.5.1　LRU+SBFL の効果

SBFL の前に LRU を置いた場合（提案手法）において，$k = 8$, $FPR = 0.001\%$としたときと，$k = 4$, $FPR = 1\%$としたときのキャッシュヒット率を図 5.4 に示す．FP の発生回数は，$k = 8$ のときは 10,000 前後，$k = 4$ のときは図 5.2 の通りであった．

ここで，LRU キャッシュを単体で用いた場合と，提案手法を用いた場合において，同等のキャッシュヒット率のときに，それぞれの手法が必要とするメモリ使用量を比較する．

まず，提案手法において，$k = 8$，$FPR = 0.001\%$のときと，LRU キャッシュを単体で用いたときのキャッシュヒット率を比較した結果を図 5.5 に示す．なお，特定のキャッシュヒット率における，LRU キャッシュのサイズは，キャッシュヒット率を実際に計測したキャッシュサイズ(2 の n 乗)に基づいて線形補間をすることで，およそのサイズを求めた．提案手法で発生した FP は 10,000 前後であった．実験結果より，キャッシュヒット率が低いときには，提案手法を用いる事で使用メモリを削減できることがわかった．ただし，キャッシュヒット率が高くなると LRU を単体で用いた方

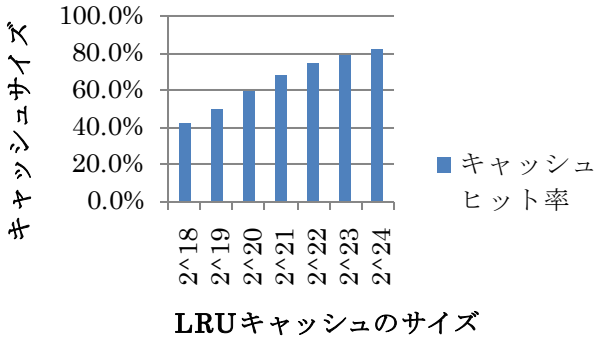

図 5.1 LRU のキャッシュヒット率

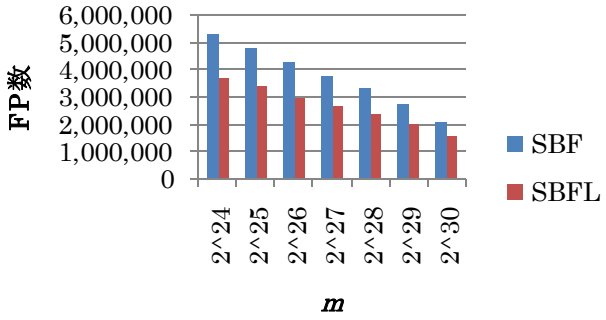

図 5.2 SBF と SBFL の FP 発生回数

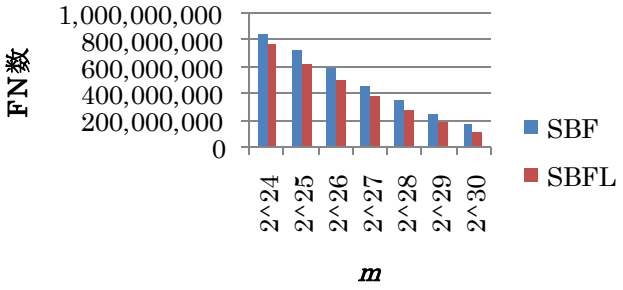

図 5.3 SBF と SBFL の FN 発生回数

表 5.1 SBF と SBFL のデクリメント時間の比較

手法	時間(秒)
SBF	320.7
SBFL	9.435
SBFL（32 並列）	0.9672

がメモリ使用量を抑えることができた．

同様に，k = 4，FPR = 1%としたときの結果を図 5.6 に示す．提案手法において FP 発生率を高めに設定した場合は，LRU キャッシュを単体で用いたときと比較して，メモリ使用量を最大約 63%削減することができた．

6 おわりに

本稿では，Stable Bloom Filter の近似的な実装である Stable Bloom Filter Light を提案し，LRU キャッシュと Stable Bloom Filter Light を用いた，ウェブクローラ向けの効率的な重複 URL 検出手法を提案した．本手法を用いることで，LRU キャッシュを単体で用いたときと比較して，同等のキャッシュヒット率のときにメモリ使用量を約 63%削減することに成功した．

提案手法はいくつかの課題を残している．1 つは，提案手法の実行時間に関する評価である．提案手法では，LRU を単体で用いたときと比較して，使用メモリを削減できることはわかったが，クローリングに用いたときの実行時間に関しては未調査である．

もう 1 つの課題は，FP の扱いである．提案手法では重複 URL 検出時に FP が発生する．このとき，重要なウェブページが FP と判定されることは問題である．そのため，重要なウェブページの FP 発生率を下げるか，クロール後に得られるウェブグラフから，未クロールの重要なウェブページを検出する手法が必要となる．

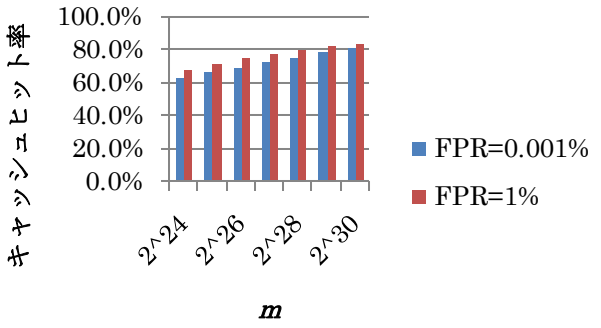

図 5.4 提案手法のキャッシュヒット率

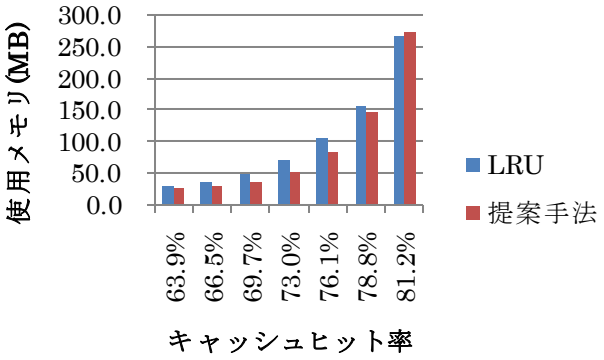

図 5.5 使用メモリの比較(k=8，FPR=0.001%)

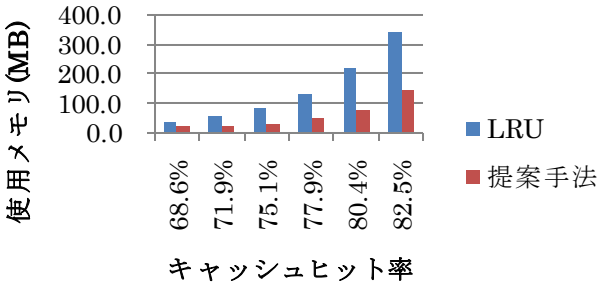

図 5.6 使用メモリの比較(k=4，FPR=1%)

謝　辞

本研究の一部は，科学研究費補助金「情報爆発に対応する高度にスケーラブルなモニタリングアーキテクチャ」によるものである．

参　考　文　献

[1] Ahmed Metwally, Divyakant Agrawal and Amr El Abbadi, “Duplicate Detection in Click Streams”, Proc. of WWW, pp.12-21, 2005.

[2] Andrei Z. Broder, Marc Najork and Janet L. Wiener, “Efficient URL Caching for World Wide Web Crawling”, Proc. of WWW, pp.679-689, 2003.

[3] Burton H. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors”, Communications of the ACM, Volume 13, pp.422-426, 1970.

[4] e-Society プロジェクト, http://www.yama.info.waseda.ac.jp/e-society/.

[5] Fan Deng and Davood Rafiei, “Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters”, Proc. of ACM SIGMOD, pp.25-36, 2006.

[6] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang and Dmitri Loguinov, “IRLbot: scaling to 6 billion pages and beyond”, Proc. of WWW, pp.427-436, 2008.

[7] Li Fan, Pei Cao, Jussara Almeida and Andrei Z. Broder, “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol”, IEEE/ACM Transactions on Networking, Volume 8, No. 3, pp.281-293. 2000.

[8] Makoto Matsumoto, Takuji Nishimura, “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator”, ACM Transactions on Modeling and Computer Simulation(TOMACS), pp.3-30, 1998.

[9] 鶴田真一, “複数のビットフィールドを持つ数値の並列演算”, http://www.emit.jp/prog/prog_b.html, 2000.

[10] 山名早人, “検索エンジンの信頼性”, 人工知能学会誌 Volume 23 No. 6, pp.752-759, 2008.