Vol.2014-OS-128 No.12
2014/3/7

ソフトウェアOpenFlowスイッチにおけるBitmask対応の高速なフロー検索手法の検討

日比 智也[1,a)]　中島 佳宏[1,b)]　高橋 宏和[1,c)]　尾花 和昭[1,d)]

概要：SDN や NFV を実現する基盤として，x86 サーバ上のソフトウェアとよる OpenFlow スイッチの高速な実装が求められているが，OpenFlow スイッチのフロー検索は，マッチ条件に任意の Bitmask が指定可能であるため高速化が難しく，転送性能劣化の大きな要因となっている．本研究では任意の Bitmask に対応可能な検索手法を検討し，フロー検索の性能向上を目指す．本稿ではトライ木とハッシュを用いる 2 つのアプローチでそれぞれ検討を行った．トライ木を用いたアプローチでは H-Trie と SP-Trie の 2 つの手法とそれらの組合せた手法を，ハッシュを用いたアプローチでは M-List をそれぞれ提案し，実装，評価を行った．評価の結果，任意の Bitmask を含むフロー検索は 2 つのトライ木を組み合わせた提案手法が有効であること得た．

1. はじめに

近年，Software defined networking(SDN) とそれを実現する技術の一つである OpenFlow[7] に注目が集まっている．OpenFlow プロトコルを用いることによって，外部からパケットのマッチ条件や処理を記述したフロールールを OpenFlow スイッチ (OFS) に指定でき，柔軟なフロー制御や仮想ネットワークの構築が可能となる．

OpenFlow/SDN の技術はデータセンタを中心に発展してきたが，その適応領域は拡大し，コスト削減，サービス差異化，サービス開発の迅速化の観点からキャリアネットワークへの適用が進んでいる [17].

OpenFlow の適用範囲の拡大に伴い，OFS に求められる性能も高くなっている．近年のデータセンタにおいては数千˜数万台のサーバが稼働しており，さらにそのサーバ上に仮想マシンが複数台起動することから，これらをつなげる OFS は数万から数十万のフローエントリを扱う必要であり，さらに高いスループットの実現も望まれる．また OFS の実装は専用ハードウェアによる高速な実装だけでなく，プロトコルの新バージョンへの迅速な対応及びコスト低減の観点から x86 サーバ上のソフトウェア (SW) による実装が求められている．しかし，SW による OFS では高い性能を実現することが難しい．特にマッチ条件に任意の bitmask が指定されるため，IP ルーティングで検討されてきた高速な検索手法が適用できず，フロールールの検索が転送性能劣化の大きな要因となっている．

本研究では任意の Bitmask に対応可能なフロー検索手法を検討し，SW 実装による検索性能の向上を目指す．本稿ではトライ木とハッシュを用いる 2 つのアプローチでそれぞれ検討を行った．トライ木を用いたアプローチでは H-Trie と SP-Trie の 2 つの手法とそれらの組合せた手法を提案，実装し，ハッシュを用いたアプローチでは M-List を実装した．これらをランダムな Bitmask を含むフロー検索の性能を測った結果，H-Trie と SP-Trie の 2 つの手法を組み合わせた提案手法が任意の Bitmask を含む OFS のフロー検索に有効であることを得た．

以下，二章では OFS のフロー検索の課題と関連研究を示す．関連研究として OFS のフロー検索とともに，OFS と似た検索を行う Packet Classification について示す．三章で Bitmask に対応したフロー検索手法を示す．まずトライ木を用いたアプローチのアルゴリズムとして H-Trie と SP-Trie を示し，後に両アルゴリズムの組合せを提案する．また同時にトライ木の高速化の為の実装手法について述べる．次にハッシュを用いたアプローチのアルゴリズムとして M-List を示す．M-List は Open vSwitch の MegaFlow と呼ばれる機能の検索アルゴリズムと同じアルゴリズムである．四章ではそれぞれの提案手法について実装，評価を行った結果を示し，最終章で本稿のまとめを示す．

[1] 日本電信電話株式会社 未来ねっと研究所
[a)] hibi.tomoya@lab.ntt.co.jp
[b)] nakajima.yoshihiro@lab.ntt.co.jp
[c)] takahashi.hirokazu@lab.ntt.co.jp
[d)] obana.kazuaki@lab.ntt.co.jp

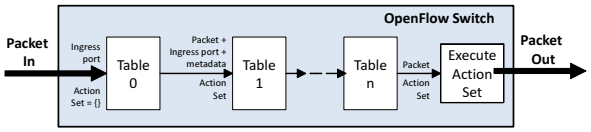

図 1　パイプライン

Match Field	Priority	Counters	Instructions	Timeouts	Cookie

図 2　フローエントリ

2. OFS におけるフロー検索の課題と関連研究

2.1 OFS におけるフロー検索の課題

OFS におけるパケット処理の基本的な流れを図 1 に示す [3]．OFS はいくつかのフローテーブルを持ち，それぞれのフローテーブルでフローエントリを検索し，パケットに対する処理を決定する．この処理を行うフローテーブルの集合をパイプラインと呼ぶ．

フローエントリは図 2 に示す要素を持つ．各フローテーブルで match fields にマッチするフローエントリの中で最も Priority が高いフローエントリを検索する．match fields には図 3 にあるようなパケットのヘッダ情報が与えられる．図 3 において，Mask の項目が Yes となった領域については，フローエントリ毎に任意の bitmask を与えることができ，フローエントリを検索する際には入力されたパケットのヘッダ情報に bitmask を適用し，マッチするか比較することになる．またマスクを指定できないフィールドにおいても，必ず情報を指定する必要はなく，any として扱うことができる．

図 4 に MatchFields の例を示す．各フィールドの値は address/mask で表し，any であるフィールドは”*”とする．この例では各フローエントリを単純なプレフィックスマッチで検索できない．また 1 番目のエントリと 2 番目のエントリが同時にマッチするパケットが存在する為，そのようなパケットが来た場合には，priority の高い 2 番目のエントリをマッチさせる必要がある．

このようなプレフィックスマッチではないエントリや複数のエントリがマッチする例の存在が OFS のフロー検索を難しくしている．

Field	Bits	Mask
Ingress port	32	No
Metadata	64	Yes
Destination MAC address	48	Yes
Source MAC address	48	Yes
Ethernet type after VLAN tags	16	No
VLAN-ID from 802.1Q header	12+1	Yes
IPv4 or IPv6 protocol number	8	No
IPv4 source address	32	Yes
IPv4 destination address	32	Yes
TCP/UDP destination port	16	No
TCP/UDP Source port	16	No

図 3　Match Field(一部)

Priority	MAC src	IPv4 dst	IPv4 src	L4 port dst	Instruction
100	00:00:00:11:11:00 /00:00:00:FF:FF:00	10.0.0.0 /255.255.0.0	192.168.1.0 /255.255.255.0	*	Apply-Actions
9999	*	192.168.0.1 /255.255.255.255	*	5533	Drop
0	*	*	*	*	Drop

図 4　Priority と MatchFields の例 (address/mask)

2.2 関連研究

2.2.1 OpenFlow lookup

2008 年に OpenFlow の Web サイトで公開された”reference implementation”では，フロー検索を bitmask や any を含まない Entry に対してハッシュで検索を行い，bitmask や Wildcard を含むものに対して，線形探索を行うものであった．Naous らはこれを NetFPGA 上で実装した [9]．x86 サーバ上の実装では，ハードウェアによるオフロードによって高速化するアプローチが検討されている．Han らは GPU を用い [6]，Tanyingyong らは NIC にフロー検索の一部をオフロードすることによって高速化を行った [14]．しかし，これらの実装は線形探索を含むため，Mask を持つエントリ数が増大するに従い，検索性能が線形に劣化する．

Open vSwitch（OVS）は広く利用されているソフトウェア OpenFlow スイッチである [1]．OVS ではフローキャッシュを用いることでフロー検索を高速化している．最初にきたパケットを通常どおり検索し，検索結果をフローキャッシュに貯める．2 つ目以降のパケットは 1 つ目のパケットヘッダの情報がキャッシュに登録されているため，高速な検索（Hash による検索）が可能となる．OVS の 1 つ目のパケットのフロー検索は基本的に線形探索である．フローをプライオリティで降順にならべ，先頭から探索することで最もプライオリティの高いフローが最初に検索できるよう工夫している．また OFS は検索するヘッダ領域に対して，検索に使用しないヘッダ領域が大きい為，miniFlow という圧縮をかけ，Match Fields とパケットヘッダの比較を高速化している．

OVS1.11 以降では，フローキャッシュに MegaFlow と呼ばれる機構が追加された．これまでヘッダ情報をすべてフローキャッシュに登録していたが，MegaFlow ではマッチしたフローのマスクとマスクをかけたキー（ヘッダ情報）を登録する．これにより，一つのフローキャッシュで複数のフローをマッチさせることができる．MegaFlow における Lookup アルゴリズムは Mask-List として 3.3 節に示す．

2.2.2 Packet Classification

OFS におけるフロー検索と似た技術として，Packet Classification がある．Packet Classification では優先度とプレフィックスマッチを幾つか含むテーブルに対して検索を行

い，パケットの処理を行う．OpenFlow におけるフロー検索と同様に，検索は L2 から L4 までマルチレイヤのヘッダ情報を扱い，各フィールドには Wildcard の他に L3 の最長プレフィックスマッチや L4 のポートの範囲が指定される．

Packet Classification の検索アルゴリズムはこれまで様々な検討がなされている [5], [15]．この中で最も単純な検索手法は線形探索である．線形探索は $O(n)$ の時間と $O(n)$ の空間で検索が可能である．

この他にシンプルなアルゴリズムとして Hierarchical Trie(H-Trie) が知られている．H-Trie の検索は $O(W^d)$ の検索時間と $O(NdW)$ の空間を要する．ここで W は各フィールドの中で最も大きい bit 長であり，d は検索するフィールドの数，N はエントリ数を表す．同じトライ木を扱うアルゴリズムとして Set-Pruning Trie(SP-Trie) が知られている．SP-Trie は H-Trie から検索のバックトラックを削除したトライ木であり，$O(dW)$ の計算時間と $O(N^d)$ の空間でフローの検索が可能である．Qiu らはこの 2 つのアルゴリズムを改善し，各フィールドにおけるメモリアクセス数を削減し，Directed Acyclic Graph(DAG) を利用することによって，SP-Trie のメモリ使用量を抑えた [11]．本稿ではこの H-Trie と SP-Trie について検討を行っている．

その他のアプローチとして，フロー検索を d 次元の点位置決定問題に還元し，これを解くことでマッチするエントリを検索するアルゴリズムが提案されている．Rovniagin らは L3 の送信元と宛先，及び L4 の送信元と宛先の値を用いて 4 次元の長方形とし，$O(4 \log n)$ の探索時間と $O(n^4)$ の空間で検索する手法を示した [12]．これは一般の d 次元の問題に対しては $O(d \log n)$ の探索時間と $O(n^d)$ の空間を要する．

また決定木を用いてマッチする d 次元の長方形を決定するアプローチも提案されている．決定木を用いたアプローチは Gupta らが Hicuts と呼ばれるアルゴリズムを示し [4]，Singh らがそれを改良した HyperCuts を示した [13]．近年では，Qi らがこれらの検索速度を改善した HyperSplit を，vamanan らがメモリ効率を改善した Efficuts をそれぞれ提案している [10], [16]．これらのアルゴリズムは $O(d)$ の探索時間と $O(n^d)$ のメモリを必要とする．

3. 手法検討

Packet Classification の検索手法の多くは Bitmask 対応ではない．しかし，フィールドの各 bit をそれぞれ一つのフィールドとして扱うと，Bitmask を含むフロー検索の問題は L 次元の Packet Classification とすることができる．L は検索を行うヘッダの bit 長である．これにより，Packet Classification の検索アルゴリズムを OFS のフロー検索に適用することができる．

点位置決定問題や決定木を用いたアプローチは検索するテーブルがすべて最初から与えられていることを想定しており，検索は高速であるがテーブル書き換えによる検索木の更新に時間がかかる．OpenFlow ではコントローラから頻繁なテーブルの書き換えが要求されることもあるため，前処理に時間がかかるアルゴリズムを適用することは難しい．その為，本稿では単純なトライ木とハッシュの 2 つアプローチで検索手法を検討する．

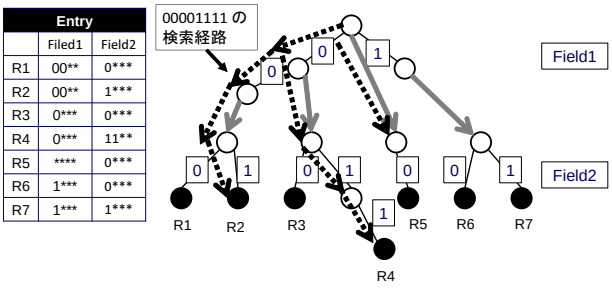

図 5　H-Trie の例

3.1 トライ木によるアプローチ

本節ではトライ木を用いたアプローチとして，Hierarchical Trie(H-Trie) と Set-Pruning Trie(SP-Trie) を示す．まず H-Trie を提示し，H-Trie を 0,1,*の三種類の文字列を扱うパトリシア木とすることで最悪計算量を $O(2N-1)$ に抑えられることを示す (ただし，bit 列の比較は定数時間で完了することを仮定する)．次に SP-Trie について説明し，最後に両手法を組合せた検索手法について提案する．

3.1.1 Hierarchical Trie

Hierarchical Trie(H-Trie) は Packet Classification において，最もメモリ効率の良い検索手法の一つである．図 6 に H-Trie の例を示す．図の左はテーブル例であり，右はそれに対する H-Trie である．テーブルの各エントリは Field1 と Field2 にそれぞれ 4bit の情報を持つとする．また，この H-Trie に対する”00001111”の検索経路を破線で示す．

H-Trie の検索はマッチする可能性のある辺はすべて辿るバックトラッキングのアルゴリズムである．図 5 の例では，Field1 でマッチした 3 つの頂点に対して，Field2 の検索を行っている．

H-Trie の Bitmask 対応は各 bit を 1 つのフィールドとすることで実現する．各 bit を一つのフィールドとすることで，各 bit での比較はプレフィックスマッチであると言い換えることができる．ビット長が 1 のプレフィックスマッチは 0,1,*のいずれかしか無いことから，Bitmask 対応 H-Trie は 0,1,*の三分木のトライ木となる．

更にこれを {0,1,*} の三種類の文字を扱うパトリシア木として捉えることで Bitmask 対応 H-Trie のメモリアクセス数を $2N-1$ とできる．パトリシア木はトライ木の中の子がひとつしかないノードを削除したトライ木であり，パトリシア木の各辺は単一の bit ではなく bit 列でラベル付けされる [8]．そのため，パトリシア木では葉以外のノードは

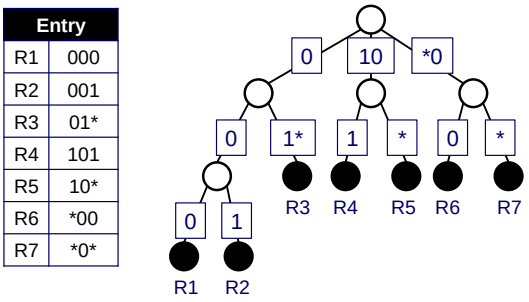

図 6　bitmask に対応した H-Trie の例

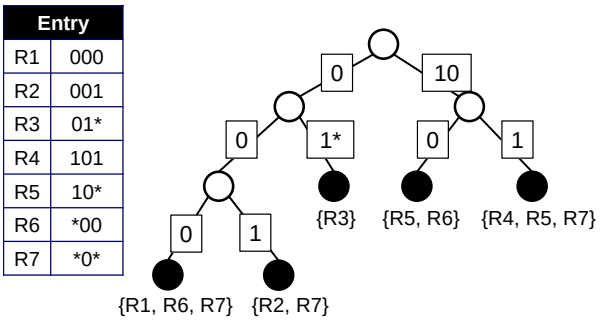

図 7　SP-Trie の例

少なくとも 2 つの子を持つ．さらに葉は高々 N 個であることから，木全体のノード数は高々 $2N-1$ であり，Bitmask 対応 H-Trie の最悪のメモリアクセス回数も $2N-1$ で抑えることができる．

線形サーチはメモリアクセス回数が N であるため，最悪のメモリアクセス数は線形サーチの方が少ないが，実際には H-Trie の方が探索空間を効率的に削減ができ，一度のメモリアクセスで比較するビット長も短いため，検索速度は H-Trie の方が速い．

3.1.2　Set-Pruning Trie

H-Trie は線形サーチよりも速いが計算量は $O(n)$ であり，高速な検索とは言えない．Set-Pruning Trie(SP-Trie) は H-Trie のバックトラックをなくし，検索を高速化する．各頂点でマッチする可能性のあるエントリを集合として扱い，木を辿る毎に，その集合を刈り取ることで検索を行う．図 7 に SP-Trie の例を示す．

SP-Trie におけるエントリの追加は，通常のトライ木の追加操作とほぼ同じであるが，*で分岐する場合にのみ，0 側と 1 側の両方の部分木にエントリを追加する．各頂点で 2 つの部分木にエントリを追加する可能性があるため，パトリシア木によって木のノード数を $2N-1$ におさえたとしても，一つのエントリの追加でノード数が $2N$ 個増加することがある．このため，SP-Trie の最悪空間計算量は $O(2^n)$ か，木の深さは高々 L であることから $O(2^L)$ である．ここで L は各エントリの bit 長である．

Qiu らは SP-Trie を木ではなく DAG とすることでメモリ使用量を抑えた [11]．しかし，DAG は管理が複雑であり，最悪のケースに対する保証は変わらない．

3.1.3　H-Trie と SP-Trie の組合せ

SP-Trie は高速に検索することが可能であるが，空間効率が悪い．最悪の空間計算量が $O(2^L)$ であることから，L が 48（MAC アドレス長）であっても 10^{12} 個以上のノードが必要となり，そのまま実装すれば大半の環境でメモリ不足が発生する．その為，SP-Trie を直接 OFS のフロー検索に適用することは難しい．

これを解決するため，H-Trie と SP-Trie の組み合わせた手法を考える．Qiu らは，各エントリに対し，検索経路が長くなるものには SP-Trie と同じ追加方法を行い，検索経路が短い物は H-Trie と同じ追加を行う組合せ手法を示している [11]．しかし，この手法では前節で述べたように，SP-Trie としてエントリを追加する場合には，一つのエントリの追加でメモリ使用量が倍になることがあり，有効な手法であるとは言えない．そこで，本節では別の H-Trie と SP-Trie の組合せ手法を提案する．

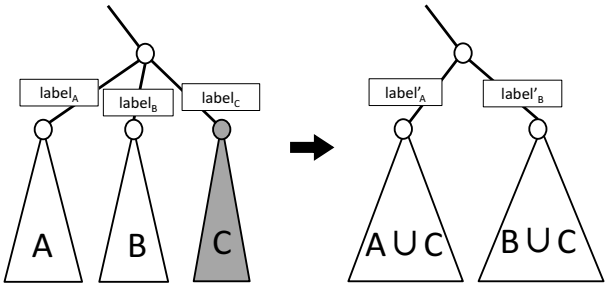

図 8　バックトラックの削減

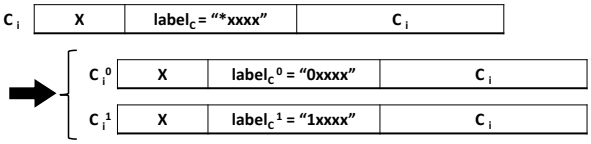

図 9　エントリの分割

H-Trie のあるノード x を観察する．ノード x は A,B,C の 3 つの部分木を持つ (図 8 左)．部分木 A は”0”で始まるラベル $label_A$ を持つ辺で接続し，B は”1”で始まるラベル $label_B$，C は”*”で始まるラベル $label_C$ を持つ辺で接続される．H-Trie の検索では，ノード x まで辿った場合，次の bit が 0 なら A と C，1 なら B と C を辿る．バックトラックは x で 2 つの部分木をたどる必要があることが原因で発生する．その為，図 8 の右に示すように，部分木 C に含まれるエントリ集合を A と B にそれぞれ追加することで，ノード x におけるバックトラックを削減できる．

C に含まれるエントリを A および B に追加する．根から x まで辿った label をつなげたビット列が X，C からエントリ C_i が登録された葉まで辿った label をつなげたビット列が C_i とすれば，C に含まれるエントリ C_i は図 9 のようなビット列となる．ここで，$label_c$ の先頭ビットを 0 にしたものを $label_c^0$，1 にしたものを $label_c^1$ とする．$label_c$ を

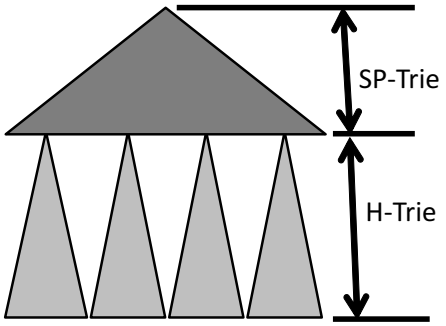

図 10 SP-Trie と H-Trie の組合せ

$label_c^0$，$label_c^1$ に置き換えたエントリ C_i^0，C_i^1 とする．この時，明らかに C_i にマッチするエントリは C_i^0 か C_i^1 のどちらかにマッチし，更に C_i^0 と C_i^1 をトライ木に追加する場合には部分木 A か B にエントリとして追加される．以上のことから，C_i^0 と C_i^1 をトライ木に追加し，C_i のエントリを削除することによって，部分木 C に含まれていたエントリを A と B に移すことができる．これを部分木 C を持つようなノードに対して繰り返し行うことで H-Trie の構造を SP-Trie に近づけることができる．この操作の各繰り返しにおけるエントリの増加は高々一つであるため，メモリの上限まで可能なかぎりバックトラックを減らすことができる．

バックトラックの削減操作をどのノードから行うと効率良く高速化できるかの判断は簡単ではない．その為，本稿では根から順に幅優先探索のようにエントリの移し替え処理を行った．図 10 に概略図を示す．根から順に SP-Trie のようなバックトラックの無い頂点に切り替えることによって，バックトラックを葉付近の短い距離のみに抑えることができる．

3.2 トライ木の高速な実装

3.2.1 Double Array

トライ木の高速な実装として，Double Array が知られている [2]．通常，木を構成するためには，各ノードで子の数だけポインタを確保する必要がある．しかし，Double Array では各ノードは base と呼ばれる index のみを持ち，子の index は (base + 辺のラベルの値) で求める．これにより配列の要素サイズが小さくなることでキャッシュメモリに乗る可能性が高くなり，高速化を実現する．

3.2.2 van Emde Boas layout

さらにキャッシュのヒット率向上の為，配列上に配置されるノードを van Emde Boas layout(vEB) で再配置をした [3]．図 11 に vEB の構造と例を示す．vEB は再帰的な構造を取り，配置換えを行う木の半分の高さまでの部分木を配列の最初に配置し，それ以下の部分木をその後に配置する．各部分木内のノードの配置は再帰的に vEB を適用

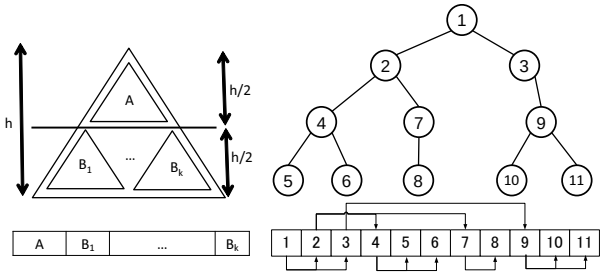

図 11 van Emde Boas layout

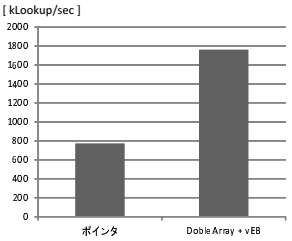

図 12 ポインタと Double Array+vEB による実装の性能比較 (Bitmask を適用しない，50 万エントリの検索性能)

する．ノード数が 1 つであるとき，vEB は配列の始点に配置する．

vEB はその配置により，あるノードの親子はメモリ空間上，近くに配置されることが多いため，キャッシュミスが減る．また再帰的な構造を取る Cache-Oblivious なアルゴリズムであることから様々なサイズのキャッシュを持つハードウェアに対しても，安定した高速化が見込める．

図 12 にポインタによるトライ木と Double Array と vEB によるトライ木の性能比較を行った結果を示す．ポインタによる実装より，Double Array と vEB を用いた方が 2 倍以上高速に動作することが分かる．

3.3 Hash によるアプローチ

本節では，ハッシュを用いて任意の bitmask を含むフローテーブルを検索するアルゴリズムを示す．代表的なソフトウェア OpenFlow スイッチである Open vSwitch のバージョン 1.11 で導入された MegaFlow において，任意の bimask を含むフローキャッシュの検索として使用されているアルゴリズムあり，本稿では Mask List(M-List) と呼ぶ [1]．

3.3.1 Mask List

図 13 で M-List について説明する．M-List では，Bitmask と key をセットにして，ハッシュテーブルに登録を行う．このとき，Mask を図中の Mask list と呼ぶテーブルに登録をする．Mask list はエントリに含まれるマスクの集合であり，図の例では 4 種類の Mask が登録される．これに対して，検索したいパケットが来た際，Mask list に含まれるすべての Mask を使用し，ハッシュ値を求め，ハッシュテーブルを検索する．もし，マッチするエントリがあ

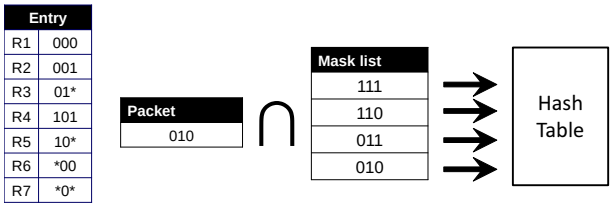

図 13　Mlist の例

る場合，そのエントリの値はパケットに Mask をかけた値と Mask の値のセットと同じであるため，ハッシュによる検索が行える．

M-List は全種類の Mask で Hash を計算するため，最悪計算量は $O(M)$ である．ここで M はテーブルに含まれるマスクの種類数である．

4. 評価

4.1 評価方法

Bitmask が 100 種類である場合と 1 種類である場合に対して，エントリ数を変化させ，各手法のフロー検索の性能評価を行った．Bitmask は乱数によって与え，テーブルの各エントリはビット長が 32bit である乱数に Bitmask を適用したものとした．

4.2 環境

試験は以下の環境で行った．今回は単純なアルゴリズム評価の為に並列化は行っていない．

ハードウェア

CPU　Intel®Xeon Processor E5-2660

メモリ　DDR3-1600 ECC
Quad-channel:8 × 8GByte

ソフトウェア

OS　Ubuntu 12.04.3 LTS

コンパイラ　GCC 4.6.3

4.3 評価結果

4.3.1 100 種類の Bitmask

ランダムな Bitmask が適用された場合のフロー検索性能の評価結果を示す．これは送信元と宛先に対して 10 種類ずつ Bitmask が利用されることを想定してもその組合せ数は高々 100 である為であることか，Bitmask の種類は 100 種類とした．また評価する手法は，H-Trie と SP-Trie，M-List の 3 つと，使用するメモリ領域を約 2，4，8GByte でそれぞれ抑えた H-Trie と SP-Trie を組合せ検索手法の合計 6 種類で行った．

図 14 に評価の結果を示す．グラフの横軸がエントリ数，縦軸が検索速度である．検索速度は”lookup/sec”で表した．”lookup/sec”は 1 秒あたりに検索したエントリ数の平均である．SP-Trie の結果は，トライ木のノード数が 2^{31} を超えるため，400000Entry までしか評価を行わなかった．

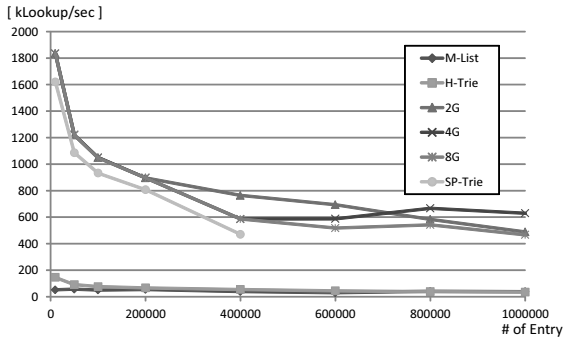

図 14　評価結果：ランダム

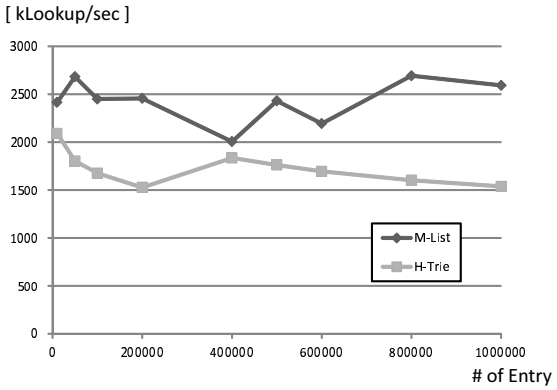

図 15　評価結果：1 種類の Bitmask

グラフより，SP-Trie と 2 つのトライ木の手法を組合せた手法が高い検索性能を得られた．SP-Trie はメモリの上限があったため，多数のエントリをもつテーブルに対するフロー検索はできなかった．また，H-Trie と M-List はメモリ効率はよいが，高速な検索ができなかった．

H-Trie と SP-Trie を組合せた手法について，メモリの上限を 2GByte とすると 200000 エントリから線形に性能が悪化していることが読み取れる．これは，SP-Trie へ変換を行うメモリ上限を超えてしまい，バックトラックが発生する頂点が増えていくためである．また 4GByte と 8GByte では性能に差がほとんど見られなかった．これはバックトラック削除による検索の高速化と共に頂点数増加によるキャッシュミスによるアクセス時間の増加が検索性能に影響を与えたためだと考えられる．

4.3.2 1 種類の Bitmask

1 種類の Bitmask に対する評価を行った．Bitmask が一種類であれば，H-Trie と SP-Trie は同じ構造のトライ木となるため，評価は M-List と H-Trie のみで行った．

図 15 に 1 種類の Bitmask における検索結果を示す．Bitmask が一種類であれば，M-List によるフロー検索は一度ハッシュを計算するのみで検索可能であるため，高速である．しかし，H-Trie による検索も，M-List ほどではないが十分に高速な結果が得られている．

5. まとめ

本研究では Bitmask 対応の高速なフロー検索手法を確立するため，トライ木とハッシュを用いたアプローチで検索手法の検討を行った．トライ木を用いたアプローチでは H-Trie と SP-Trie，さらに使用するメモリの上限を指定可能な H-Trie と SP-Trie を組合せる新しい手法を提案，実装した．またハッシュを用いたアプローチではテーブルの全種類の Bitmask に対してハッシュ検索を行う，M-List を実装した．

これらをランダムな値で評価を行った結果，SP-Trie と 2 つのトライ木を組合せた手法が良い検索性能を出すことが可能であることを示した．SP-Trie はメモリを大量に使用するため実用に耐えないが，2 つのトライ木を組合せた手法は使用するメモリの上限を指定することができ，さらに Bitmask の種類が少ない場合でも十分高速に動作することから，OFS における Bitmask 対応のフロー検索手法として有効である．

今後は，これらの手法の並列化を行い，高速化を実現する．また組合せ手法におけるバックトラックの削減操作を行う順序も検討を継続する．

参考文献

[1] http://openvswitch.org/.
[2] J-I Aoe. An efficient digital search algorithm by using a double-array structure. *Software Engineering, IEEE Transactions on*, 15(9):1066–1077, 1989.
[3] Open Networking Foundation. Openflow switch specification version 1.3.2 (wire protocol 0x04).
[4] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *Micro, IEEE*, 20(1):34–41, 2000.
[5] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *Network, IEEE*, 15(2):24–32, 2001.
[6] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.
[7] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.
[8] Donald R Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.
[9] Jad Naous, David Erickson, G Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an openflow switch on the netfpga platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9. ACM, 2008.
[10] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. Packet classification algorithms: From theory to practice. In *INFOCOM 2009, IEEE*, pages 648–656. IEEE, 2009.
[11] Lili Qiu, George Varghese, and Subhash Suri. Fast firewall implementations for software and hardware-based routers. In *Network Protocols, 2001. Ninth International Conference on*, pages 241–250. IEEE, 2001.
[12] Dmitry Rovniagin and Avishai Wool. The geometric efficient matching algorithm for firewalls. *Dependable and Secure Computing, IEEE Transactions on*, 8(1):147–159, 2011.
[13] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224. ACM, 2003.
[14] Voravit Tanyingyong, Markus Hidell, and P Sjodin. Using hardware classification to improve pc-based openflow switching. In *High Performance Switching and Routing (HPSR), 2011 IEEE 12th International Conference on*, pages 215–221. IEEE, 2011.
[15] David E Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.
[16] Balajee Vamanan, Gwendolyn Voskuilen, and TN Vijaykumar. Efficuts: optimizing packet classification for memory and throughput. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 207–218. ACM, 2010.
[17] SATO Yoichi, Ichiro FUKUDA, and Tomonori FUJITA. Deployment of openflow/sdn technologies to carrier services. *IEICE transactions on communications*, 96(12):2946–2952, 2013.