情報指向ネットワークへの適正と実現可能性を有するCLOCK-Proに基づいたキャッシュ置換方式の提案と評価

大岡　睦[†]　オムスーヨン[†]　阿多　信吾[††]　村田　正幸[†]

† 大阪大学 大学院情報科学研究科〒 565-0871 大阪府吹田市山田丘 1-5
†† 大阪市立大学 大学院工学研究科〒 558-8585 大阪府大阪市住吉区杉本 3-3-138
E-mail: †{a-ooka,suyong,murata}@ist.osaka-u.ac.jp, ††ata@info.eng.osaka-cu.ac.jp

あらまし　情報指向ネットワーク (ICN) におけるルータキャッシング技術の実現のために、本研究では CLOCK-Pro を参考にして ICN への適性とルータハードウェア実装を考慮した低オーバーヘッド性を有するキャッシュ置換手法として CLOCK-Pro Using Switching Hash-table (CUSH) を提案し、キャッシュ困難なアクセスのキャッシュヒットが達成可能であることをシミュレーション評価によって示した。
キーワード　情報指向ネットワーク (ICN)、コンテンツセントリックネットワーク (CCN)，キャッシング、キャッシュ置換方式

A Proposal and Evaluation of Feasible Cache Replacement Policy for ICN based on CLOCK-Pro

Atsushi OOKA[†], Eum SUYONG[†], Shingo ATA[††], and Masayuki MURATA[†]

† Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
†† Graduate School of Engineering, Osaka City University
3-3-138 Sugimoto, Sumiyoshi-ku, Osaka-shi, Osaka 558-8585, Japan
E-mail: †{a-ooka,suyong,murata}@ist.osaka-u.ac.jp, ††ata@info.eng.osaka-cu.ac.jp

Abstract　Information-centric networking (ICN) requires an innovative cache replacement algorithm with performance far superior to simple policies such as FIFO and computational and memory overheads that are low enough to run on ICN router's hardware. We propose CLOCK-Pro Using Switching Hash-tables (CUSH) to satisfies the requirements and evaluated it, which reveals that CUSH can achieve cache hits against the traces that simple conventional algorithms cannot cause any hits.
Key words　Information-Centric Networking, Content-Centric Networking, Caching, Cache Replacement Policy

1. はじめに

現在のネットワークの課題を解決する将来インターネットとして、 Information-Centric Networking (ICN) が注目されている。現在のインターネットはトラフィック量の急増・携帯端末普及によるネットワーク接続機器数の爆発的増加・高度なアプリケーション運用のための品質やセキュリティ要請など、様々な問題に直面している。これら諸問題に個々に対処するのではなく、ネットワークアーキテクチャから根本的な解決を図る試みとして ICN が注目されている。インターネットが端末の在処 (where) に対してアドレスを割り振り、端末同士の通信に焦点を当てる一方で、ICN では現在の利用形態に即して、IP アドレスの代わりにコンテンツごとに name と呼ばれる識別子を割り当て、情報 (what) とユーザを直接結びつける。この発想の転換によって実現されるネットワークアーキテクチャが注目を集め、NDN [1] · CCNx [2] · PURSUIT [3] · SAIL [4] · Green ICN など、世界的に多数のプロジェクトが発足され、研究が盛んに行われている。

ICN は clean-slate なアーキテクチャであり、厳密なプロトコルとそれをサポートするデバイスは策定段階にある。本稿で扱う通信方式および名前付け規則は、NDN や CCNx [1], [2] の基本設計に基づく。NDN ではコンテンツごとに自然言語で書かれた覚えやすいユニークな識別子が割り当てられる。MTU を超えるようなデータサイズのコンテンツはチャンクに分割され、個々のチャンクごとに name が割り当てられる。例えば、YouTube の動画データの第二チャンクは “ccn://YouTube/video/A.mp4/s2” のように表しうる。name アドレスは中継ルータにおけるキャッシングを可能にするが、ルータの限られた資源を有効に活用す

る方策も当然に求められる。

特に、ネットワーク内部のルータにおけるキャッシュは期待が大きい一方で実用的な実装は困難であり、盛んに研究が行われている。ネットワーク内キャッシュの研究は、主としてネットワークトラフィックの高い重複度に動機付けられている。ネットワークトラフィックは Zipf 則に従うことが明らかにされており [5]、多数のパケットが重複した情報を運ぶ冗長な通信が行われている。観測期間によっても値は異なるが、2 回以上アクセスのあるコンテンツがトラフィック量に対して占める割合は 5 割を超えるという結果もある [6]。その可能性の大きさから多くの研究が行われているものの、主にルータがどのコンテンツをキャッシュすべきかというキャッシュ配置・判断戦略の研究に焦点が当てられており [7]〜[11]、キャッシュストレージが溢れた際にどのキャッシュを置換すべきかを決定するキャッシュ置換戦略について、従来手法が ICN への適性を備えているか、ルータハードウェアで実用可能であるかといった観点での検討は不十分である。キャッシュ置換戦略を題材にした先行研究は多量に存在するが、高度なアルゴリズムは高オーバーヘッドであるためルータにおける実装が困難で、単純なアルゴリズムは多様なネットワークトラフィックに対して十分な性能を発揮できない。

本稿では、ルータ内キャッシュの実装課題を解決するために、高性能なキャッシュ置換方式を低オーバーヘッドに実装する方法を提案し、実現可能性と有効性の評価を行った。ルータ内キャッシュの実現には、ルータ単位で実行可能なキャッシュ置換方式を実装する必要がある。本研究では、実装課題としてオーバーヘッドとネットワークトラフィック適性に焦点を当てる。過去の研究では検索機構へのオーバーヘッドを無視した条件の下での手法を研究したが [12]、今回は検索機構へのエントリ追加が不要で、少ないメモリ資源と計算量で実装可能かつ、ワンタイマーコンテンツのキャッシュを回避しつつチャンク単位アクセスにも対処可能なキャッシュ置換戦略として、LIRS/CLOCK-Pro [13], [14] に基づく方式 CLOCK-Pro Using Switching Hash-table (CUSH) を提案する (3. 章)。4. 章ではキャッシュ置換方式のネットワークトラフィックに対する適性および時間・空間計算量の観点からオーバーヘッドを分析し、CUSH が ICN 環境に適応可能な性質を持ちつつ、ハードウェア実装要件を満たすキャッシュ置換方式であることを示す。

2. 関 連 研 究

キャッシュ置換アルゴリズムの既存研究は数多くある。主に CPU や仮想メモリなどの計算機領域のキャッシュ方式とプロキシキャッシュなどのネットワーク領域でのキャッシュ方式 [15]〜[17] に二分されるが、LRU でもルータハードウェアでの実現が困難であることが指摘されており [10], [18]、本稿では計算機領域の低オーバーヘッドなキャッシュ置換方式として CLOCK と同等のオーバーヘッドの方式に焦点を当てる。本章では、オーバーヘッドとキャッシュヒット率の 2 つの観点からキャッシュ方式を分類し、それぞれの戦略と特徴を簡単にまとめる。ただし、以下ではキャッシュサイズを n と表記し、キャッシュする単位はチャンクと呼ぶ。また、CLOCK とそれに基づく手法で用いられている参照ビットは R ビットと表記する。参照ビットとは、キャッシュされたチャンクに最近アクセスがあったかどうかを記憶するためのビットで、アクセスされた場合は R ビットが 1 に設定される。

2.1 低オーバーヘッド・低性能

Random は、すべてのチャンクを一様の確率で削除する手法である。計算機領域における直接マッピング方式と同等であり、低オーバーヘッドに実装可能である。Random の性能は一般的に LRU に劣るが、状況によっては LRU と同等以上の性能を発揮しうる [10]。First-In, First Out (FIFO) では、キャッシュ内で最初にキャッシュされたチャンクが最初に置換される。最も単純な手法の 1 つだが、キャッシュヒット率は一般的に低い。キャッシュされたチャンクを配列上に保持しておき、置換対象の位置 (つまり、FIFO リストの tail) を置換する度に 1 ずつずらすだけで実現できる。

2.2 高オーバーヘッド・高性能

最も代表的な方法の 1 つは LRU チャンクを置換する手法である。この方式は、LRU friendly と呼ばれる、参照の局所性を備えたアクセスパターンに対しては最適に近いキャッシュヒット率を実現できる。LRU の実装方法は複数考えられるが、いずれもハードウェア実装は困難である。例えば LRU stack と呼ばれる、すべてのチャンクをアクセス順にソートしたリストを保持する方法は、置換実行時の処理が低速である。ソートされたリストは予め設定された固定メモリ領域に収められるので、任意のチャンクを削除した後にアクセスされたチャンクをリストの先頭に格納するための空き領域を確保するために、キャッシュアクセスの度にリスト先頭から削除されたチャンクの位置までの膨大な量のチャンクをシフトコピーしなければならない。

Least Frequently Used (LFU) は、チャンクの参照回数 (frequency) を記憶しておき、最も参照回数の小さいものを削除する手法である。使い古されたチャンクによってキャッシュが圧迫されるため、キャッシュヒット率は高くない。その上、参照回数に基づく優先度付きキューを維持するためのオーバーヘッドが大きすぎるため、ハードウェア実装は困難である。

Adaptive Replacement Cache (ARC) [19] は LRU と LFU の長所を兼ね揃えており、より多様なアクセスに対して適性を持つ。ARC は 2 つの LRU リスト L_1 と L_2 を保持し、それぞれ 1 回だけアクセスされたチャンク、2 回以上アクセスされたチャンクを保持する。また、キャッシュ履歴 (ghost cache)、つまり、実際のキャッシュデータは保持せずに履歴情報だけを保持することで、様々なアクセスに対応するできる。

Low Inter-reference Recency Set (LIRS) [13] は、再利用間隔 (Inter-Reference Recency; IRR) に基いてチャンクを分類する。LRU が最後のアクセスから現時点までのアクセス数 (recency) を用いるのに対して、LIRS で用いられる IRR は最後から 2 回目のアクセスから最後のアクセスまでのアクセス数として定義される。IRR が小さい一定の割合のチャンクは hot、そうでないチャンクは cold と呼ばれ、cold チャンクの中で LRU なものから順に削除する。また、cold チャンクの一部は履歴情報として保持する。この戦略によって、LRU や ARC が不得意とする

loop と呼ばれるアクセスパターン (3.1.1 項で後述) への耐性を実現している。しかし、LFU ほどではないものの、hot チャンクの陳腐化への対応が遅く、頻繁な人気の変動には弱い傾向がある。更に、可変長 (実用上は $4n$ 程度のサイズ [13]) の LRU リストが必要で、オーバーヘッドが LRU や ARC よりも大きい。

2.3 低オーバーヘッド・高性能

CLOCK は LRU の優れた近似手法である。CLOCK は各チャンクに R ビットを割り当てる。ヒットした場合は $R = 1$ に設定される。置換が必要な場合、$R = 0$ のチャンクが優先的に削除される。置換するチャンクの探索は、前回の削除チャンクの位置から開始してキャッシュリストを円環状に探索する。円環状のキャッシュリストとそれを巡る探索位置を、時計とその針に見立て、CLOCK と呼ばれる。CLOCK はこのように低オーバーヘッドであるにも関わらず、LRU と同等の性能を発揮することができる。

Clock with Adaptive Replacement (CAR) [20] は ARC に CLOCK を適用したキャッシング手法である。LRU リストを CLOCK に変更することで、ヒット時の MRU operation を不要にすることで計算量オーバーヘッドを削減する。その上、性能は ARC に匹敵する。しかし、可変長の CLOCK として実装されているので、キャッシュミス時にはキャッシュリストの任意位置への挿入や削除が必要で、LRU と同等のオーバーヘッドを要する。この解決には、[12] のような手法が必要となるが、ここではキャッシュ履歴実装による検索テーブルへの影響を考慮していない。

CLOCK-Pro [14] は LIRS を CLOCK を用いて近似した手法である。R ビットに加えて、チャンクごとに 2 つのビットを追加する。1 つはチャンクが hot か cold かを記憶するビットである。もう 1 つは test flag であり、これは LIRS の stack pruning という古い cold チャンク (履歴情報含む) を削除する処理を再現するために用いられる。これらのビットを追加した単一の CLOCK に対して 3 つの針を用いて、それぞれ cold チャンク、hot チャンク、履歴情報の削除処理を実現する。LIRS では固定パラメータだった hot チャンクと cold チャンクの割合を適応的に変化させる仕組みを持っているため、LRU-friendly なトラフィックにも適応しやすい。しかし、保持可能な履歴情報数が有限 (最大で n 個) であるため、loop への耐性は LIRS に劣る。

3. 提案方式の設計

本章では、提案方式 CUSH の設計思想と具体的なデータ構造とアルゴリズムを説明する。ICN ルータにおけるキャッシングでは、ワンタイマーアクセスやチャンク単位アクセスなど、ネットワーク特有のアクセスを考慮する必要がある。そこで、これらの特徴的アクセスに対して高いヒット率を達成可能なキャッシュ置換戦略について計算機分野の知見を用いて議論し、CLOCK-Pro の戦略の有効性を明らかにする (3.1 節)。提案方式 CUSH は、この戦略の有用性を拡大しつつ、計算機分野には見られなかった検索機構の管理コストを含む実装課題に対処する。そのために、CUSH はキャッシュ履歴機構を低オーバーヘッドに拡張可能なデータ構造を持つ戦略を採る (3.2 節)。その具体的なアルゴリズムは 3.3 節で詳細に説明する。結果として、CUSH は CLOCK-Pro と同等以下の時間・空間計算量で済み、ネットワークトラフィックに適応可能なキャッシュ置換を実現する。

3.1 ネットワークトラフィックとキャッシュ置換方式の関係

キャッシュ置換戦略の研究過程で、キャッシュ置換の性能を悪化させるアクセス系列が明らかにされてきた。そのような特徴的なアクセス系列はアクセスパターンと呼ばれる。本節では、まずアクセスパターンについて説明し、キャッシュ置換アルゴリズムごとのアクセスパターンへの適性をまとめる (3.1.1 項)。その後、ネットワークトラフィックによって scan や loop と呼ばれるアクセスパターンが形成されることを示し、CLOCK-Pro が ICN ルータでの運用に適したキャッシュ置換戦略であることを明らかにする (3.1.2 項)。

3.1.1 アクセスパターン

ここではキャッシュヒット率を悪化させる 4 つのアクセスパターンである scan・loop・correlated-reference・fickle-interest について説明する。特定のアプリケーションにのみ見られる特徴的なアクセスを除けば、あらゆるアクセスはこの 4 つのいずれかに分類される。これらの 1 つまたは複数のアクセスパターンに対する耐性を実現することがキャッシュ置換方式の目標となる。

scan: 1 度しかアクセスされないページの連続的な読み込み。一度もキャッシュヒットしないチャンクによってキャッシュされているチャンクすべてが置換されるキャッシュ汚染によって、LRU などの recency に基づく戦略の性能を大きく悪化させる。

loop: キャッシュサイズを超える長さの scan の繰り返し。チャンクがキャッシュされても、次のアクセスまでにキャッシュから溢れてしまい、キャッシュヒットが発生しない。すべてのチャンクの recency と frequency が等しいため、それらに基づく方式 (LRU・LFU・ARC など) ではキャッシュヒットが発生しなくなる。

correlated-reference: 1 つのページに対するアクセスが短期間に集中するようなアクセスパターン。複数回アクセスされてキャッシュヒットとなるが、集中アクセスが終わるとそれ以降全くアクセスされなくなるという特徴を持つ。LFU などの frequency に基づく手法は、長期的なキャッシングが不要なチャンクに高い優先度を付与してしまい、キャッシュする価値のないチャンクが長期間キャッシュに留まるという問題が発生する。

fickle-interest: アクセスされるチャンク集合が頻繁に切り替わるアクセスパターンで、異なる correlated-reference の繰り返しとも見なせる。frequency に基づく LFU だけでなく、LIRS の hot チャンクのように、チャンクの一部を優遇して長期的に保持する手法では、アクセス列の切り替わりの際に新しいアクセス列への対応が遅れてしまい、キャッシュヒット率が低下する。

アクセスパターンとキャッシュ置換方式の関係は表 1 のように表せる。ARC は 2 回以上のアクセスは等価に扱うため、priority は limited-frequency としている。その結果、correlated-reference の悪影響を回避可能であり、LFU の上位互換としての性能を獲得している。しかし、loop に対処できないという LRU や LFU

表 1: アクセスパターンによるキャッシュ置換方式への影響

policies	LRU, CLOCK	LFU	ARC, CAR	LIRS	CLOCK-Pro
priority	recency	frequency	limited-frequency	IRR	limited-IRR
scan	キャッシュヒットしない	影響を受けない	影響を受けない	影響を受けない	影響を受けない
loop	キャッシュヒットしない	キャッシュヒットしない	キャッシュヒットしない	影響を受けない	長い **loop** にはキャッシュヒットしない
correlated-reference	影響を受けない	性能が大きく下がる	影響を受けない	影響はあるが小さい	影響はあるが小さい
fickle-interest	影響を受けない	性能が大きく下がる	影響を受けない	性能が大きく下がる	性能が下がりやすい

の欠点は解決できていない。CLOCK-Pro については履歴長が有限であるため、priority は limited-IRR としている。そのため、loop への耐性は LIRS より低い。また、CLOCK-Pro は LIRS の戦略に加えて、hot チャンク数の調整アルゴリズムを備えるため、correlated-reference や fickle-interest に強い。しかし、調整アルゴリズムが十分に適応的でないために、人気度の変動が頻繁な場合には性能を発揮できない場合がある。

3.1.2 ネットワークトラフィックに適したキャッシュ置換戦略

本項では、前項で議論したアクセスパターンの特徴に基づいて、各アクセスパターンがどのようなネットワーク利用によって発生するかを分析する。本稿ではチャンク単位アクセスに注目し、ネットワーク内キャッシュで特に注目すべきアクセスパターンが scan と loop であることを述べる。そして、そのアクセスパターンに対処可能な CLOCK-Pro が、ネットワーク内キャッシュにおいても高ヒット率を達成しうる戦略を持つことを明らかにする。

ネットワークトラフィックはワンタイマーコンテンツを多く含むため、scan の発生頻度が高い。特に、ネットワークは計算機と異なり多数のユーザが同時並列的に利用する。多数同時アクセスが発生した場合、互いに無関係なワンタイマーコンテンツ要求は scan を頻繁に形成しうる。実際、ネットワークトラフィックの観測データの中には、1 回しかアクセスのないコンテンツがトラフィック量の 6 割を占めるという結果もある [6]。更に、CCNx や NDN など代表的な ICN アーキテクチャでは、下位層を考慮してコンテンツをチャンクに分割する。このチャンク分割によって、単一コンテンツへのアクセスが scan を形成しうる。したがって、scan の生成頻度は非常に高いと見積もられる。

複合コンテンツや、細かい粒度のチャンクアクセスは loop を生じうる。Web ページや OS など大規模ソフトウェアのアップデートでは、1 つのコンテンツが多数の構成要素から形成されうる。言い換えれば、ユーザにとって単一のサービスでも、複数のコンテンツの同時要求が必要な状況がある。それが人気のあるサービスである場合、複数のユーザからの複合コンテンツの連続的要求は loop を形成しうる。また、チャンク単位アクセスを考慮すると、単にコンテンツのサイズが大きいというだけで loop が発生しうる。例えば、100MB の動画コンテンツはルータのキャッシュに対して約 0.7M 回のアクセスを発生させる。したがって、loop の存在もネットワーク内キャッシュでは考慮すべきである。

一時的にアクセスが集中するような一部のコンテンツやネットワークアプリケーションのために、correlated-reference や fickle-interest への耐性も無視はできない。ニュースサイトや SNS などのリアルタイム性の高いコンテンツは correlated-reference になりやすいだろう。また、ネットワークでは、計算機領域ではありえないような多数ユーザからのアクセスが発生するため、単にコンテンツ人気度の移り変わりだけでなく、ネットワークにアクセスするユーザの移り変わりによっても要求されるコンテンツ傾向が変化する。加えて、VoIP やライブ放送などをキャッシュを用いてマルチキャストする場合は、多数のユーザが極短時間だけそのコンテンツにアクセスして、それ以降は一切アクセスされない。この場合は、fickle-interest の問題が顕著に現れるだろう。

したがって、ネットワークトラフィックのキャッシュで有用な置換戦略は、LIRS/CLOCK-Pro である。ネットワークでは scan が頻繁に出現し、チャンク単位アクセスでは loop が生じやすい。限定的な条件下で、correlated-reference や fickle-interest も考慮する必要がある。それに対して、CLOCK-Pro は scan に強く、限定的ながら loop 耐性も併せ持つ。Compact CAR の研究ではワンタイマーアクセスに起因する scan・fickle-interest に焦点を当てた [12]。今回はチャンク単位のアクセスに焦点を当て、scan のみならず loop にも対処する。一方で、CLOCK-Pro の欠点として、loop および fickle-interest 耐性を欠く点や、計算量オーバーヘッドの課題が挙げられる。次節で、この課題の解決方法を明らかにする。

3.2 CUSH の設計

3.1.2 で述べた通り、本稿では CLOCK-Pro に着目する。CLOCK-Pro はネットワークトラフィック適正と低オーバーヘッドという特徴を併せ持つが、CLOCK-Pro の ICN ルータでの運用には課題がある。本稿では、これらの課題を解決するためにキャッシュ履歴のデータ構造を修正した提案方式 CUSH を提案する。CUSH は低オーバーヘッドなキャッシュ履歴の拡張を主とする工夫によって、CLOCK-Pro の有する課題を解決することができる。

3.2.1 LIRS/CLOCK-Pro の特徴

CLOCK-Pro のオリジナルである LIRS は、IRR に基づくチャンク分類によって loop 耐性を実現する。loop に対応するためには、極めて長いアクセス列の中から同一チャンクへのアクセスを検出しなければならない。LRU や CLOCK のように recency(直近のアクセスから現在までの間隔) だけを用いる戦略の欠点は、長い間チャンクのアクセス情報を保持できないことである。繰り返されるアクセス列長が長い場合、特定のチャンクに対する 2 回目のアクセスが行われる前に、最初のアクセスでキャッシュされたチャンクが削除される。一方、LIRS では loop に対処す

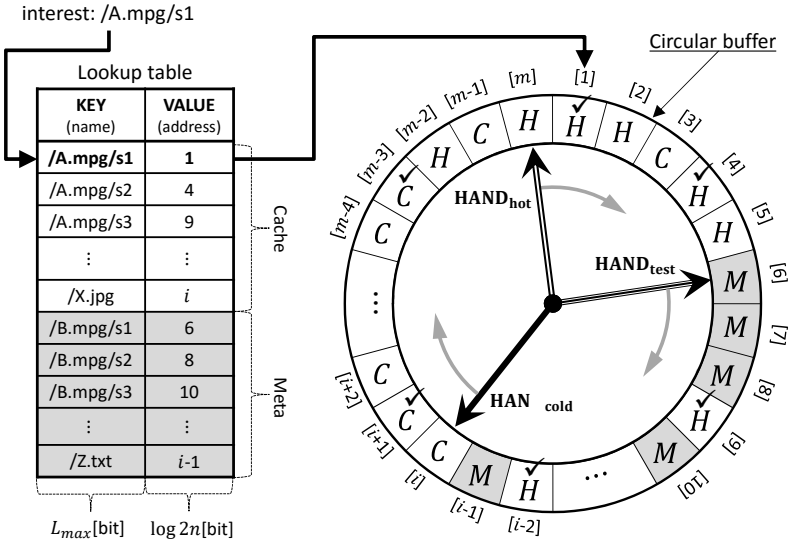


図 1: 理想的な検索テーブルを伴うキャッシュ履歴を持つ CLOCK-Pro の概要図

るために、IRR(直近の連続する 2 回のアクセスの間隔) を用いる。キャッシュ履歴によって理論上無限にアクセス情報を保持することで、繰り返し間隔が長い loop でもその繰り返しを検出できる。そして、IRR が小さい順にチャンクを分類し、キャッシュに保持できるだけの量を hot チャンクとして優先的に保持し、それ以外は cold チャンクとして削除する。この戦略によって、LIRS は loop への耐性を獲得している。scan 中のチャンクのようなワンタイマーアクセスは IRR が無限大として定義され、cold チャンクとして優先的に削除されるため、scan 耐性も備えている。

CLOCK-Pro は、LIRS の特徴を引き継ぎつつ、CLOCK を用いて低オーバーヘッド化された方式である。LIRS の IRR の管理機構が LRU で実装されているだけでなく、キャッシュ履歴は理論上無限長である。更に、古くなったキャッシュ履歴を削除する stack pruning と呼ばれる処理は、削除されるキャッシュ履歴の数に応じて計算オーバーヘッドが肥大化してしまう。この問題を解決するために、CLOCK-Pro は LRU の代わりに CLOCK を用いる。また、キャッシュ履歴サイズは有限長であり、実キャッシュと同量の履歴を保持する。キャッシュ履歴長の制限によって loop 耐性は限定的となるが、キャッシュ履歴によって追加されるオーバーヘッドを抑えられる。

CLOCK-Pro は、図 1 のような 3 本の針を持ち、エントリごとに 4 ビットのフラグを割り当てる CLOCK の拡張として実現される。3 つの針は $\mathrm{HAND_{cold}}$、$\mathrm{HAND_{hot}}$、$\mathrm{HAND_{test}}$ である。ビットフラグは、R ビットの他に、hot/cold チャンクを区別するフラグと、キャッシュ履歴保持期間を定める $\mathrm{HAND_{test}}$ 用の test フラグ、およびキャッシュが履歴かそうでないかを区別するフラグである。1 では、cold チャンクを C、hot チャンクを H、メタ情報のみを持つキャッシュ履歴を M と表記している。影をつけたエントリはキャッシュ履歴用であることを意味する。また、チェックされているチャンクは $R = 1$ のチャンクである。

CLOCK-Pro の各針の具体的な動作は、以下のように要約される。まず、チャンクの置換は $\mathrm{HAND_{cold}}$ によって実行され、cold チャンクが優先的に削除される。hot チャンクと履歴を無視することを除けば、これは CLOCK の針と同じ役割を持つ。削除されたチャンクは履歴となる。履歴ヒットしたチャンク、もしくは単に $R = 1$ の cold チャンクは、hot チャンクとしてキャッシュされる。一度削除されて履歴だけになったチャンクでも、この仕組みによって hot チャンクとして分類され、優先的に保持される。このとき、$\mathrm{HAND_{hot}}$ によって、古い hot チャンクが cold チャンクに格下げされると同時に、古い cold チャンクの test を 0 にする。test フラグが 0 のチャンクは、履歴にせず即座に削除する。なぜなら、古い履歴を残しておくと、ヒットしてしまった場合に hot チャンク汚染を発生させるためである。すでに履歴化されていた cold チャンクの場合、test フラグが切れた時点で削除される。また、チャンクが古い場合だけでなく、履歴数が多すぎる場合にも削除しなければならない。この削除処理を担当するのは $\mathrm{HAND_{test}}$ である。cold チャンク削除時に履歴チャンクが発生し、履歴が閾値を超えた場合は、$\mathrm{HAND_{test}}$ によってチャンク削除を行う。閾値はアクセス系列に対して適応的に調節され、履歴が有効なアクセスに対しては履歴数を増加させ、そうでない場合は減少させる。以上の処理によって、LIRS と CLOCK の利点を両立した戦略を低オーバーヘッドに実現することを可能にしている。

3.2.2 CLOCK-Pro の課題

本項では、CLOCK-Pro を ICN ルータで運用する際の 3 つの課題について詳述する。第一の課題は不十分な loop 耐性である。loop への耐性は履歴情報の大きさによって決まる。CLOCK-Pro はキャッシュサイズと同じだけの履歴情報を保持するため、キャッシュサイズの 2 倍までの長さの loop にしか対応できない。特に、ルータのキャッシュ容量が限られた状況下では、対応できる loop 長も必然的に小さくなる。しかし、CLOCK-Pro では単純に履歴情報を多くすると円環バッファが肥大化し、置換処理に伴う針の回転数の増加につながる。

第二の課題はキャッシュ履歴の管理に必要な針の動作である。図 1 に示すように、CLOCK-Pro はキャッシュ履歴を円環バッ

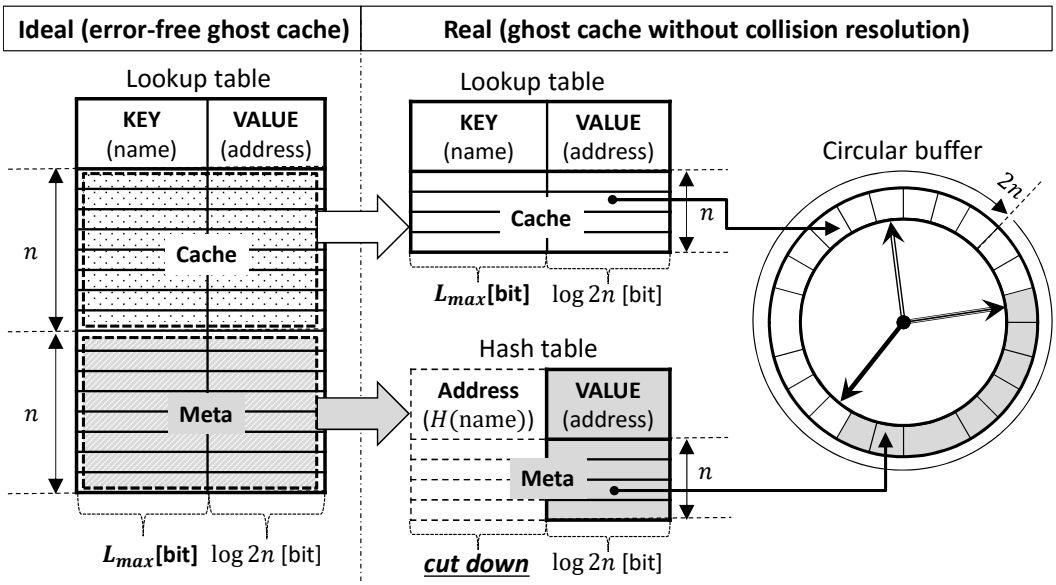


図 2: CLOCK-Pro におけるキャッシュ履歴の現実的な実装の概要図

ファ中に取り込んでいる。特に $\mathrm{HAND}_{\mathrm{cold}}$ の回転ではメタ情報エントリを無視するため、$\mathrm{HAND}_{\mathrm{cold}}$ の全回転処理中の少なくとも半数が無意味に費やされる。loop 耐性向上のためにメタ情報エントリ数を増やせば、増やした数に比例して無意味な回転処理の数が増加してしまう。

第三の課題はキャッシュ履歴導入のための検索機構のオーバーヘッドである。オーバーヘッドの大きさを説明するために、図 1 のような Key-Value ストアとして実装された検索機構を用いた理想的なキャッシュ機構について考える。CLOCK-Pro は、キャッシュサイズを n とすると、$2n$ 個のエントリを持つ検索テーブル (lookup table) と円環バッファ(circular buffer) から構成されている。検索テーブル中の各エントリは、name 保持のための L_{max}[bit](name の最大長) と、円環バッファのアドレスを記憶するための $\log 2n$[bit] の記憶領域を持つ。検索テーブル全体では $2n(L_{max}+\log 2n)$[bit] の記憶容量が必要で、キャッシュ履歴の実装のために $n(L_{max}+\log 2n)$[bit] が割り当てられている。

検索機構に対するオーバーヘッドは、L_{max}[bit] の値に依存するとはいえ、キャッシュ履歴エントリのために多大なオーバーヘッドが要請されることは明らかである。当然ながら、name の木構造を利用するなどしてメモリオーバーヘッドを大きく削減する検索機構は研究されている。しかし、検索機構は ICN ルータ上の乏しい資源で実装されており、また、ネットワークトラフィックを実用時間内で処理しなければならない。この資源と速度の制約に曝された検索機構に対するエントリ追加・管理のオーバーヘッドが大きいことは、図 1 のような単純かつ直感的な実装を用いるまでもなく明らかだろう。計算機領域と異なり、処理速度と容量共に余裕のない ICN ルータの検索機構に負担をかけないためにも、キャッシュ履歴の識別子が検索機構に課すオーバーヘッドを最小化する方法を考えるべきである。

キャッシュ履歴の検索機構に対するオーバーヘッドを考慮すると、name の保持が不要な実装方法が必要である。例えば、図 2 のような実装が考えられる。衝突を避けるためには、図の左部のように、必ず name を保持する必要がある。そのためには 1 エントリあたり name の最大長 L_{max}[bit] を確保する必要があり、オーバーヘッドが大きい。しかし、衝突を許容するならば、図の右部のように、name を保持する必要がなくなる。したがって、ICN ルータにおけるキャッシュ履歴の実装は、衝突を許容した検索機構を基準に考えるべきである。次項で説明するように、CUSH は、この構造を洗練して、更に低オーバーヘッドな拡張が可能なアーキテクチャを持つ。

3.2.3 CUSH のデータ構造と概要

3.2.1 項で説明した CLOCK-Pro の特徴を引き継ぎつつ、3.2.2 項の課題を解決した CUSH の概要は図 3 のようになる。図 3 左部は、図 2 右部と対応しており、検索テーブル・衝突許容テーブル・円環バッファから構成される現実的な CLOCK-Pro を示している。提案方式では、この CLOCK-Pro を実キャッシュ領域とキャッシュ履歴領域で分離し、キャッシュ履歴は 2 つの衝突許容ハッシュテーブルを用いて実装する (図 3 右部)。まず、キャッシュ履歴の保持は 1bit のフラグのみで管理できるため、追加オーバーヘッドは極めて小さい。次に、キャッシュ履歴用のハッシュテーブルの大きさを k 倍に変更することで、容易に loop への耐性を向上させることができる。更に、キャッシュ履歴を伸長したとしても、1 エントリあたりのコストは数ビットしかないためメモリオーバーヘッドは小さい。また、キャッシュ履歴が円環バッファから取り外されているため、計算オーバーヘッドの増大も防げる。

CUSH は衝突許容ハッシュテーブルを、LIRS の挙動に基づいた間隔でクリアすることで loop 耐性を実現する。そもそも、LIRS は最古の hot チャンクより古い cold チャンクを削除することで loop 耐性を実現している。最古の hot チャンクより古いチャンクは、キャッシュ可能な量を超えて hot チャンクを保持しようとして hot チャンク汚染を引き起こし、loop 耐性を脅かす。そのため、LIRS はキャッシュ履歴を保持する期間を管理

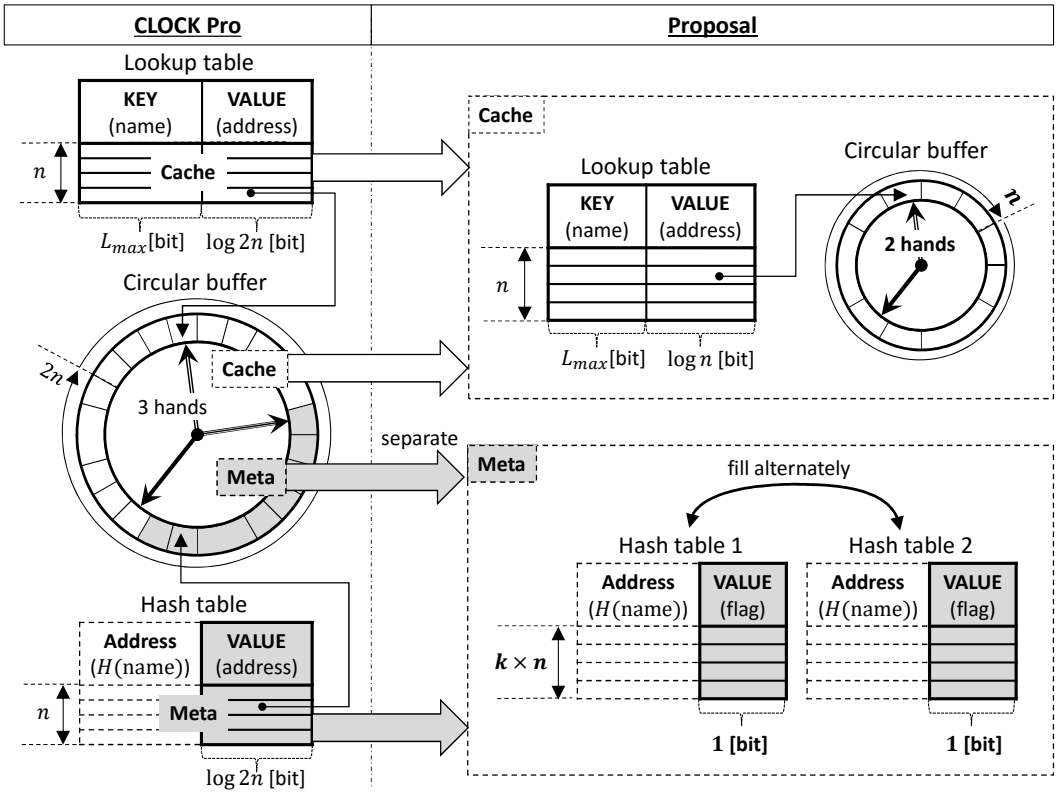


図 3: CLOCK-Pro のキャッシュ履歴を低オーバーヘッドに拡張可能とする提案方式の概念図

している。アルゴリズム上では単純な LRU によって保持期間を管理するが、動作上は hot チャンク数と同じ回数だけキャッシュヒットが発生する度に履歴情報が一新される。CUSH はこの挙動に基づき、キャッシュヒット回数が hot チャンク数 n_h を上回った場合にキャッシュ履歴をクリアすることで、個々のエントリ管理を不要にしながらも loop 耐性を備えたキャッシュ履歴を実現する。

更に、一度の削除処理によるキャッシュ履歴の全削除を防ぐために、2 つのハッシュテーブルを交互に初期化する。一度に全履歴を削除すると、correlated-reference のような一時的なキャッシュヒットによって hot チャンクが占有されうる。そのため、CUSH では 2 つのハッシュテーブルを交互に用いる。2 つのハッシュテーブルを B_1、B_2 として、B_1 からキャッシュ履歴を格納し始めたとすると、$n_h/2$ 回のヒットが発生した時点で、キャッシュ履歴の格納先を B_2 に切り替える。そして、更に $n_h/2$ 回のヒットが発生した時点で、B_1 に格納されたキャッシュ履歴をすべて削除し、キャッシュ履歴の格納先を B_1 に戻す。この処理を繰り返すことで、履歴の全削除を防ぎつつ、loop 耐性を備えたキャッシュ履歴を実現できる。ハッシュテーブルの検索時には B_1 と B_2 の両方の同時検索が必要だが、ハードウェアによるサポートで容易に並列検索を実装可能であり、処理速度の問題は解決できる。

CUSH は更にいくつかの有用な特徴を併せ持つ。第一に、円環バッファからキャッシュ履歴が取り除かれたため、キャッシュ履歴数を一定に維持するための $\mathrm{HAND}_{\mathrm{test}}$ の処理やメタ情報用の針が不要で、2 本の針とエントリごとに 2bit のメモリがあればよい。第二に、円環バッファの全長が短縮されたため、$\mathrm{HAND}_{\mathrm{cold}}$ および $\mathrm{HAND}_{\mathrm{hot}}$ の回転数の増大を防ぐどころか削減に成功している。第三に、ハッシュテーブルを交互に埋めていき、切替時にハッシュテーブル全体の削除を可能とする工夫によって、キャッシュ履歴用のメタ情報の保持・管理を完全に不要としている。この特徴と関連して、第四に、ハッシュテーブルは柔軟な実装が可能である。例えば、図 3 ではハッシュテーブルは 1bit の情報しか持たないが、衝突確率を下げるために 2bit 以上の情報を持つこともできる。

3.3 アルゴリズム

提案方式 CUSH のアルゴリズムの擬似コードをアルゴリズム 1,2,3,4 に示す。ただし、CUSH の持つ CLOCK のチャンクリストを A として、円環バッファ中の位置 p のチャンクは $A[p]$ と表す。c はキャッシュサイズ (単位はチャンク数)、n, n_h, n_c はそれぞれ現在の全チャンク数・cold チャンク数・hot チャンク数、m_h, m_c はそれぞれ現在の cold チャンクと hot チャンクの目標数とする。また、2 つのハッシュテーブルは B_1, B_2 と表記し、現在注目中のハッシュテーブルは B_* と表す。c_B は各ハッシュテーブルの格納可能なチャンク数、n_{B1}, n_{B2} はそれぞれ B_1 と B_2 が格納しているチャンク数である。n_{hit} はヒットカウント数を表す。x はアクセスされたチャンクとする。格納されているチャンク $chunk$ の情報で、$chunk.\mathrm{R}-\mathrm{bit}$ は reference bit を意味し、1 のときアクセスされたことを表す。$chunk.\mathrm{H}-\mathrm{bit}$ は hot チャンクであるか否かを判断するためのビットで、1 のときそのチャンクが hot チャンクであることを意味する。

アルゴリズム 1 は、キャッシュヒット・ミスに応じたキャッ

Algorithm 1 CUSH Replacement Algorithm

```
 1: procedure CacheReplacement(x)                    ▷ x is an accessed chunk.
 2:     if x ∈ A then                                                ▷ cache hit
 3:         x.R-bit← 1
 4:         UpdateHistory()
 5:         AdaptSmallIRR()
 6:         return
 7:     else if x ∈ B_* then                                         ▷ ghost hit
 8:         UpdateHistory()
 9:         AdaptSmallIRR()
10:         h ←true
11:         if n = m then                                            ▷ A is full.
12:             Run HAND_cold
13:             if n_h > m_h then
14:                 Run HAND_hot
15:             end if
16:         end if
17:     else                                                         ▷ cache miss
18:         if n < m then                                            ▷ A is not full.
19:             h ← (n_h < m_h)
20:         else                                                     ▷ A is full.
21:             Run HAND_cold
22:             h ← (n_h < m_h  &  2n_c > m_h)
23:         end if
24:     end if
25:     p ← an available address in A
26:     A[p] ← x
27:     if h then A[p].H-bit← 1
28:     end if
29: end procedure
```

Algorithm 2 Algorithm for Adapting Parameters

```
1: procedure AdaptLargeIRR
2:     m_c ← max(m_c − max(m_h/m_c, 1), 1)
3:     m_h ← m − m_c
4: end procedure
5:
6: procedure AdaptSmallIRR
7:     m_h ← max(m_h − max(m_c/(m_h + 1), 1), 0)
8:     m_c ← m − m_h
9: end procedure
```

シュ置換処理を定義している。キャッシュヒットの場合、アクセスチャンク x の R ビットを 1 に設定した後、キャッシュヒット回数とチャンク目標数の更新処理を行って処理を終了する (行 3–6)。キャッシュミスの場合、履歴ヒットしたか否かによって処理が更に分かれる。履歴ヒットした場合 (行 7–15)、ヒット時と同様に、キャッシュヒット回数とチャンク目標数の更新処理を行う (行 8–9)。次に、x を hot チャンクに設定するフラグ h を 1 に設定しておく (行 10)。そして、キャッシュが一杯ならば $\mathrm{HAND}_{\mathrm{cold}}$ による置換処理を実行し、更に、hot チャンク数 n_h が目標数 m_h を超えている場合は $\mathrm{HAND}_{\mathrm{hot}}$ による hot チャンクの降格処理を実行する (行 11–16)。キャッシュミスした場合 (行 17–23)、キャッシュに空きがあるか否かで処理が分かれる。キャッシュに空きがある場合 (行 18–19)、hot チャンク数 n_h が目標数 m_h 以下ならば hot チャンクとしてキャッシュする。ヒットしていなくても hot チャンクとしてキャッシュするのは、loop の最初の繰り返しでその系列を保持できるようにするためである。キャッシュに空きがない場合 (行 20–22)、$\mathrm{HAND}_{\mathrm{cold}}$ による置換処理を実行する。こちらの場合はキャッシュに空きがある場合よりも hot チャンクとしてキャッシュする条件が厳し

Algorithm 3 Algorithms of Hand Movement

```
 1: procedure Run HAND_cold
 2:     while (A[HAND_cold].H-bit= 0 or A[HAND_cold].R-bit= 1) do
 3:         while n_c = 0 do
 4:             Run HAND_hot
 5:         end while
 6:         if A[HAND_cold].R-bit= 1 then
 7:             A[HAND_cold].R-bit← 0
 8:             A[HAND_cold].H-bit← 1
 9:         end if
10:         Move HAND_coldforward
11:     end while
12:     Discard A[HAND_cold] and add it to B_*
13:     if n_b1 = c_B then
14:         SwitchHashTable()
15:     end if
16:     Move HAND_coldforward
17: end procedure
18:
19: procedure Run HAND_hot
20:     while (A[HAND_hot].H-bit= 0 or A[HAND_hot].R-bit= 1) do
21:         if A[HAND_hot].R-bit= 1 then
22:             A[HAND_hot].R-bit← 0
23:         end if
24:         if A[HAND_hot].H-bit= 0 then
25:             AdaptLargeIRR()
26:         end if
27:         Move HAND_hotforward
28:     end while
29:     A[HAND_hot].H-bit← 0
30:     Move HAND_hotforward
31: end procedure
```

Algorithm 4 Algorithm for Updating Hash Tables

```
 1: procedure UpdateHistory
 2:     Increment n_hit
 3:     if (n_hit > Threshold() or n_{B_*} = c_B) then
 4:         SwitchHashTable()
 5:     end if
 6: end procedure
 7:
 8: procedure SwitchHashTable
 9:     AdaptLargeIRR()
10:     Switch B_*          ▷ If B_* = B_1 then B_* ← B_2; else B_* ← B_1.
11:     Clear B_*                                   ▷ Reset all bits in B_*.
12:     n_hit ← 0
13: end procedure
14:
15: function Threshold
16:     return max(n_h/2, 1)
17: end function
```

く、$n_h < m_h$ に加えて cold チャンク数 n_c が多い場合 (ここでは hot チャンク目標数 m_h の倍以上 cold チャンクが存在する場合) に hot チャンクとしてキャッシュする。そして、キャッシュデータがキャッシュ内に存在しなかった場合 (つまり履歴キャッシュとキャッシュヒットの場合) は、そのデータをキャッシュする (行 25–28)。このとき hot チャンクとしてキャッシュするかどうかは、フラグ h に応じて決定する。

アルゴリズム 2 は、アクセス系列の IRR に適応するために、hot/cold チャンク目標数の調整を行う。アクセス系列の IRR が小さい場合は、LRU-friendly な状況である。この場合、特定のチャンクを優先的に保持する方法を採ると、correlated-reference や fickle-interest に対応できない。したがって、最新のチャンクを重視する LRU/CLOCK に近い動作をすることが望ましい。し

たがって、cold チャンク数を増加させるべきである。一方、アクセス系列の IRR が大きい場合は、scan や loop が発生している状況を意味する。scan に対処するためには、アクセスの多いチャンクを優先的に保持しておくべきである。また、loop に対処するためには、loop の一部を hot チャンクとすることで削除対象外にする戦略が必要である。したがって、IRR が大きい状況では、hot チャンク数を増加させるべきである。hot/cold チャンクの目標数はそれぞれ m_c と m_h として定義されており、アルゴリズム 2 でこの値を調整することによってアクセス系列の特徴に合わせて適応的に振る舞うことができる。

アルゴリズム 2 において CLOCK-Pro と異なる点は、調整タイミングと調整速度である。CLOCK-Pro では、test フラグに依存して m_c を大きくするか小さくするかを判断していた。しかし、CUSH は test フラグを持たない。test フラグの代わりに、CUSH は CLOCK-Pro と対応する処理が実行されたタイミングで調整を行う。具体的には、IRR が小さいと判断するのはヒット時である。IRR が大きいと判断するのは、$\mathrm{HAND}_{\mathrm{hot}}$ 処理時に cold チャンクを通り過ぎた場合と、ハッシュテーブル切り替え時である。また、CLOCK-Pro ではパラメータを 1 ずつ増減させていた。しかし、test フラグに依存して調整タイミングを決定していた CLOCK-Pro と異なり、CUSH は調整する契機が少ないため、加算的増減では高速に適応できない。したがって、ARC のパラメータ調整を参考に、乗算的に速度を定義する。

アルゴリズム 3 は、2 つの針 ($\mathrm{HAND}_{\mathrm{cold}}$ と $\mathrm{HAND}_{\mathrm{hot}}$) の動作を定義している。$\mathrm{HAND}_{\mathrm{cold}}$ の処理は、基本的には $R = 0$ の cold チャンクを発見して (行 2–11)、それを削除する (行 12) ことである。針の回転においては、hot チャンクは無視する。$R = 1$ の cold チャンクも通り過ぎるが、その際にこれを hot チャンクに変換する (行 6–8)。この処理に伴って cold チャンク数が 0 になる可能性があるため、それを防ぐために $\mathrm{HAND}_{\mathrm{hot}}$ の処理をその前に行っている (行 3–5)。cold チャンクを削除して履歴化した後、ハッシュテーブルが一杯になったら、ハッシュテーブルを切り替える (行 12–15)。

$\mathrm{HAND}_{\mathrm{hot}}$ の処理の定義はアルゴリズム 3 の下段に示す。$\mathrm{HAND}_{\mathrm{hot}}$ の処理の目的は、$R = 0$ の hot チャンクを発見して (行 20–28)、それを cold チャンクに降格することである (行 29)。このとき、$R = 1$ の hot チャンクを発見した場合は、$R = 0$ に設定して通り過ぎる。cold チャンクを発見した場合は、IRR が大きいと判断してチャンク目標数を更新する。これは、CLOCK-Pro において、$\mathrm{HAND}_{\mathrm{hot}}$ によって test フラグを OFF にする処理に伴って m_c を減少させる処理と対応している。

アルゴリズム 4 はハッシュテーブルに関連する処理を記述している。UpdateHistory 関数は、ヒットカウント数 n_{hit} を更新し (行 2)、必要ならばハッシュテーブルの切り替えも行う (行 3–5)。ハッシュテーブルの切り替えは、 3.2.3 項で議論したように、n_{hit} が閾値 $n_h/2$(Threshold 関数 (行 15–17) で定義) を超えた場合に行われる。また、ハッシュテーブルが満杯になった場合にも切り替えを実行する。SwitchHadhTable 関数はハッシュテーブルの切り替え処理を定義している。ここで AdaptLargeIRR 関数が実行されている (行 9) のは、CLOCK-Pro において履歴削除時に test フラグが切れたものと判断して m_c を減少させる処理と対応している。注目中のハッシュテーブルを切り替えた後 (行 10)、今から格納を開始するハッシュテーブルを空にする (行 11)。ヒットカウント数も初期化する (行 12)。以上の処理によって、ハッシュテーブルを用いた loop 耐性を備えたキャッシュ履歴を実現する。

4. 提案方式の評価

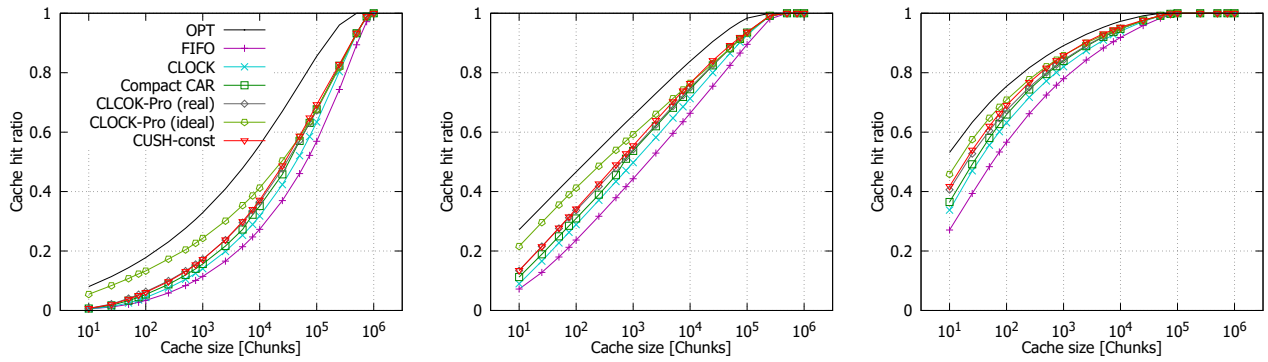
提案方式 CUSH のネットワークトラフィックの特徴的アクセスに対しても高ヒット率を達成できることをシミュレーションによって評価する。まず、CLOCK-Pro のネットワークトラフィックへの適性に関する考察の実証と、CUSH の近似方法による性能への影響を確認する (4.1.1 項)。次に、実環境における評価として、阪大キャンパス内部から YouTube へのアクセスに基いて生成されたコンテンツ単位・チャンク単位のアクセス列を用いて、CUSH が現実のネットワークトラフィックに対して実際に適性を有することを示す (4.1.2 項)。シミュレーション評価に加えて、提案方式 CUSH の計算コストが十分低オーバーヘッドであることを示すために、空間・時間計算量の解析を行う。空間計算量に関しては、実キャッシュ・キャッシュ履歴・検索機構の管理に必要なビット数に基いてメモリオーバーヘッドを算出する。針の回転回数に基いて、CUSH の平均時間計算量を評価する。そして、CLOCK と CUSH の計算量を比較し、メモリ・計算時間共に十分に低オーバーヘッドであることを示す。

4.1 シミュレーション評価

4.1.1 CLOCK-Pro および CUSH の評価

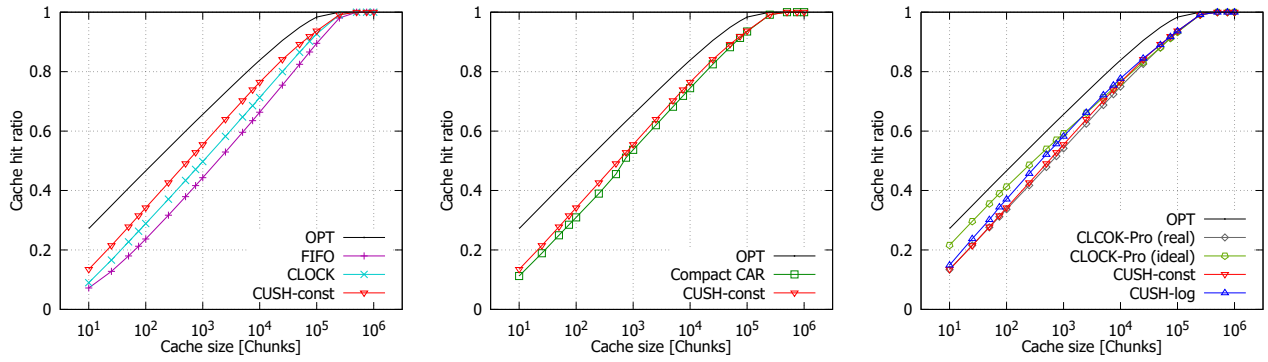
まず、CLOCK-Pro のネットワークトラフィックへの適性に関する考察の実証と、CUSH の近似・改善効果を検証する。ネットワークトラフィックの特徴として、ここでは多量のワンタイマーコンテンツに起因する scan と、チャンク分割に起因する loop について評価を行う。多量のワンタイマーコンテンツを含む要求列を再現するために、実際のネットワークトラフィックが従うとされる Zipf 則に基づいて人口トレースを生成した。更に、後述 (4.1.2 項) の実トレースの統計データに基いて、人口トレースのチャンク数を計算し、チャンク単位での要求列を生成した。これらのコンテンツ単位の人口トレースとチャンク単位の人口トレースについて、キャッシュヒット率を評価する。

評価対象方式は、提案方式 CUSH と、そのオリジナルである CLOCK-Pro、および比較対象として OPT・FIFO・CLOCK・Compact CAR を用いる。OPT は未来の要求列が既知という理想的状況下での最適方式であり、それ以外の方式は ICN ルータでの実運用が可能な方式に焦点を当てる。Random は極めて単純な置換方式であり、置換するチャンクをランダムに選出する低オーバーヘッドな戦略を採る。CLOCK は LRU の近似方式であり、scan や loop に対して性能が発揮できないことを示すために用いる。Compact CAR は scan 耐性を持つように修正された低オーバーヘッドな ARC の近似方式であり、CUSH が scan 耐性の指標として用いる。注意として、CLOCK-Pro と Compact CAR はキャッシュ履歴を持つが、実現可能な方式に焦点を当てるという目的上、これらは図 2 で示した現実的なキャッシュ履



(a) $\alpha = 0.8$ の人口トレース　(b) $\alpha = 1.0$ の人口トレース　(c) $\alpha = 1.2$ の人口トレース

図 4: コンテンツ単位の人口トレースに基づく提案方式のネットワーク適正評価結果



(a) 単純な方式との比較　(b) Compact CAR との比較　(c) CLOCK-Pro との比較

図 5: 比較目的別のコンテンツ単位の人口トレース ($\alpha = 1.0$) に基づく提案方式のネットワーク適正評価結果

歴を用いる。

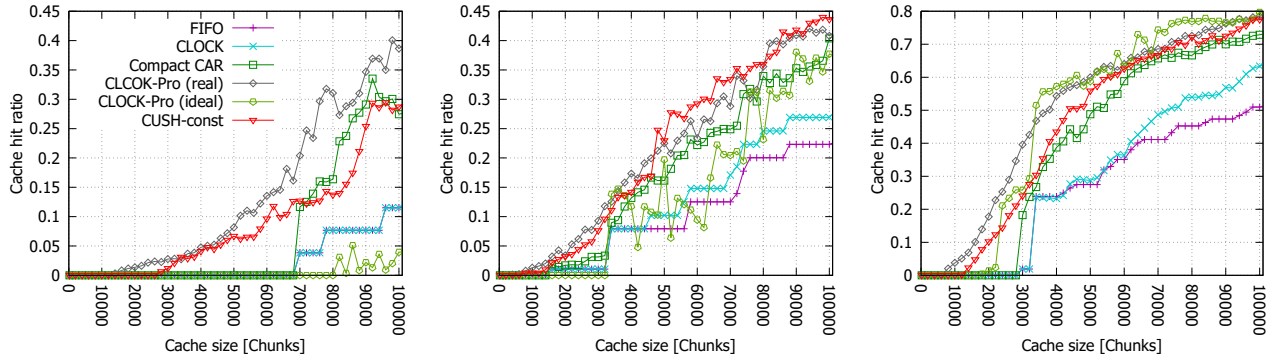
キャッシュサイズなどのパラメータは、多様な環境での運用を想定して、幅広い値を設定する。人口トレース長が 10^7 個程度であるため、キャッシュサイズは 10^1 から 10^6 までの範囲から選択した。ただし、キャッシュサイズの単位はエントリ数とする。評価結果では、loop の影響が最もよく観察された 10^3 から 10^5 の範囲を主に表示している。Zipf 則の偏りを表すパラメータ α には $0.8, 1.0, 1.2, 1.4$ を用いる [21]。チャンクサイズは 1500B・1.5KB・60KB を対象とし、実トレースの統計データからコンテンツサイズを決定してチャンク数を計算した。チャンク単位のシミュレーションでは、シミュレーション規模の限界のために、チャンク分割したアクセス列の一部を取り出して評価している。

CUSH のキャッシュ履歴サイズは、実キャッシュ可能なエントリ数の 4 倍のビット数のハッシュテーブルを持つもの (CUSH-const) と、CLOCK-Pro と同じ $n \log 2n$[bit] だけ用いるもの (CUSH-log) の 2 種類を用いる。CUSH-const は低オーバーヘッドなキャッシュ履歴の性能評価のために、CUSH-log は CLOCK-Pro と同メモリオーバーヘッドでの性能比較を行うために用いる。

評価結果を図 4,5,6,7 に示す。紙面の都合上、全シナリオでの評価結果ではなく、コンテンツ単位の評価結果の一部を図 4、チャンク単位の評価結果の一部を図 6 に示している。見やすさのために、その中から比較目的ごとに一部の方式を取り出した図がそれぞれ図 5,7 となる。

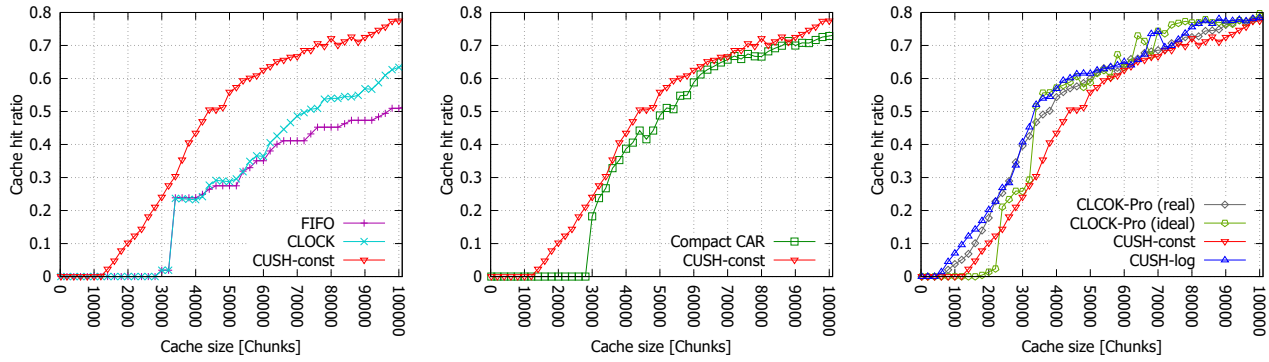
まず、コンテンツ単位の結果である図 5 について見る。このシナリオでは loop は無く、主に scan 耐性の有無が分かる。図 5(a) は単純な方式と CUSH を比較しており、単純な方式に対して提案方式が十パーセント強改善している。CLOCK に 1 bit 追加するコストだけで、CUSH が scan 耐性を実現できている。図 5(b) は scan 耐性を持つ高性能な Compact CAR との比較であり、提案方式が優れている。loop のないコンテンツ単位の結果では Compact CAR が優位と予想されたが、キャッシュ履歴の衝突によって人気のチャンクの保持に失敗したことが原因で CUSH より性能が低くなったと推測される。図 5(c) はオリジナルの方式である CLOCK-Pro との比較を行っている。衝突を解決する理想方式が性能が高い一方で、低オーバーヘッドな提案方式がオリジナルの方式の現実的実装と同等の性能を有することが確認できる。

次に、チャンク単位の結果である図 7 について見る。このシナリオは loop を持つため、loop 耐性の影響を視覚化できる。図 7(a) は単純な方式と CUSH を比較しており、単純な方式がキャッシュサイズがある値を上回るまではヒットが発生しないのに対して、提案方式はキャッシュサイズに応じたキャッシュヒット率を達成できており、loop 耐性を持つことが分かる。図 7(b) は scan 耐性を持つが loop 耐性を持たない Compact CAR との比較であり、loop の影響を受けるキャッシュサイズが小さい領域では図 7(a) とほぼ同様の結果が得られた。図 7(c) はオリジナルの方式である CLOCK-Pro との比較を行っている。CUSH



(a) $\alpha = 1.0, L = 1.5\text{K}$ の人口トレース　(b) $\alpha = 1.2, L = 1.5\text{K}$ の人口トレース　(c) $\alpha = 1.4, L = 1.5\text{K}$ の人口トレース

図 6: チャンク単位の人口トレースに基づく提案方式のネットワーク適正評価結果の拡大版



(a) 単純な方式との比較　(b) Compact CAR との比較　(c) CLOCK-Pro との比較

図 7: 比較目的別のチャンク単位の人口トレース ($\alpha = 1.4, L = 1.5\text{K}$) に基づく提案方式のネットワーク適正評価結果

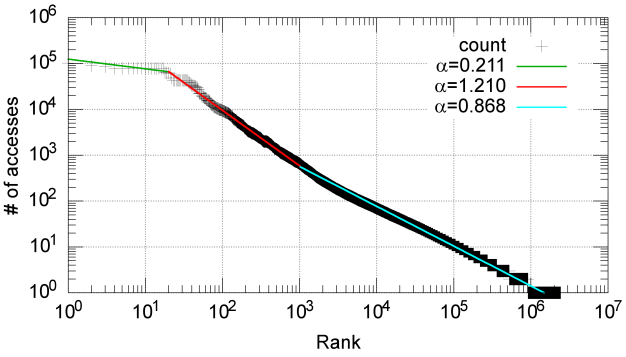


図 8: 阪大の動画アクセス回数の分布

は loop 耐性を持つものの、単純なキャッシュ履歴の実装ではオリジナルの方式よりも性能がやや低い。一方、CUSH-log は CLOCK-Pro を凌ぐ性能を見せている。このように、CUSH は CLOCK-Pro の良い近似であると同時に、loop 耐性を柔軟に拡張できる利点を持つ。また、CLOCK-Pro に関して、衝突がある現実的方式よりも衝突を解決する理想的方式の方が loop 耐性が低くなっている。これは、衝突がある場合は、キャッシュ履歴がランダムに削除されることで履歴が間引かれ、結果的により長い loop に対応できたためである。

4.1.2 実トラフィックにおける性能評価

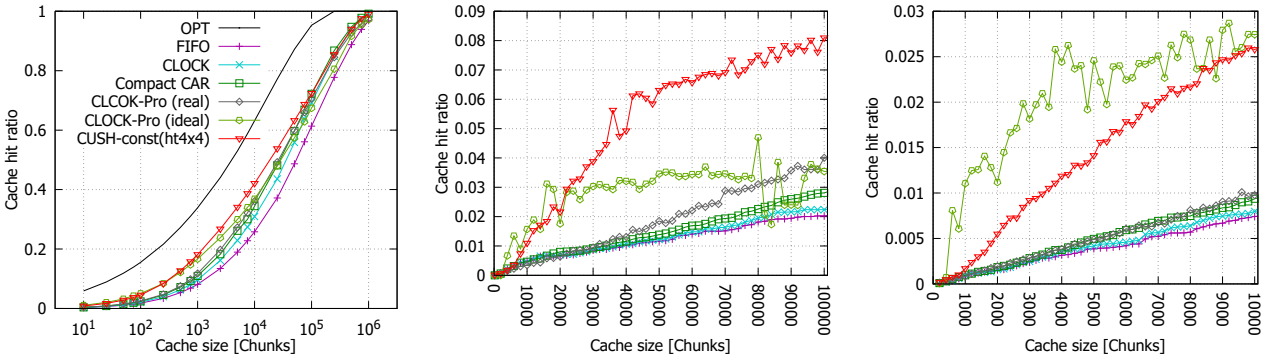
人口的に生成した理想的な特徴を持つトレースではなく、実環境における評価として、阪大キャンパス内部から YouTube へのアクセスに基いて生成されたコンテンツ単位・チャンク単位のアクセス列を用いて、CUSH が現実のネットワークトラフィックに対しても十分な性能を発揮する適性を持つことを示す。

表 2: 動画の秒あたりパケット数 [pck/sec]

チャンクサイズ	1.5KB	15KB	60KB
SD(4.5[MB/min])	50	5	1.25
HD(9.0[MB/min])	100	10	2.50

実トレースデータは、YouTube, nicovideo, dailymotion の 3 つの動画サイトに対する、2013 年 7 月 26 日から 2015 年 2 月 26 日までのアクセスに基づく。ユニークコンテンツ数は 1,451,558 個、2 回以上アクセスのあるコンテンツ数はその約 4 分の 1 の 381,527 個である。この約 1 年半の間の全体のアクセス数は 3,378,925 アクセスである。アクセス分布は図 8 のようになっており、Zipf 分布に近い形で、人気が高いコンテンツへの集中が多いことが確認できる。実際、最もアクセスされたコンテンツのアクセス数は 3,949 回である。

これらのアクセス列はコンテンツ単位でのアクセスだが、そこからチャンク単位での入力列も生成した。具体的には、アクセス日時と動画情報を利用し、動画の長さと対応画質の情報から動画サイズを決定して、予め決定したサイズのチャンクが再生時間を等分割した時間間隔で要求されるようなアクセス列を想定する。チャンクサイズ L は 1500B・15KB・60KB を対象と



(a) コンテンツ単位　(b) チャンク単位 ($L = 15$K)　(c) チャンク単位 ($L = 60$K)

図 9: 実トレースにおける性能評価結果

する。また、動画長 1 分あたりの通信量は、簡単な事前調査の結果、表 2 の規則に従って動画ごとのチャンク数を決定した。$A_{\mathrm{Zipf}(\alpha)}$ に関しては、観測された動画の再生時間と画質の分布に従って、コンテンツごとに再生時間と画質を決定した。

評価結果を図 9 に示す。CUSH のキャッシュ履歴拡張の有効性実証のために、CUSH-const は 4 ビットのキャッシュ履歴エントリを用いて拡張したものを用いている。現実のトラフィックでは人気が推移するため、correlated-reference や fickle-interest を考慮した手法が有効となる。loop のないコンテンツ単位の結果 (図 9(a)) では、CUSH は Compact CAR を凌ぐ性能を発揮しており、十分な scan 耐性を備えていることが分かる。loop のあるチャンク単位の結果 (図 9(b),9(c)) では、CLOCK-Pro(real) は人気の推移に対応できず、十分な性能を発揮できていない。一方、CUSH は CLOCK-Pro(ideal) 以外の方式の性能を大きく凌駕しており、十分な loop 耐性を持つことが分かる。このように、CUSH は単純で拡張コストの低いデータ構造ながらも scan と loop に耐性を持ち、実ネットワークトラフィック環境にも適応可能な優れた方式である。

4.2 計算量評価

4.2.1 空間計算量

代表的なキャッシュ置換方式に関して空間計算量を算出し、CUSH が低メモリオーバーヘッドでキャッシュ履歴を拡張可能な方式であることを示す。メモリオーバーヘッドとして、実キャッシュ制御とキャッシュ履歴に必要なオーバーヘッドを別々に計算する。更に、キャッシュ履歴に関しては、キャッシュ履歴が検索テーブルにかける負荷も考慮する。

計算結果は表 3 にまとめる。ただし、各キャッシュ方式は n 個のエントリをキャッシュできるとする。また、キャッシュ履歴の検索テーブルは、衝突ありハッシュテーブルで実装し、ハッシュテーブルのエントリサイズは n の k 倍の形で表記している。

FIFO と CLOCK は単純な方式で性能は低いが、空間計算量は $O(n)$ で小さい。二重連結リストを用いた LRU である LRU_{DLL} や、ヒープを用いた LFU である LFU_H は空間計算量が $O(n \log n)$ で大きい。LRU と同程度の計算量を目指す LIRS や、可変長の CLOCK を保持する CAR はそれと同等の空間計算量が実キャッシュに必要であるのに加え、キャッシュ履歴保持のために追加の空間計算量が必要である。CLOCK-Pro と Compact CAR は、どちらもキャッシュ履歴を含む各エントリごとに数ビットを割り当て、それに対する索テーブルを保持する必要があるため、実キャッシュおよびキャッシュ履歴共に CLOCK と同じ $O(n)$、それに加えて検索テーブルに n エントリ分の負荷を要する。検索用のハッシュテーブルには冗長な空間が必要であるため、実際にはその k 倍である $O(kn \log n)$ のメモリが必要となる。

一方、CUSH は実キャッシュに必要な履歴のメモリオーバーヘッドが CLOCK と同じオーダーでありながら、検索テーブルに対する負荷を削除し、自身の中に検索と履歴保持機能を兼ね揃えたハッシュテーブルを保持する。更に、容易にキャッシュ履歴のサイズを任意長 kn に拡大可能である。例えば、$k = 1$ ならば CLOCK-Pro と同程度のキャッシュ履歴長を低オーバーヘッドに実現可能である。$k = \log n$ ならば、CLOCK-Pro と同程度のメモリオーバーヘッドでキャッシュ履歴を伸長し、loop 耐性を向上することができる。

4.2.2 時間計算量

CUSH の時間計算量に関して、CUSH の針の平均回転数を表 4 に示す。CLOCK 系統の方式は、針の 1 回の動作ごとに処理が続行するか否かを判断しなければならないため、1 度の針の動作を単位としてその時間計算量を推定する。平均回転数の評価には、キャッシュヒット率に対する平均回転数に着目して、キャッシュサイズのべき乗にキャッシュヒット率が比例する $\alpha = 1.0$ の人口トレースを用いている。平均回転数は、キャッシュヒットとミスを含めた全体の平均回転数と、キャッシュミス 1 回あたりの平均回転数の両方を示している。これは、キャッシュミスの場合しか針が回転しないためである。最悪時間計算量はいずれの場合も $O(n)$ であるため省略する。

CUSH はほとんどの場合で CLOCK と同程度の低い平均計算量を示しており、時間計算量に関しても低オーバーヘッドであることが分かる。表 4 から分かるように、キャッシュヒット率やキャッシュサイズに関わらず、CLOCK は全体で見て 1 回未満、ミスあたりで見て 2 回以下の針の回転数を実現している。一方、CLOCK-Pro は全体的に CLOCK の数倍以上、場合によっては数百倍以上に大きくなってしまっている。これは、CLOCK

表 3: キャッシュ置換方式の空間計算量

方式	キャッシュ管理 [bit]	キャッシュ履歴管理 [bit]	キャッシュ履歴用検索テーブル [bit]
FIFO	$O(\log n)$	-	-
LRU_{DLL}	$O(n \log n)$	-	-
LFU_{H}	$O(n \log n)$	-	-
LIRS (with LRU_{DLL})	$O(n \log n)$	$O(kn \log kn)$	$O(kn \log kn)$
CLOCK	$O(n)$	-	-
CAR (with LRU_{DLL})	$O(n \log n)$	$O(n \log n)$	-
CLOCK-Pro	$O(n)$	$O(n)$	$O(kn \log n)$
Compact CAR	$O(n)$	$O(n)$	$O(kn \log n)$
CUSH	$O(n)$	$O(kn)$	-

表 4: 針の回転数から見たキャッシュ置換方式の時間計算量

		全体の平均回転数				キャッシュミス時の平均回転数			
キャッシュ ヒット率	n	CLOCK	CLOCK-Pro	CUSH	Compact CAR	CLOCK	CLOCK-Pro	CUSH	Compact CAR
0.0–0.1	10	0.98	4.33	1.17	2.91	1.06	3.79	1.35	3.25
0.1–0.2	32	0.92	8.05	1.09	2.78	1.11	6.29	1.41	3.46
0.2–0.3	100	0.84	13.82	0.99	2.52	1.14	9.52	1.45	3.52
0.3–0.4	317	0.74	22.44	0.86	2.21	1.17	13.33	1.47	3.56
0.4–0.5	1000	0.65	32.26	0.71	1.85	1.20	16.11	1.47	3.68
0.5–0.6	3163	0.55	62.34	0.58	1.52	1.25	25.20	1.51	3.73
0.6–0.7	10000	0.45	618.03	0.43	1.17	1.32	190.44	1.51	3.78
0.7–0.8	31623	0.34	3402.53	4.18	0.83	1.46	734.00	20.79	3.86
0.8–0.9	100000	0.23	32.68	0.65	0.40	1.73	4.20	5.14	3.20

よりも 2 本多く針を持つのに加え、履歴ページを円環バッファ中に含むため、それを走査するために針の回転数が増加するためである。特に、キャッシュヒット率が高い状況では、置換用の針が円環バッファ中の大量の hot チャンクを無視するために、針の回転数が大幅に増大している。Compact CAR もチャンクを分類しているが、分類したチャンクを別々の円環バッファで管理しているため、キャッシュサイズ・キャッシュヒット率に関わらずほぼ一定の回転数を維持している。

CUSH は、CLOCK-Pro の近似方式だが、CLOCK とほぼ同程度の時間計算量を達成する。CUSH では、履歴ページを円環バッファ外に保持する工夫と、パラメータ調整の工夫によって針の動作回数を最低限にするアルゴリズムによって、CLOCK-Pro で見られたオーバーヘッドを大幅に削減できている。したがって、CUSH はキャッシュサイズ・キャッシュヒット率に関わらず、CLOCK と同様に計算オーバーヘッドを一定の低い値に抑えることができる。ただし、CLOCK-Pro と同様、一部 (キャッシュヒット率 0.8 付近) では針の回転数が増大傾向にある。このオーバーヘッドが許容できない場合には、Compact CAR のようにチャンク種類ごとにバッファを分割する工夫が必要となりうる。

5. 結　　論

ICN ルータは限られたメモリ・計算資源を用いて高速なネットワークトラフィックを処理しなければならない。また、ネットワークトラフィック中のワンタイマーコンテンツの割合が大きく、単純な方式では対処できない。また、チャンク単位アクセスを考慮すると、loop と呼ばれるキャッシュ置換方式上の問題が発生する。更に、発展的なキャッシュ置換方式で採用されるキャッシュ履歴は、ICN ルータの検索機構に多大なコスト追加を要請する。Compact CAR は、キャッシュ履歴による検索機構への負荷を無視すればワンタイマーアクセスには強い。しかし、検索機構への負荷が無視できない状況ではその特徴を十分に発揮できず、チャンク単位アクセスにも弱い。

これらの課題に対して、本論文では CLOCK-Pro に基づいてワンタイマーコンテンツとチャンク単位アクセスに対処しつつ、検索機構への負荷も含めて少ないメモリ・計算資源で実装可能な方式 CUSH を提案した。CUSH は検索機構に負荷をかけずに拡張可能なキャッシュ履歴機構の近似方式によって、低オーバーヘッドに loop への耐性を向上することができる。シミュレーション評価では、CUSH は CLOCK-Pro の優れた近似としての性能だけでなく、キャッシュ履歴の拡張による適応力を持つ結果が得られ、ネットワークトラフィックに対する有効性を明らかにできた。オーバーヘッドの評価では、CLOCK と同等の実用可能な空間・時間計算量を持つことを示した。結果として、本研究は ICN ルータの厳しい制約条件下で動作するキャッシュ置換方式の設計を示すことで、ICN ルータの実現可能性の実証に貢献した。本研究では単一ルータにおける評価を行ったが、今後ルータ間の協調動作を考慮したキャッシングアルゴリズムを考慮する必要があるだろう。最終的には、そのキャッシュ機構を組み込んだ ICN ルータの実機による実現性の実証を目指す。

謝辞

本研究は、総務省・戦略的情報通信研究開発推進事業 (SCOPE) 受付番号 165007007 の委託による。

文　献

[1] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J.D. Thornton, D.K. Smetters, B. Zhang, G. Tsudik, K. Claffy, D. Krioukov, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley, and E. Yeh, "Named data networking (NDN) project," Oct. 2010. http://named-data.net/techreport/TR001ndn-proj.pdf

[2] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N.H. Briggs, and R.L. Braynard, "Networking named content," Proceedings of the ACM CoNEXT 2009, pp.1–12, Dec. 2009.

[3] N. Fotiou, P. Nikander, D. Trossen, and G.C. Polyzos, "Developing information networking further: From PSIRP to PURSUIT," Proceedings of the 7th International ICST Conference on Broadband Communications,Networks, and Systems, pp.1–13, Oct. 2010.

[4] T. Levä, J. Gonçalves, R.J. Ferreira, et al., "Description of project wide scenarios and use cases," Feb. 2011. http://www.sail-project.eu/wp-content/uploads/2011/02/SAIL_D21_Project_wide_Scenarios_and_Use_cases_Public_Final.pdf

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," Proceedings of IEEE INFOCOM'99, vol.1, pp.126–134, March 1999.

[6] F. Guillemin, B. Kauffmann, S. Moteau, and A. Simonian, "Experimental analysis of caching efficiency for YouTube traffic in an ISP network," Proceedings of the 25th International Teletraffic Congress, pp.1–9, Sept. 2013.

[7] S. Arianfar, P. Nikander, and J. Ott, "On content-centric router design and implications," Proceedings of the ACM Re-Architecting the Internet Workshop, pp.1–6, Nov. 2010.

[8] A. Ioannou and S. Weber, "A taxonomy of caching approaches in information-centric network architectures," Technical report, School of Computer Science and Statistics, Trinity College Dublin, Jan. 2015.

[9] L. Wang, S. Bayhan, and J. Kangasharju, "Optimal chunking and partial caching in information-centric networks," Computer Communications, vol.61, no.1, pp.48–57, May 2015.

[10] D. Rossi and G. Rossini, "Caching performance of content centric networks under multi-path routing (and more)," Technical report, Telecom ParisTech, July 2011.

[11] W.K. Chai, D. He, I. Psaras, and G. Pavlou, "Cache "less for more" in information-centric networks," Computer Communications, vol.36, no.7, pp.758–770, May 2012.

[12] A. Ooka, S. Ata, and M. Murata, "A proposal and evaluation of cache replacement policy for the implementation of ICN router," Technical Committee on Information-Centric Networking (ICN), pp.1–10, July 2015.

[13] S. Jiang and X. Zhang, "Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance," IEEE Transactions on Computers, vol.54, no.8, pp.939–952, Aug. 2005.

[14] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," Proceedings of the USENIX 2005, pp.323–336, April 2005.

[15] J. Wang, "A survey of web caching schemes for the Internet," ACM SIGCOMM Computer Communication Review, vol.29, no.5, pp.36–46, Oct. 1999.

[16] K.-Y. Wong, "Web cache replacement policies: a pragmatic approach," IEEE Network, vol.20, no.1, pp.28–34, Jan. 2006.

[17] A.-M.K. Pathan and R. Buyya, "A taxonomy and survey of content delivery networks," Technical report, University of Melbourne Grid Computing and Distributed Systems Laboratory, Feb. 2007.

[18] S. Arianfar, P. Nikander, and J. Ott, "Packet-level caching for information-centric networking," Technical report, Finnish ICT SHOK, June 2010.

[19] N. Megiddo and D.S. Modha, "ARC: a self-tuning, low overhead replacement cache," Proceedings of the 2Nd USENIX Conference on File and Storage Technologies, pp.115–130, March 2003.

[20] S. Bansal and D.S. Modha, "CAR: Clock with adaptive replacement," Proceedings of the 3rd USENIX Conference on File and Storage Technologies, pp.187–200, March 2004.

[21] C. Fricker, P. Robert, J. Roberts, and N. Sbihi, "Impact of traffic mix on caching performance in a content-centric network," Proceedings of the IEEE Conference on Computer Communications 2012, pp.310–315, March 2012.