学位論文　博士（工学）

アプリケーションルータにおける情報抽出および テーブル検索のアクセラレーションに関する研究

2015 年度

慶應義塾大学大学院理工学研究科

八巻 隼人

論文要旨

近年のインターネット関連技術の急速な進歩によって，IDS（Intrusion Detection System）といった高度な機能を提供するサービスの需要が高まっている．ルータにおいてもパケット解析技術を獲得することで，ネットワーク経路上のパケットに対し，ペイロードまで含めた情報抽出を可能とするアプリケーションルータが広く研究されている．アプリケーションルータは，積極的にパケットのコンテンツ情報を解析し，サービスに用いる機能を持ったルータである．アプリケーションルータによって，ネットワーク型 IDS のようなセキュリティ用途のみならず，負荷分散や QoS（Quality of Service）保証といった様々なサービスを高度に提供することが期待できる．

アプリケーションルータは，ペイロードに対する情報抽出を実現するために，GZIP 圧縮された HTTP パケットの展開機構とペイロードに対する文字列探索機構を備える必要がある．既存研究では，これらの処理機構に関して数 Gbps 程度を得ることが限界であった．今後の帯域幅の向上やコアネットワークへの対応を考慮すると，アプリケーションルータにおいてこれらの機構がボトルネックとなりうる．加えて，アプリケーションルータは情報抽出結果をもとにテーブルエントリの追加や変更を頻繁に行う．これによって，テーブル検索機構もボトルネックとなることが懸念される．

そこで本論文では，アプリケーションルータにおける GZIP 展開および文字列探索，テーブル検索に焦点を当て，それぞれのボトルネックを解決するアーキテクチャを提案し，評価した．そして，100Gbps ネットワーク環境においても，アプリケーションルータがワイヤレートでパケットを処理可能となったことを示した．

第 1 章では，本研究の目的と本論文の構成について述べた．第 2 章では，ルータの変遷と，アプリケーションルータの概要および実現されるサービス例に関して述べた．第 3 章では，一般的なルータおよびアプリケーションルータの処理機構と，それに関する既存研究を紹介し，アプリケーションルータにおける性能ボトルネックを明らかにした．第 4 章では，GZIP 圧縮された HTTP パケットの展開処理に関して，キャッシュ及び処理並列性を活かしたハードウェアアーキテクチャと，ピギーバックパケットを用いた複数台アプリケーションルータ間の処理高速化手法を提案した．提案アーキテクチャにより GZIP 展開において 100Gbps 以上の実効スループットが得られることを示した．第 5 章では，ペイロードに対する文字列探索処理に関して，ラビン-カープ法を基にした処理オフローダのハードウェアアーキテクチャを提案した．提案アーキテクチャにより従来の実装に対し 36%の回路規模で，文字列探索処理負荷を 5%まで削減でき，既存手法により 100Gbps 以上の実効スループットが得られることを示した．第 6 章では，テーブル検索処理に関して，従来の TCAM 方式に併せてフローキャッシュを用いることで，テーブル検索を高速化，省電力化するアーキテクチャを提案した．加えて，キャッシュミスを適切に削減するエントリ制御手法を実装することで，最短パケット長における 400Gbps 以上のスループットを従来の 17.9%の消費電力で行えることを示した．最後に，第 7 章において，各章の内容をまとめ，本研究の成果を要約するとともに，研究の発展性について言及した．

Abstract

The rapid progress of Internet technologies in recent years makes increase of demand for high functionality services, such as IDS (Intrusion Detection System). To meet the demand, Application Router, which enables to extract and analyze information including a payload of a packet on a network route by using packet analysis technologies, has been studied. Application Router analyzes content information of packets actively and provides the results of analysis as a service. It is expected to provide various services to high functionality for not only security such as Network IDS but also load balancing and QoS (Quality of Service) guarantee by using Application Router.

Application Router needs a GZIP decompression mechanism for compressed HTTP packets and a string match mechanism for extracting information from the packet payload. Existing studies of these mechanisms could achieve only several Gbps. For taking the increase of future network bandwidth and handling a core network into account, the GZIP decompression process and string match process will become bottlenecks in Application Router. Additionally, Application Router often adds and updates table entries based on results of information extraction. There is a concern that the table retrieval process will become a bottleneck too.

In this paper, we focused on the GZIP decompression, string match, and table retrieval in Application Router and proposed and evaluated architecture for eliminating the each bottleneck. Finally, enough performance was achieved for processing information extraction and table retrieval at 100Gbps wire speed.

In chapter 1, a purpose of this study and construction of this paper were described. In chapter 2, the transition of routers and the outline and services of Application Router were explained. In Chapter 3, the details of packet processing architecture in general router and Application Router were introduced. Furthermore, related studies of them were introduced. Based on these studies, the bottlenecks of Application Router were disclosed. In chapter 4, about the GZIP decompression process for compressed HTTP packets, the hardware architecture used cache mechanism and effective parallelism and the decoding dictionary piggybacking function for high-speed decompression were proposed. We showed the architecture could decode the GZIP packets at over 100Gbps effective throughput. In chapter 5, about the string match process for packet payload, hardware architecture based on Rabin-

Karp algorithm for offloading a load of string match process was proposed. We showed the architecture reduces the hardware cost to 36% and the processing load to 5% compared with conventional architecture and achieve over 100Gbps effective throughput by using existing studies. In chapter 6, about the table retrieval process, flow cache architecture combined with conventional TCAM method for fast table retrieving with low energy consumption was proposed. In this architecture, several techniques for reducing the cache-miss were implemented. As a result, the architecture could achieve over 400Gbps throughput under a minimum packet size processing and reduce the power consumption to 17.9% compared with conventional architecture. In chapter 7, summaries of these studies were described. Furthermore, future visions of these studies were referred.

目 次

表 目 次

図 目 次

第1章　緒論

1.1　本研究の目的

近年，インターネットは世界中の人々にとってコミュニケーションおよび情報収集に欠かせないツールとなっている．この背景には，スマートフォンといった携帯通信端末の普及やプロセッサ性能の向上，ネットワークの大容量化などがある．

1990 年代初頭，インターネットを利用する個人や民間企業などのユーザの間では，ダイヤルアップや ISDN による数十 Kbps 程度の低速な回線が利用されていた．このため，ネットワークのトラフィック量も今日と比較すると少なかった．2000 年以降になると，数十 Mbps 程度の ADSL（Asymmetric Digital Subscriber Line）が安価に提供されはじめ，2009 年では ADSL よりも更に広帯域で 100Mbps から 1Gbps を提供する FTTH（Fiber To The Home）による光ファイバー通信の契約者数が ADSL の契約者数を超えた．そして 2012 年 6 月末のブロードバンド(注 1)契約者数は国内合計で 4,182 万契約に至る．これは，10 年前の 2002 年 6 月の 238 万契約のおよそ 18 倍の契約件数である [14, 15]．このような通信基盤の普及に合わせて，安価かつ高速なネットワーク規格である Ethernet が急速に成長した．Ethernet 技術の標準化は，1998 年の Gigabit Ethernet（1Gbps），2002 年の 10GbEthernet（10Gbps）と早い速度で革新がなされており，2010 年 6 月 17 日には IEEE802.3ba による 40Gbps および 100Gbps Ethernet の標準化が承認された [16]．これによって既存ネットワークは 1Gbps はおろか，10Gbps や 100Gbps といった高速な回線が普及し始めている．

更に，携帯通信端末の普及も目覚ましい速度で進んでいる．スマートフォンの登場をきっかけとして，2011 年に我が国の携帯電話の普及率は 100%を超えた．これに伴い，無線 LAN や 3G/4G といった無線ネットワーク環境も急速に整えられ，同時に高速化が進んでいる．我々にとってネットワークは場所や時間を選ぶことなく利用できる環境が整い始めている．

図 1.1 は，総務省の発表する，日本国内の主要 ISP6 社(注 2)が持つブロードバンドサービス契約者の通信トラフィック総量を示すグラフである [1]．国内ネットワークトラフィック総量の推移を観測してみても，ここ 2 年間でトラフィックは 2 倍に増加しており，特に 2011 年以降には携帯通信端末による通信トラフィックが顕著となった．

更には，人々が情報を送受信するのみではなく，M2M（Machine to Machine）や IoT（Internet of Things[17]）といった人以外のデバイスが通信を行うシステムの実現に向けた動きも活発である [18]．このように，インターネット契約者数の増加と回線速度及びモビリティの飛躍的な向上により，今日のネットワークのトラフィック量は飛躍的に増加している．

一方で，近年のインターネット関連技術の急速な進歩によって，web サービスや web コンテン

(注 1)FTTH，DSL，CATV，FWA，BWA 及び 3.9 世代携帯電話の総称をブロードバンドとする
(注 2)インターネットイニシアティブ（IIJ），NTT コミュニケーションズ，ケイ・オプティコム，KDDI，ソフトバンク BB，ソフトバンクテレコム

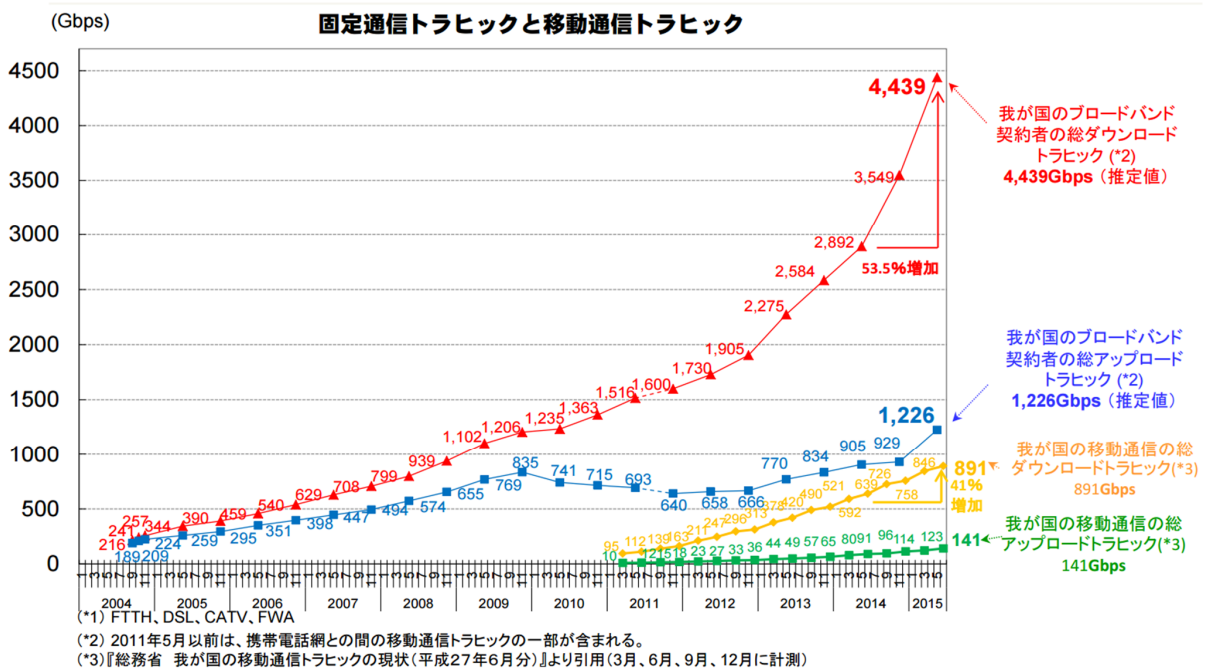


図 1.1: 日本国内におけるネットワークトラフィック総量 [1]

ツといったネットワークアプリケーションは多様化している．回線速度が格段に向上したため，YouTube やニコニコ動画といった動画配信サービスや P2P，クラウドサービスなど，従来とは桁違いに大容量なコンテンツの配信が可能となった．現在のネットワークトラフィックの大部分はこのような動画データまたは P2P パケットにより占められている [19]．また，データ量に限らず，twitter や IP 電話，メッセンジャーといった，データサイズの小さなパケットを短時間に大量に送信することでリアルタイム性を確保するサービスも増加している．

ネットワークアプリケーションの多様化に伴い，ネットワークの用途は多岐に渡るようになった．例えば，近年注目されているスマートグリッドは，電力網と情報網を統合することで電力の自動制御や遠隔からの製品コントロールを，ネットワークを介して可能とする．更に IoT では，電力網に限らず交通や医療，生活のあらゆるものを情報網と結びつけることで，あらゆるものがネットワークを介して互いに制御可能となる．今やネットワークの高度化は，大容量化と並んでネットワークに求められる要素となった．

特に近年では，ルータが Deep Packet Inspection（DPI）といったパケット解析技術を用いて，コンテンツ情報まで踏み込んで通信トラフィックを解析し，その結果を用いることで従来よりも高度なサービスを提供する新たなネットワーク高度化の手段が提案され，広く研究されている [20, 21, 22]．以降では，このように自身が積極的にパケットのコンテンツ情報を解析し，サービスに用いる機能を持ったルータをアプリケーションルータと呼んで扱う．

アプリケーションルータを用いることでセキュリティや QoS，web 行動解析，ネットワークの負荷分散といった様々なサービスに対して従来より高度な機能を提供できる可能性がある．例えば，ネットワークにおける問題として多く報告される個人情報や秘匿情報の漏洩問題に対して，従

来はネットワークを使用するユーザの意識改善や，データ使用手順の複雑化でしか対処することができなかった．これに対して，アプリケーションルータはパケットの情報抽出によってネットワーク内外へと向かうデータの内容を把握できるため，許可なくネットワーク外へと向かう個人情報，秘匿情報をファイルやメールの内容であっても検知し，警告や遮断することができる．更には，ネットワーク内へと侵入する不正なデータ，悪意あるウィルスを検知することで Intrusion Detection System（IDS: 侵入検知システム）として利用することもできる．

また，アプリケーションルータを Web 行動解析に用いる手段が考えられる．アプリケーションルータは，ユーザのネットワーク上での行動を追従することができ，これによってユーザの嗜好を反映したレコメンデーション機能を提供することが可能である．例えば，Web ページの閲覧履歴からオンラインショッピングの商品をリコメンデーションすることが考えられる．従来は，Amazon なら Amazon サーバ内の情報しか利用できないというように，サービスを提供するサーバ内でしか情報収集することができなかったため，このような Web ページを横断した情報収集は不可能であった．更に，アプリケーションルータでは，ユーザ毎のネットワーク上での行動をもとに，ユーザ同士の類似度を計算することも可能である．これによって，様々なネットワーク上の行動からユーザ同士の類似度を算出し，サービスやコンテンツをレコメンデーションすることが可能となる．

従来提案されてきたアプリケーションルータは，パケット処理の複雑さから数 Gbps 程度を達成することが限界であった [2]．従って，現状ではトラフィック量の少ないエッジネットワークにおいてしか情報抽出を行うことができない．これに対し，より広範かつ多岐に渡る情報が流れるコアネットワークにアプリケーションルータを導入することは，多くの意義を含む．

多量のデータを活用することでユーザにとって有意なサービスを提供可能であることは，近年のビッグデータに関する様々な研究からも明らかである [23]．更に，コアネットワークへのアプリケーションルータの導入は，アプリケーションルータのサービスをより広範に提供可能とする．例えば，従来のアプリケーションルータによってネットワーク全体で IDS を実現するためには，エッジネットワーク毎にアプリケーションルータを導入する必要があった．しかしながら，コアネットワークにおいてアプリケーションルータを導入することで，より少数のアプリケーションルータにより同様の機能が提供できる．一方で，前述した個人情報や秘匿情報の漏洩防止サービスなど，エッジネットワークにおいて効果があるサービスも存在する．従って，アプリケーションルータにおけるサービスは，サービスの種類に応じて適切な規模のネットワークにおいて提供されることが望ましい．また，近年の通信トラフィックの増加傾向や，将来的な IoT の実現によるパケット数の増大を考慮すると，今後はエッジネットワークであってもアプリケーションルータによる情報抽出が困難となることが懸念される．アプリケーションルータの処理性能の向上は重要な課題である．

アプリケーションルータにおいて性能ボトルネックとなる処理として，パケットペイロードに対する情報抽出処理がある [2]．情報抽出処理には，ペイロードに対する文字列探索処理のみならず，断片化されたパケットの再構築処理や圧縮パケットの展開処理が含まれる．これは，ネットワークにおいてパケットは分割や圧縮がなされた状態で転送されることがあり，このようなパケットに対応するために境界データや圧縮の展開にも考慮しなければならないためである．一方で，アプリケーションルータが提供するサービスはルータ本来の役割であるパケット転送処理にも影響を及ぼす．具体的には，サービスによってルーティングテーブルやフィルタリングテーブ

ル，QoS テーブルといった様々なテーブルに追加や変更を加えることで，テーブルアクセスの負荷増大を招く．従来のパケット転送ルータではテーブル検索が一番の性能ボトルネックであったが，これによってアプリケーションルータのテーブル検索処理は更にボトルネックとなることが懸念される．

そこで本論文では，アプリケーションルータにおける性能ボトルネックである，情報抽出機構の GZIP 展開処理および文字列探索処理，そしてテーブル検索処理に関して，それぞれを改善するアーキテクチャを個別に提案する．そして，各提案アーキテクチャによってアプリケーションルータの性能ボトルネックが解消され，現在のコアネットワークにおける最大帯域である 100Gbps ネットワーク環境において情報抽出処理およびテーブル検索処理をワイヤレートで実現できることを示す．

1.2 本論文の構成

本論文の構成を図 1.2 に示す．まず第 1 章では，現在のネットワークを取り巻く状況を踏まえ，本研究の目的がアプリケーションルータにおける情報抽出およびテーブル検索ボトルネックの解決であることを述べた．続く第 2 章では，ルータのこれまでの変遷と，近年注目を浴びている，レイヤ 7 まで踏み込んで情報を抽出するアプリケーションルータについて述べる．そして，アプリケーションルータにより実現される高度な機能を持つサービス例を紹介する．第 3 章では，一般的なルータと既存のアプリケーションルータのパケット処理アーキテクチャを，それぞれの関連研究を紹介しながら詳述する．これによって，アプリケーションルータが広帯域ネットワークに対応する上で，情報抽出機構の GZIP 展開処理および文字列探索処理，そしてテーブル検索処理の三点が性能ボトルネックとなることを明らかにする．第 4 章では，第 3 章で挙げた性能ボトルネックの一つ目である，GZIP 圧縮パケットの展開処理に関して，GZIP 圧縮パケットを高速に逐次展開するハードウェアアーキテクチャを提案する．GZIP 展開処理全体を並列処理およびキャッシュ機構を用いたアーキテクチャによりハードウェア化し，更にアプリケーションルータが複数存在する状況下ではピギーバックパケットを併せて用いることで，100Gbps ネットワーク環境における GZIP 圧縮データの逐次展開処理を低回路コストなアーキテクチャにより実現できることを示す．第 5 章では，第 3 章で挙げた性能ボトルネックの二つ目である，パケットペイロードに対する文字列探索処理に関して，完全一致検索処理負荷の削減を目的としたオフローダのハードウェアアーキテクチャを提案する．ラビン・カープ法をベースとした本アーキテクチャにおいて，アプリケーションルータの抽出ルールをもとにアーキテクチャの最適化やフィルタリング機能の実装を行い，100Gbps ネットワーク環境であっても既存の文字列探索手法と本手法を併用することでワイヤレートな文字列探索処理が可能となることを示す．第 6 章では，第 3 章で挙げた性能ボトルネックの三つ目である，テーブル検索処理に関して，近年のテーブル検索処理で採用されている TCAM ベースの手法に併せてフローキャッシュ機構を用いた高スループットかつ低消費電力なテーブル検索アーキテクチャを提案する．本論文では，フローキャッシュ機構にキャッシュエントリを適切に制御するマルチコンテクストキャッシュを併せて実装することで，キャッシュ容量を増やすことなくキャッシュミスを削減できることを示す．これにより，提案機構を用いることで TCAM のみを用いた従来のテーブル検索手法と比べ，高スループットかつ低消費電力にテーブ

ル検索が可能となったことを示す．第 7 章では，本研究の成果をまとめ，アプリケーションルータにおいて 100Gbps ネットワーク環境でのワイヤレートな情報抽出およびパケット転送が可能となったことを示す．

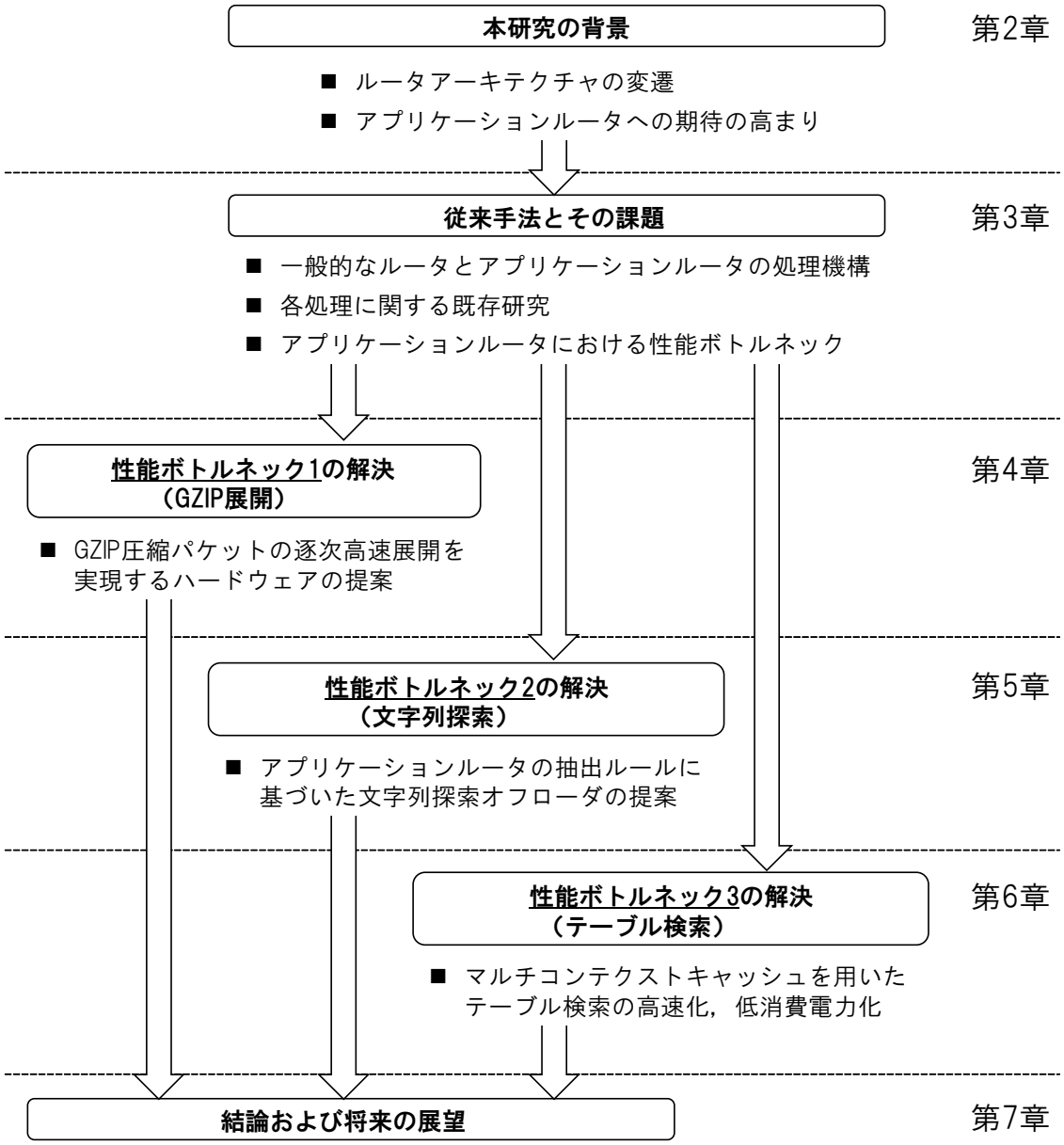

本研究の背景
第2章
■ ルータアーキテクチャの変遷
■ アプリケーションルータへの期待の高まり
従来手法とその課題
第3章
■ 一般的なルータとアプリケーションルータの処理機構
■ 各処理に関する既存研究
■ アプリケーションルータにおける性能ボトルネック
性能ボトルネック1の解決
（GZIP展開）
第4章
■ GZIP圧縮パケットの逐次高速展開を
実現するハードウェアの提案
性能ボトルネック2の解決
（文字列探索）
第5章
■ アプリケーションルータの抽出ルールに
基づいた文字列探索オフローダの提案
性能ボトルネック3の解決
（テーブル検索）
第6章
■ マルチコンテクストキャッシュを用いた
テーブル検索の高速化，低消費電力化
結論および将来の展望
第7章

図 1.2: 本論文の構成

第2章　背景

2.1　ルータの変遷

ルータはインターネットにおいてパケットの転送経路選択の役割を担ってきた．その歴史は，1976 年に米 BBN 社により初の TCP 対応ルータが製造されたことより端を発する [24].

当時のルータは IP プロトコル（当時は TCP プロトコルと呼ばれた）に対応したパケット転送機器として，世界初のコンピュータネットワーク ARPANET における使用を目的に開発された．図 2.1(a) に示すように，パケット転送処理は CPU によってソフトウェア上で行われ，100pps（packet per second）程度の性能であった．1986 年になると，米プロテオン社の ProNET p4200 や米シスコシステムズ社の AGS といった，IP 以外のマルチプロトコルに対応したルータが開発され，ルータの普及が進んだ．この当時のルータ処理性能は 10kpps 程度まで向上している．

そして，1990 年代からルータの処理性能は急激に向上する．まず，1993 年にパケット処理の一部をハードウェア化することで 270kpps の処理性能を達成するシスコシステムズ社の Cisco7000 が発売された．本アーキテクチャでは，図 2.1(b) に示すように，従来の CPU によるコントロールプレーンと，ハードウェア処理によるデータプレーンを分け，パケット転送処理の多くをデータプレーンで行うことにより，処理の高速化を実現した．本アーキテクチャの採用されていた時代では半導体技術が現代と比べ成熟しておらず，ルーティングテーブルなど，パケット転送処理部全体をハードウェア化することはまだ困難であった．そこで，ハードウェア処理とソフトウェア処理を併用する手法が用いられた．例えば，ルーティングテーブル検索においてキャッシュを用いることで，従来のソフトウェアによるルーティングテーブル検索とは異なり，キャッシュ中に経路情報が存在する場合にはキャッシュによって高速な経路決定が可能となった．

1997 年になると，パケット処理専用ハードウェアであるラインカードをクロスバースイッチにより複数接続した分散アーキテクチャの Cisco12000 が発売され，本製品の処理性能は 10Mpps にまで達した．本アーキテクチャの概要を図 2.1(c) に示す．本アーキテクチャでは，前述したコントロールプレーンをルートプロセッサカードに，データプレーンをラインカードに実装し，それぞれをクロスバースイッチにより接続する．ラインカードは，Network Processor（NP）と呼ばれるパケット処理専用のチップを内蔵し，NP によって全てのパケット処理を行う．また，一つのルータシャーシに複数枚のラインカードを接続することで，パケットの並列処理が可能となった．一般的には 4 枚から 32 枚のラインカードを一つのシャーシに接続可能であり，これにより高いパケット転送性能が実現される．ルートプロセッサは，カードの接続されたルータシャーシ全体の管理や Command Line Interface（CLI)，ルーティングプロトコルの実行やテーブルの作成など，パケット単位の処理とは異なる複雑な処理を担う．各ラインカードおよびルートプロセッサカードはクロスバースイッチにより接続される．クロスバースイッチの概要を図 2.2 に示す．クロス

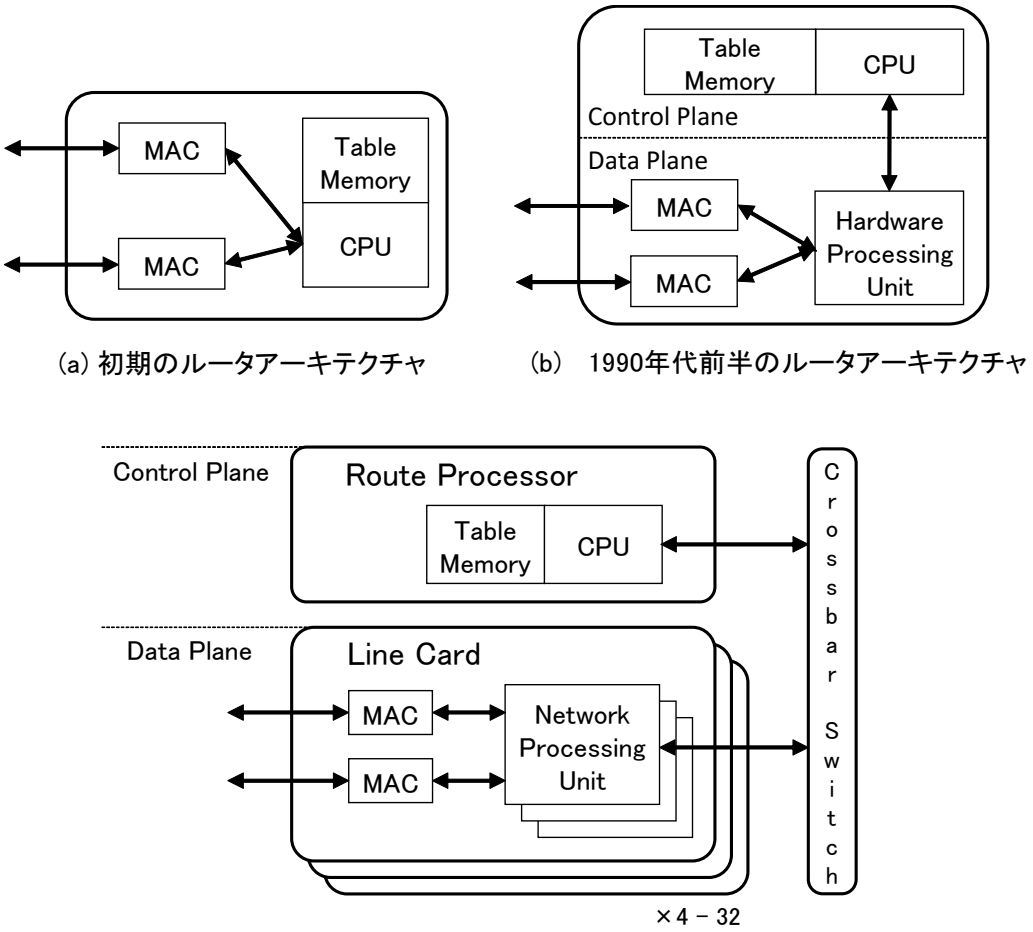
表 2.1: 代表的なハイエンドルータ

ベンダ	機種名	交換回線数	スループット	出荷年
Juniper [25, 26, 27]	T1600	40Gb Eth × 16 or 100Gb Eth × 8	1.6Tbps	2007 年 10 月
Juniper [25, 27, 28]	T4000	40Gb Eth × 16 or 100Gb Eth × 16	4Tbps	2011 年 6 月
Cisco [29, 30]	CRS-3 16 スロット	10Gb Eth × 320 or 100Gb Eth × 16	4.48Tbps	2010 年 3 月
AlaxalA [31]	AX7816R 16 スロット	10Gb Eth × 32 or OC-192 × 16	768Gbps	2004 年 10 月
AlaxalA [32]	AX8632R 32 スロット	10Gb Eth × 192 or 100Gb Eth × 16	6.4Tbps	2013 年 9 月

バースイッチは，バスを網目状に配置したクロスバーに対し，クロスバーの交点にあるスイッチを動的に切り替えることで，各ラインカードおよびルートプロセッサカード間の通信路を要求に応じて動的に構築する．図では，各ラインカードおよびルートプロセッサのの入力ポートが図下側のバスに接続され，出力ポートが図右側のバスに接続されている様子を示している．クロスバースイッチでは，各入力ポートの出力先が被らない場合においては，最大で一辺のバス数と同数のデータを同時転送することができる．これによって，高いデータ転送容量を確保することができる．Cisco12000 のアーキテクチャは現在のルータにおいても一般的に採用されているアーキテクチャである．

その後もインターネットの急速な成長に伴い，ルータは処理性能を向上させてきた．1998 年には，OC-48（およそ 2.4Gbps）ポートを持ち 400Mpps を達成する日立製作所の GR2000 が，また 2004 年には，OC-768（およそ 40Gbps）ポートを持ち，複数筐体を組み合わせることで 100Gpps まで達成可能なシスコシステムズ社の CRS-1 が発売された．このように，20 世紀においてルータの進化の方向性は，いかに大容量のトラフィックの経路選択を可能とし，転送するかに注力していた．21 世紀初頭にかけて，トランジスタの小型化や動作周波数の向上を受けてプロセッサの性能が向上し，また Ternary Content Addressable Memory（TCAM）のような新たなメモリ技術が開発されるにつれ，ルータのパケット処理機構は洗練されてきた．この結果，パケット処理のワイヤレート化，大容量化を達成することが比較的容易となった．特に近年では，ネットワーク回線の成長速度が緩やかになってきており，ルータに求められる処理性能の達成は困難ではなくなった．2004 年から現在にかけて発売された主なハイエンドルータを表 2.1 に示す．2010 年に標準化の完了した 100Gbit Ethernet であるが，表 2.1 に示すように，既に 2007 年において Juniper 社の T1600 では多数収容可能となっている．

一方で，第 1 章で述べたようにインターネットの用途は多様化してきた．インターネット初期のメールや web ページ閲覧といったサービスから，IP 電話やオンラインショッピング，ファイルの管理など様々なサービスがネットワークを介して行われるようになった．これに伴い，ネットワーク上のコンテンツも多様化した．動画トラフィックのようなデータサイズが GB を超えるデー



(a) 初期のルータアーキテクチャ　(b) 1990年代前半のルータアーキテクチャ

(c) 1990年代後半以降の分散アーキテクチャ

図 2.1: ルータアーキテクチャの種類

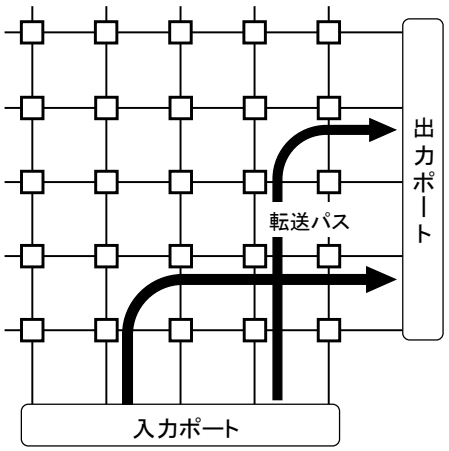


図 2.2: クロスバースイッチの構造

タや，センサデータのような小規模なデータサイズのデータ，またアカウント情報やネットワークストレージに保存した社内の重要なファイルといった漏洩を防止すべきデータ，IP 電話のようなネットワークの遅延の影響を大きく受けるデータ，悪意あるユーザによるネットワークアタックを引き起こすデータなど，様々な目的のデータがネットワーク上を混じり合い流れている．このような中で，ルータには，大容量化のみならずセキュリティやトラフィック制御といったパケット転送以外の機能が必要とされるようになった．

そこで，ルータが経路選択やパケット転送のみならず，パケットの解析を行うことでこれらのニーズに応える手段が提供され始めた．例えば，ACL（Access Control List）を用いたパケットフィルタリングではリストに書かれたルールを参照し，該当するヘッダ情報を持つパケットに対し許可や拒否といったアクションを施す．また，Diffserv 機能では，パケットヘッダの IP アドレスやポート番号をもとにパケットフローの転送優先度を決定し，アプリケーション毎に異なる QoS（Quality of Service）を保証する．ルータがパケットのヘッダ情報をアプリケーションに用いることで，経路選択やパケット転送機能だけでなく，より多くの高度な機能を提供することが可能となった．このような中で，近年では，パケットヘッダのみならず，OSI 参照モデルのレイヤ 4 より上層，特にレイヤ 7 で表されるペイロードまで踏み込んで情報を解析するアプリケーションルータが注目され，研究されている．

2.2 アプリケーションルータ

従来のルータは，レイヤ 4 までのパケットヘッダに付随した情報を基にパケット転送処理を行っており，より上層に含まれるデータを参照，加工することはなかった．このような背景として，パケット転送処理にはペイロードの情報が不要であるという点，またパケットヘッダが 100Byte 以下であるのに対し，ペイロードは最大で 1500Byte 程度となるため，処理負荷が増大するという点が挙げられる．一方で，前節で述べたように，近年の半導体技術の向上やアーキテクチャの洗練に伴う処理性能の向上から，ルータのパケット転送に要求される性能を達成することは困難ではなくなった．そして，レイヤ 4 より上層に含まれるデータは，解析してアプリケーションに用いることで，ネットワークユーザに対してより高度な機能を提供できる可能性がある．例えば，レイヤ 7 ペイロードに含まれるウィルスを検知し，ネットワーク経路において遮断することができる．更に，近年では IoT の実現に向けたセンサデバイスの通信においても，ペイロードにセンサデータが含まれるため，ルータがセンサデータを把握することで，閉じたネットワーク内においてセンサデバイスを制御することも可能となる．そこで，近年では，レイヤ 4 より上層のデータを解析してアプリケーションに用いる，アプリケーションルータの研究，開発が多くなされている．

2.2.1 アプリケーションルータの種類

本論文では，ルータの機能による分類および代表的なアプリケーションルータを図 2.3 に示した．データ量の多いコアネットワークにおけるパケット処理に対応したコアルータに対し，パケット解析機能にフォーカスしたルータはパケット処理性能で劣るため，主にエッジネットワークで

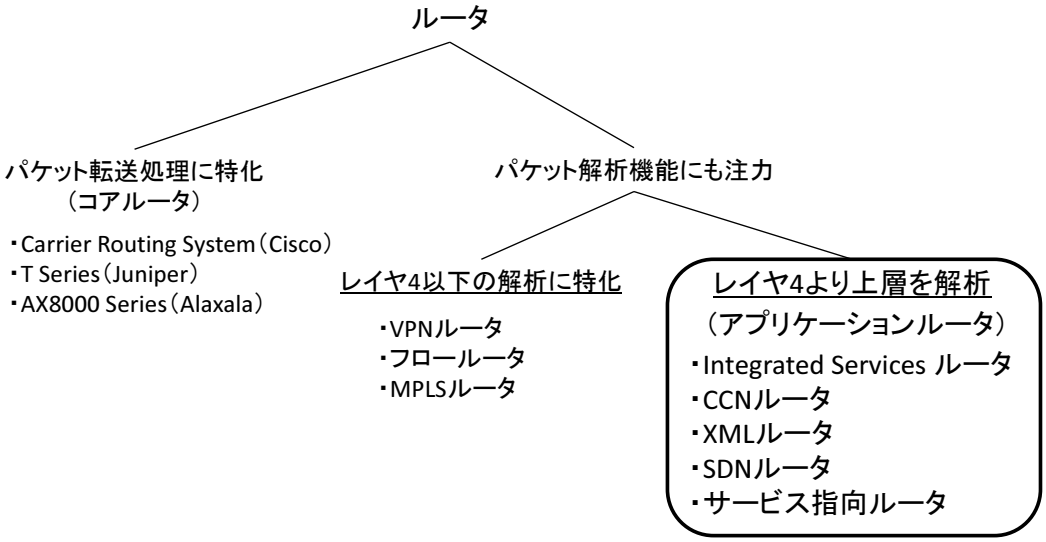


図 2.3: ルータの主な分類

の使用を目的として開発されてきた．以降，図 2.3 をもとに代表的なアプリケーションルータについて説明する．

Integrated Services ルータ

アプリケーションルータの先駆けであり，現在も開発の進められているルータにシスコシステムズ社の Integrated Services ルータ（ISR）がある [33]．ISR は，主にエッジネットワーク向けに，パケット転送機構と併せて高度なサービス機能を提供するルータとして開発がされた．その機能は，Voice Over IP（VoIP）データの制御や，Virtual Private Network（VPN）における暗号化，ファイアウォールの高機能化，侵入検知/防止システムの実現など多岐に渡る．更に，ISR に Cisco Application Extension Platform（AXP）を搭載することで，システム設計者，利用企業等が Cisco の提供する API や SDK を用いて ISR 上で自由にアプリケーションを設計，利用することが可能となる．

Content-Centric-Networking ルータ

従来の IP によるパケット経路制御をレイヤ 7 プロトコルで補完した，コンテンツ指向ルーティングを実現する Conten-Centric-Networking（CCN）ルータが提案された [34, 35]．コンテンツ指向ルーティング [36] では，パケットの宛先は IP アドレスではなく，“/sample.com/papers/doctor-thesis/chapter1.pdf”といった階層型のコンテンツ名をもとに決定される．従来のルーティングが IP アドレスによる 1 対 1 の通信であったのに対し，コンテンツ指向ルーティングでは同一コンテンツを持つ複数のホストの中から自分に最も有利なホストを選択することができる．これによって，ネットワーク内にコンテンツを分散コピーするキャッシュ技術の適用と併せて，従来の 1 対 1 通信によるコンテンツ要求の集中を解消し，より近いローカリティを持つホストとの通信によってコンテンツをダウンロードすることが可能となる．

CCN ルータは，このようなコンテンツ指向ルーティングの基となるコンテンツ抽出機構や，抽

出したコンテンツ自体をキャッシュする Content Store，コンテンツを持つホストへの経路を決定する従来のルーティングテーブルに該当する Forwarding Information Table といった機構を持ち，従来の IP によるパケット転送の基準に縛られない，コンテンツ指向なルーティング機能を提供する．

Extensible Markup Language ルータ

近年の web サービスにおいて，Extensible Markup Language（XML）形式のデータを用いたアプリケーションが増加している．XML とは，タグを用いてデータの構造化，階層化を可能としたデータ形式である．このような XML 形式のデータを解析し，前述したコンテンツ指向ルーティングを基にデータを転送する XML ルータが提案された [37].

XML ルータは，パケットのレイヤ 7 に含まれる XML メッセージを解析し，XPath[38] を用いて XML ベースのルーティング用データに整形する．XML ルータがルーティングを行う際は，整形されたルーティング用データを基に，コンテンツ指向ルーティングの手法を用いてデータを転送する．XML ルータによって，web アプリケーションに適したルーティングが容易に実現できる．

Software Defined Network ルータ

ネットワークの構成や機能，性能などをソフトウェアによって動的に変更可能な Software Defined Network（SDN）ルータがある [39]．これまでのネットワークは，一旦構築するとほとんど変更が生じないスタティックなものとして扱われてきた．従って，ネットワーク構成に変更があった場合には，ネットワーク管理者がルータ毎にネットワーク設定等を更新していく必要があった．この際，ルータではベンダ独自の形式が採用されているため，複数ルータを一括して設定を更新することは困難であった．更に，近年のネットワークでは仮想サーバの普及やクラウドアプリケーションの増加に伴い，ネットワーク構成が頻繁に更新されることが多くなった [40]．そこで，ネットワークの設定やアプリケーション機能をソフトウェアにより仮想化し，統一されたプロトコルのもとで設定可能とする SDN が注目されてきた．SDN ルータでは，OpenFlow[41] といった SDN コントローラとして動作する制御プロトコルを用いて，複数ルータの制御を一括に行うことができる．SDN ルータによって対応可能なアプリケーションは，負荷分散や侵入検知/防止システム等，レイヤ 4 より上層の情報を用いたアプリケーションも対象となる．

サービス指向ルータ

SDN ルータと同時期に，より広範囲なサービスに対応するための手段としてサービス指向ルータが提案された [42]．サービス指向ルータはパケットのペイロードをルールに基いて抽出し，抽出結果をデータベースに蓄え，その情報を用いてサービスを提供する．特にサービス指向ルータは，ユーザによる柔軟な抽出ルールの設定やデータベースに格納されたデータの加工を可能とし，ユーザが新たなサービスを作り出すことができる．また，サービス指向ルータでは抽出データのプライバシを考慮し，ハードウェアによる匿名化機構が提案されている [43, 44]．これにより，ユーザが自由にデータを解析するためのインフラストラクチャがサービス指向ルータによって構築される．サービス指向ルータの提供するサービスは，負荷分散や侵入検知/防止システムといったサービスに限らず，web 行動解析 [45, 46] や CDN（Contents Delivery Network）への応用 [47, 48] など，広範囲に渡る．

2.2.2 アプリケーションルータのサービス例

web サービス，web コンテンツの多様化したネットワークにおいて，パケットペイロードは多くの新たな付加価値サービスに提供可能な情報を含んでいる．例えば web ページのタイトルからメールの内容，センサネットワークにおけるセンサの値など，様々な情報がペイロードに存在する．アプリケーションルータは，パケット転送と同時にこのような情報の抽出を行う．更に，抽出情報を基にルーティングの変更やパケット転送の制御を行うことで，高度なサービスを提供する．本項では，このようなアプリケーションルータによる経路変更やパケット転送制御を用いたサービス例として，セキュリティへの応用，CDN におけるキャッシュサーバへの誘導，細粒度の高い QoS 制御について紹介する．また，アプリケーションルータの情報抽出および解析機能を活かした，よりユーザフレンドリーなサービス例として，ユーザの web 行動解析を利用したレコメンデーション機能について紹介する．

2.2.2.1 セキュリティへの応用

アプリケーションルータの情報抽出機能，経路変更機能を応用することで，様々なセキュリティ確保の手段が実現される．これらの方法を図 2.4 に示す．

パターンを増し複雑化するネットワークアタックに対するセキュリティ手段として，ネットワーク上でトラフィックを解析し，不審な通信を検知するネットワーク型侵入検知システム（Network Intrusion Detection System: NIDS）が広く注目されている [49]．NIDS ではトラフィックの不正検出と異常検出を共に実現できる．まず不正検出では，既知のアタックの特徴（シグネチャ）をまとめたルールセットを文字列探索を用いてパケットに照合することで，アタックを検出する．不正検出により，ブラックリストに登録されている web ページへのアクセスや，バッファオーバフローを狙った意図的に長い文字列など，既知のアタックを検知できる．一方で異常検出では，ログイン時刻やトラフィック状況など様々な条件に閾値を設定し，閾値を越えたパケットを異常な通信として検知する．これによって，例えば DDoS 攻撃といった従来検知することが困難であったアタックに対しても，レイヤ 7 情報の類似したパケットの時間的統計をとることで検知できる可能性がある．

NIDS において不審な通信を検知した場合のアクションは，システム管理者へ警告を行うのみであった．これでは，システム管理者が対処する頃にはアタックが対象へ影響を及ぼしている可能性がある．そこで，特に近年では侵入防止システム（Intrusion Prevention System: IPS）の適用が注目されている．IPS は，ネットワーク経路上で不正なパケットを精査し，遮断することによって不正な侵入自体を阻止する．IPS では，通信路上にインライン接続された機器においてパケットを精査する必要があり，処理の複雑さから処理の大容量化が困難であった．

これに対して，図 2.4(a) に示したような，アプリケーションルータを用いたアタック検知処理の負荷分散手法が考えられる．本手法では，アプリケーションルータはアタックの簡易な判断のみ行う．そこで，より詳しく検査する必要があるパケットに関しては，ルーティング経路を変更し，精査サーバへと転送する．既存の IPS では，通信路上において全パケットを精査するためパケット転送の停滞が問題視されていたが，本手法によってパケット精査のボトルネックを解消することができる．

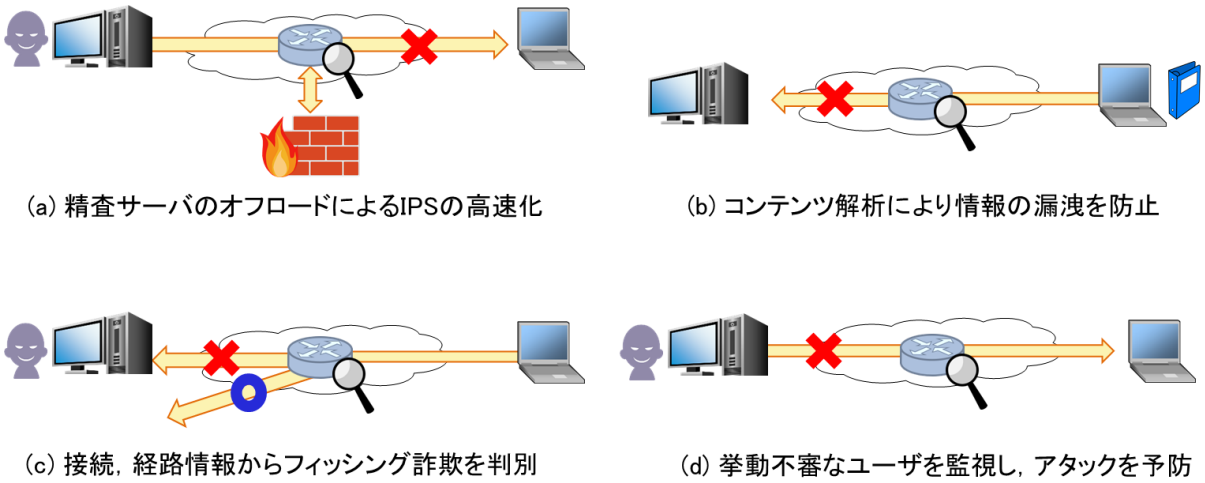


図 2.4: アプリケーションルータを用いたセキュリティサービス例

アプリケーションルータの情報抽出機構を用いたセキュリティ技術は，アタック検知以外の用途にも使用可能であると考えられる．組織内ネットワークレベルにおいてゲートウェイに単独のアプリケーションルータを導入すれば，図 2.4(b) に示したように機密ファイルや個人情報といった対外秘情報の漏えいを防ぐことが期待できる．近年では，ワームやウィルスに感染した組織内 PC から個人情報が漏洩する問題も多々発生しており，アプリケーションルータがこれらの情報を抽出し対策することで，情報漏えいの防止が期待できる．

更に，図 2.4(c) に示したように，アプリケーションルータを用いたフィッシング詐欺サイトの対策手段が考えられる．フィッシング詐欺サイトの web ページは，本来の web ページと同じ情報を自由に生成可能であるため，コンテンツに着目して判別することは難しい．アプリケーションルータは，ある銀行のサイトは X 番インタフェースに接続されているといった，物理的な接続情報を管理することができる．この情報を用いることで，本来の番号ではないインタフェースから出ていこうとするフィッシング詐欺サイトのパケットを検知する．加えて，各アプリケーションルータが協調し，パケット配送の際にパケットを重畳して符号化し，特定のキーチェインで復号できることを確認すれば，パケットの配送経路が保証できる．このような情報のトレーサビリティをネットワークが持つことで，フィッシング詐欺サイトのみならず，不正なコンテンツやアタックパケットも検知できる可能がある．

また，図 2.4(d) に示したように，後述するインターネットにおけるユーザの行動履歴解析をセキュリティに応用する手段が考えられる．過去に検出したアタックユーザの行動履歴から，類似した挙動をするユーザを検知することで，事前に対策を講じることが可能となる．類似アクセスの確認機能は，例えばオンライン機械学習向け分散処理フレームワーク Jubatus[50] を利用することで実装できる．

2.2.2.2 CDN におけるキャッシュサーバの決定

近年，インターネットにおける web コンテンツダウンロードの高速化手法として CDN（Contents Delivery Network）が広く用いられている．インターネットでは，通信を行う二者間の物理的な距離をユーザが意識することはない．しかしながら，地理的に遠いホストからコンテンツをダウンロードする場合，ダウンロードに時間がかかり，またトラフィックの混雑を招く．そこで，CDN は web コンテンツを各地に用意したキャッシュサーバにコピーすることで，ユーザにとって有利なキャッシュサーバからのダウンロードを可能とさせる．ユーザは CDN の恩恵を受けることで，地理的に近く，負荷の少ないサーバから高速にコンテンツをダウンロードできる．コンテンツの提供元に対しても，キャッシュサーバがコンテンツダウンロードを肩代わりすることで，ダウンロード負荷の軽減が見込める．更には，大局的な観点から見ても，CDN によりダウンロード経路が最適化されることで，ネットワークを流れるトラフィック量を削減できる．YouTube といった動画ストリーミング配信サービスや Google といった web ブラウジングサービスなど，様々なネットワークサービス企業が既に CDN を導入しており，CDN の需要は年々高まっている．

従来の CDN は，主に web コンテンツ提供企業の DNS 情報を書き換え，コンテンツダウンロード要求パケットを CDN 提供先へとリダイレクトし，CDN 提供先の DNS 情報を用いることで実現される．従って，その通信は，ユーザからコンテンツ提供元への DNS 問い合わせ，コンテンツ提供元から CDN 提供先へのリダイレクト，CDN 提供先からユーザへの応答という順路を辿る．これに対して，DNS の役割をアプリケーションルータが肩代わりする手法が研究されている [47, 48]. 本手法の概要を図 2.5 に示す．アプリケーションルータは，情報抽出機構を用いることでトラフィックからコンテンツダウンロードのリクエストパケットを抽出できる．ここで，アプリケーションルータが近隣のキャッシュサーバの情報を持つことで，リクエストパケットに対してルーティングを変更し，より有利なキャッシュサーバへと誘導することが可能となる．この際，近隣のキャッシュサーバの情報は，より上層のネットワークに位置するアプリケーションルータがコンテンツ提供企業または CDN 提供企業からキャッシュサーバ情報を受け取り，下層のネットワークに位置するアプリケーションルータに対し適切に配布することで把握可能となる．本サービスにより，コンテンツ提供元を介することなく，ネットワーク経路上において高速にキャッシュサーバの IP アドレスを得ることができる．また，アプリケーションルータによりネットワークの混雑度や障害を考慮したキャッシュサーバの決定が可能となる．

よりユーザに近いロケーションからのコンテンツ提供を可能とする技術として Google Global Cache（GGC）がある [51]．GGC では，Google の提供するキャッシュサーバを，Internet Service Provider（ISP）やネットワーク管理者が自らの管理するネットワーク内に設置する．これによって，Google データセンタが管理するコンテンツを，管理内ネットワークのユーザに高速に配信することが可能となる．Google は，通常 70%から 90%程度のコンテンツトラフィックを GGC から配信可能であると述べている．しかしながら，現状では GGC の導入は Google からの招待によってのみ可能となっており，ISP やネットワーク管理者が自由に CDN を導入することはできない．そこで，アプリケーションルータ自身がコンテンツをキャッシュする手段が考えられる．ネットワーク上においてコンテンツは必ずルータを経由することから，ルータにはコンテンツが受動的に集まる．そこで，コンテンツがアプリケーションルータを通過する際に，コンテンツを抽出，キャッ

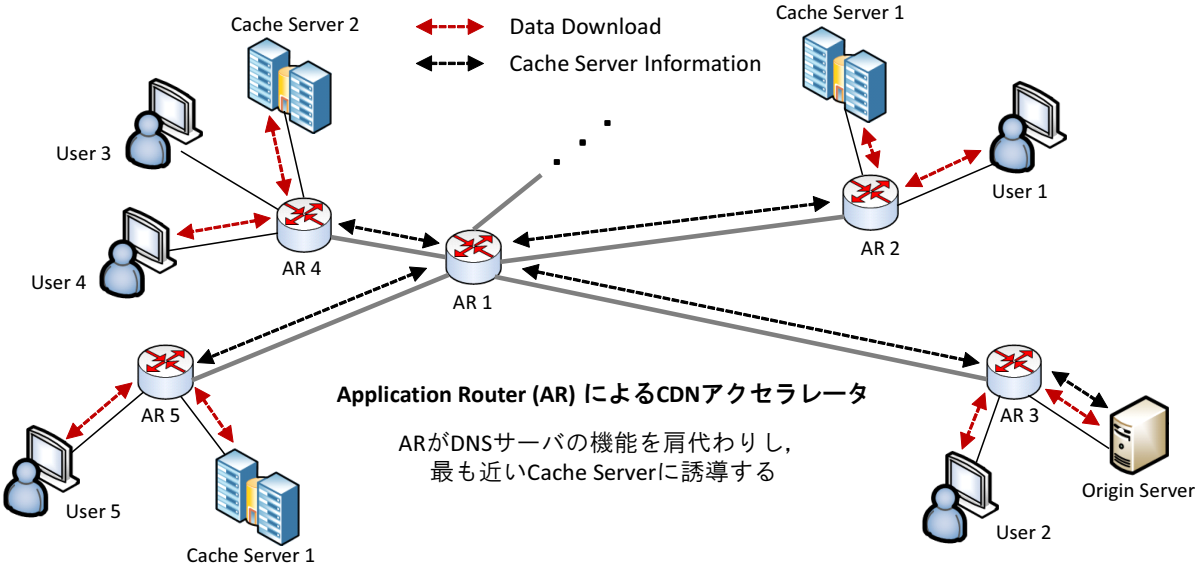


図 2.5: アプリケーションルータによる CDN アクセラレータの概要

シュし，ルータ自体がキャッシュサーバとなることで，ローカリティの高い CDN サービスが提供できる．本手法は，ISP やネットワーク管理者が自由に CDN を導入でき，なおかつキャッシュ対象は Google の提供するコンテンツに限らないため，GGC よりも柔軟な CDN 技術となりうる．このように，アプリケーションルータが CDN を担うことで，既存技術のアクセラレータとしても，新たな CDN プラットフォームとしても利用することが可能となる．

2.2.2.3 細粒度の高い QoS 制御

QoS（Quality of Serivice）とはサービスの品質を意味する．具体的には，通信データの内容に応じて優先度を決定し，データを優先的にもしくは意図的に遅らせて転送する．ストリーミング配信や IP 電話といったサービスでは，ネットワークが混雑し通信データがまばらに到着すると動画や音声が途切れて再生されるため，快適な利用が困難となる場合が多い．一方で，コンテンツダウンロードや web ブラウジングといったアプリケーションでは，データ転送が多少遅延したとしても，ユーザが受ける影響は前述したアプリケーションよりも少ない．そこで，前者のデータ転送に高い優先度を与え，後者のデータ転送優先度を下げることで，全体的なネットワークの利便性は向上できる．また，一つのアプリケーション内においても，例えば IP 電話における緊急電話時の優先度を通常電話時よりも高くする，動画視聴アプリケーションにおいて非課金ユーザより課金ユーザに高いデータ転送優先度を与えるといった，異なる QoS を提供することで利便性の向上が期待できる．

従来，QoS の実現は，アプリケーション提供元のサーバにおけるトラフィック制御か，ネットワーク機器によるレイヤ 4 以下の情報を用いたトラフィック制御の 2 種類であった．前者の例としては，アクセスの集中するページに対し，会員登録済みのユーザに優先してアクセス権限を与え

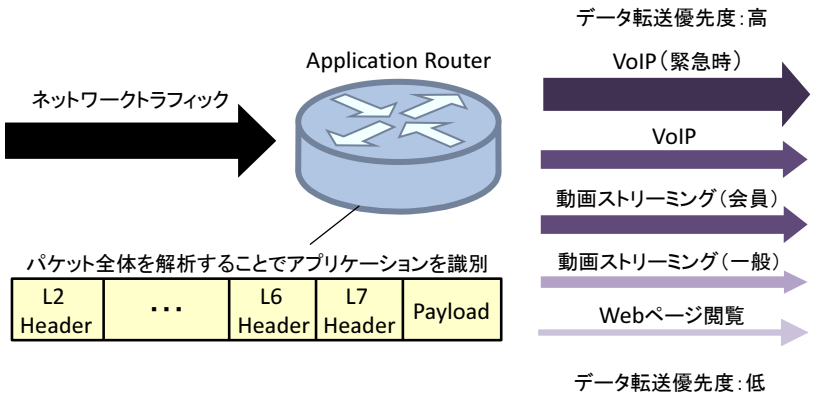


図 2.6: アプリケーションルータを用いた柔軟な QoS の実現

るといった制御がある．このように，アプリケーション提供元による QoS 保証はそのアプリケーション内でしか適用できず，web ブラウジングに対して VoIP のデータ転送優先度を高くするといったアプリケーション毎の QoS は提供できない．また経路上のネットワークが混雑している場合などは QoS が保証できないといった問題点があった．

一方で，後者の場合にはアプリケーション毎の QoS を保証できる．この場合のトラフィック制御は，パケットの主にレイヤ 4 のポート番号をもとにアプリケーションを識別し，QoS を保証する．そのため，ネットワーク機器では前者のような同一アプリケーション内で異なる QoS を提供することが不可能であった．また，近年ではポート番号を動的に変更するアプリケーションが数多く見られ，このようなアプリケーションに対する QoS を保証できないといった問題点があった．

これに対し，アプリケーションルータが抽出した，より高層の情報を用いてアプリケーションを識別する手段が考えられる．本手法の概要を図 2.6 に示す．従来はポート番号によって IP 電話などのマルチメディア通信を識別していたが，近年はポート番号を動的に変えるようなアプリケーションが多く用いられている．そこで，例えばレイヤ 6 のセッション確立プロトコル SIP（Session Initiation Protocol）の情報を解析することで，ポート番号によらずアプリケーションの識別が可能である．また，レイヤ 7 まで踏み込んで情報を抽出することで，アプリケーション内における QoS の差異を実現できる可能性がある．例えば，IP 電話アプリケーションにおいて，通常時の通話と緊急時の通話のデータ先頭に別々の識別情報を付加することで，アプリケーションルータが通常時と緊急時を判断し，異なる優先度を与えることが可能となる．

また，周囲のアプリケーションルータと協調することで，より高度に QoS を実現する手段が考えられる．従来の QoS 保証はネットワークにおける一点でのトラフィック制御であった．従って，当然のことながら経路途上の別回線が混雑していた場合には快適なデータ転送は実現されない．これに対し，アプリケーションルータが協調しネットワークの混雑度を知らせ合うことで，ルーティングを変更し，混雑の少ない経路を選択することが可能である．ネットワークの混雑度を考慮した QoS によって，ネットワークにおける線でのトラフィック制御が期待できる．

2.2.2.4 サイト間を横断した情報収集によるレコメンデーション機能

ネットワークにおけるコンテンツの多様化はユーザにとってコンテンツ選択の幅が広がる反面，情報氾濫による混乱を招く側面を持つ．このような問題に対し，近年ではレコメンデーション機能を活用し，ユーザの趣向に合わせたコンテンツを推薦するサービスが多く見られる．例えば，オンラインショッピングサービスを提供する Amazon では，ユーザが閲覧した商品履歴と同じような履歴を持つ，他のユーザが閲覧した商品をレコメンデーションする．同様に動画配信サービス YouTube では，ユーザの動画閲覧に際し，同じ動画を観た別ユーザの多くが観ている動画をレコメンデーションする．無数に存在するコンテンツの中から，ユーザが興味を持つであろうコンテンツをレコメンデーションすることで，サービスの利便性は飛躍的に向上する．

従来のレコメンデーション機能は，コンテンツの提供元が自身のサーバ内の情報をもとに計算を行うことで提供されていた．従って，Amazon は Amazon を利用したユーザの行動情報のみに，YouTube は YouTube を利用したユーザの行動情報のみにもとづいてレコメンデーション機能を提供していた．そのため，コンテンツ提供元の情報量にレコメンデーションサービスの質が左右されるという問題があった．

これに対して，アプリケーションルータは，取得した情報を基にレコメンデーションの計算を行うことで，コンテンツの提供元によらない横断的なレコメンデーション機能が提供できる．例えば，オンラインショッピングサービスにおいてアプリケーションルータを利用することで，ユーザは Amazon のみではなく，楽天市場など他のオンラインショッピングサービスを利用したユーザの閲覧履歴まで用いたレコメンデーション機能を受けられる．また，ユーザの YouTube の閲覧履歴を基に興味のあるジャンルを分析し，オンラインショッピングにおいて商品を推薦するコンテンツ横断的なレコメンデーション機能の提供が可能となる．このようなレコメンデーション機能は，将来的な Internet of Things（IoT）が実現されるネットワークにおいて，道路交通情報を用いた道路空き情報のレコメンデーションや，医療情報を用いた病気予防法のレコメンデーションなどにも適用可能であり，多くの可能性を秘めている．更に，アプリケーションルータによるレコメンデーション機能はネットワークのローカリティを反映する．これは，例えば日本国内のユーザが興味を持つコンテンツが日本製の製品や日本語のコンテンツであるのに対し，アメリカ国内のユーザが興味を持つコンテンツはアメリカ製の製品や英語のコンテンツであるといった地域性を考慮し，より指向性の高いレコメンデーションが可能であることを示す．

更に，石田らはアプリケーションルータにおいてユーザの web ページ閲覧情報を収集することでユーザの行動を解析できると述べている [2]．石田らの行った web ページ閲覧の履歴解析結果を図 2.7 に示す．図 2.7 は，あるユーザがオンラインショッピングを目的として web ページの閲覧を行った際の，web ページと滞在時間の推移をアプリケーションルータで計測したグラフである．閲覧した順に web ページに割り当てた ID を縦軸としている．この結果より，まずユーザはサイト A においてある商品に興味を引かれページ閲覧を繰り返し，次にサイト B でも同様の行動をしていることがわかる．そして，サイト C で商品を閲覧した後，最終的にサイト A において商品を購入したことが推測できる．このようにユーザの行動を解析することで，滞在時間や閲覧履歴を考慮したレコメンデーションが可能であると石田らは述べている．実際に，このような web ページの閲覧回数や滞在時間を考慮したアプリケーションルータのレコメンデーションサービスは増

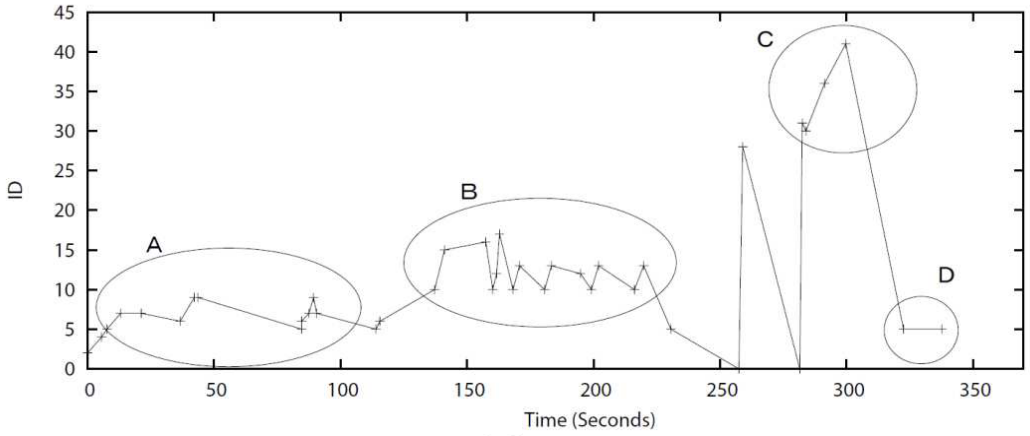


図 2.7: 石田らによる web ページ閲覧に関する履歴解析 [2]

田らの研究により実現されている [45, 46]．増田らは，ユーザ毎の web ページ閲覧履歴をもとにピアソン相関係数および協調フィルタリングを用いて，あるユーザが特定 web ページに興味を示す度合いを予測し，興味を持つと予測される web ページのレコメンデーションを行った．AB テストにより提案されたレコメンデーションサービスを評価したところ，レコメンデーションに従うことでユーザの満足度が向上することが示されている．

第3章　アプリケーションルータの処理機構および関連研究

近年のネットワークは 1Gbps や 10Gbps といったギガオーダの回線が普及しており，コアネットワークにおいても 100Gbps の高速な回線が用いられ始めている．これに対して，従来のアプリケーションルータは処理の複雑さからスループットの向上が困難であった [2]．これにより，今後のネットワークではアプリケーションルータによるサービスの提供が困難となりかねない．アプリケーションルータのスループット向上は極めて重要な課題である．

そこで本章では，アプリケーションルータが持つ処理機構について詳述し，ボトルネックとなる部分を明らかにする．まず，一般的なルータにおけるパケット処理プロセスに基き，一般的なルータが持つパケット処理機構について述べる．次に，アプリケーションルータ全体のパケット処理プロセスを示し，アプリケーションルータが持つ，一般的なルータとは異なる処理機構とその既存研究について詳しく述べる．最後に，これらのまとめとして，アプリケーションルータが抱える問題点や現状で得られるスループットなどを分析することで，アプリケーションルータのスループット向上の足がかりとする．

3.1　一般的なルータのパケット処理機構

本節では，アプリケーションルータへの理解の足掛かりとして，まず一般的なルータの処理プロセスと，そのアーキテクチャについて述べる．一般的なルータのパケット処理プロセスについて，文献 [52] を参考として図 3.1 に示した．ルータでは，図 3.1 に Parsing で示される処理ブロックによって，受信したパケットの構文解析を行い，パケットデータを各ヘッダ，更にはヘッダ内の各フィールドに分割する．以降の処理ブロックは，Parser で分割されたデータの中で必要な部分のみを受け取り処理する．ルータに求められる主要な機能はルーティング機能，プロセッシング機能，フォワーディング機能の三つに分類できる．まず，ルーティングでは，到着パケットに対して宛先ホストに到達するためのネットワーク経路が選択される．ここでは，主に出力ポートおよびネクストホップ IP アドレスの探索，ネクストホップ MAC アドレスの探索などが行われる．次にプロセッシングにおいて，到着パケットのフィルタリングや情報抽出といった，パケット転送とは直接関係のない拡張機能に関する処理が施される．最後にフォワーディングにおいて，ルーティングで決定した出力ポートからパケットを転送するための処理が施される．ここでは，主に転送の際のポリサー，シェービングといった QoS に関するキューイング処理やフィルタリング処理といったパケット転送制御が行われる．以降では，図 3.1 をもとに構文解析，ルーティング，プロセッシング，フォワーディングの各アーキテクチャについて順に述べる．

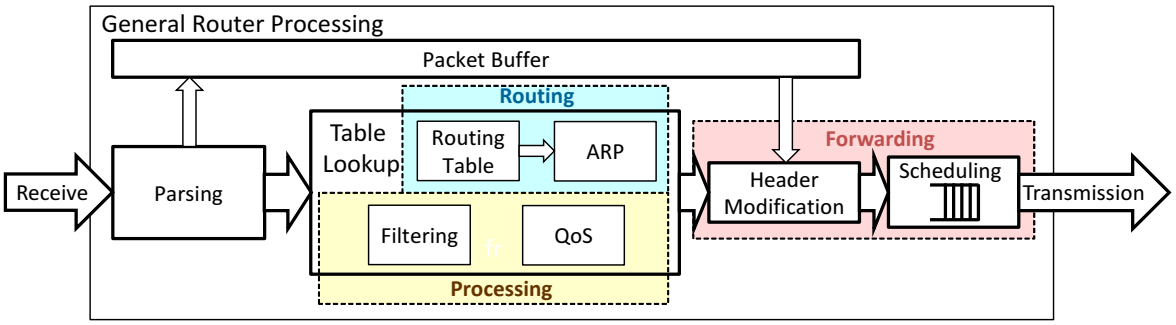


図 3.1: 一般的なルータにおけるパケット処理プロセス

3.1.1 構文解析機構

ルータの受信するパケットは，区切りのない一連のビット列によって構成される．従って，このままではパケットデータから IP アドレスやポート番号，ペイロードといった意味のある情報を識別することはできない．そこで，ルータでは，構文解析機構によって一連のビット列を意味のあるブロック単位に分割する．

インターネットを用いて通信を行う際，コンテンツ情報などの通信内容はパケット単位にデータ分割され転送される．この際，データは様々な通信プロトコルによって何層にもカプセル化され，一つのパケットへと成形される．今日で最も一般的な通信規格である Ethernet による TCP/IP（Transmission Conrtol Protocol / Internet Protocol）を用いた通信の場合には，図 3.2 に示すように，まずコンテンツデータにアプリケーションヘッダを付加し，更に TCP ヘッダ，IP ヘッダ，Ethernet ヘッダおよび Frame Check Sequence（FCS）を順に付加していくことで一つのパケットが完成する．各ヘッダには，当該のプロトコルにおいて必要となる情報が含まれており，これをフィールドと呼ぶ．例えば，IP ヘッダには宛先 IP アドレスや送信元 IP アドレスといったフィールドが存在する．

このような通信プロトコルの階層化を表現する手段としては，OSI 参照モデルが広く用いられてきた [53]．OSI 参照モデルは，通信プロトコルをその役割毎に 1 から 7 の層（レイヤ）に分類する．上述した TCP パケットの Ethernet 転送においては，Ethernet がレイヤ 2，IP がレイヤ 3，TCP がレイヤ 4 に分類される．OSI 参照モデルにおいて，コンテンツ情報を含む通信内容本体はレイヤ 7 に分類される．ルータにおいては，レイヤ 4 程度まで扱うことで，IP アドレスや MAC アドレスを用いたルーティング，TCP ポート番号を用いたフィルタリング，QoS 制御などが実現可能となる．

構文解析機構では，受信したパケットデータから，レイヤ 2 およびレイヤ 3，レイヤ 4 のヘッダとそれ以降のパケットデータを分離し，更に各ヘッダのフィールド毎に値を抽出する．以下では，各ヘッダ毎にそれぞれが持つフィールドと，その用途に関して説明する．

Ethernet ヘッダ:

レイヤ 2 に位置する Ethernet ヘッダ [54] の各フィールドとデータサイズ，その役割，ルータ内においてそのフィールドを使用する処理機構を表 3.1 にまとめた．なお，表 3.1 に示したフィー

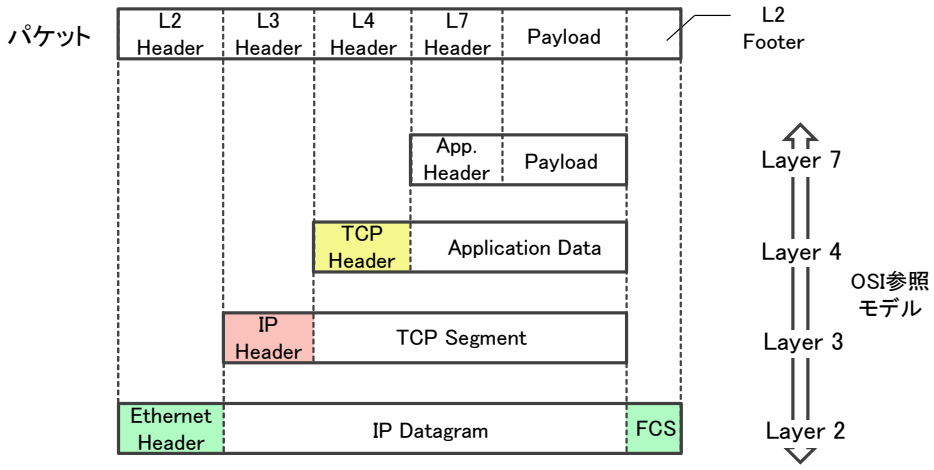


図 3.2: 一般的なパケットの構造

ルドは，上から順番に，ヘッダ内において当該フィールドが現れる順番を示している．以下では，各フィールドに関して説明する．

宛先 MAC アドレスおよび送信元 MAC アドレスは，それぞれパケットが次に到達すべき機器と一つ前の機器のアドレスを示す．これらのフィールドは，アドレス解決機構において IP アドレスをもとに得られ，ヘッダ修正機構においてヘッダへと反映される．また，QoS において優先度決定に使用される他，Modification において更新される．タイプフィールドは上層のレイヤ 3 プロトコルを示す．構文解析機構に上層ヘッダのフォーマット特定に用いられる．タイプ値が 0x0800 ならばレイヤ 3 プロトコルは IP である．

IP ヘッダ:

レイヤ 3 に位置する IP ヘッダ [55] の各フィールドとデータサイズ，その役割，ルータ内においてそのフィールドを使用する処理機構を表 3.2 にまとめた．表 3.2 に示したフィールドは，Ethernet ヘッダと同様に，上から順にヘッダ内において当該フィールドが現れる順番を示している．以下では，ルータのパケット処理において用いられるフィールドに関して説明する．

バージョンフィールドは IP のバージョンを示す．IPv4 なら 4，IPv6 なら 6 となる．この値は構文解析機構で IP ヘッダ長を決定するために用いられる．ヘッダ長フィールドは，IP ヘッダ長を 32bit ブロック単位で表す．この値もバージョンフィールドと同様に，構文解析機構において IP ヘッダ長を決定するために用いられる．Type of Service（ToS）フィールドはパケットの優先度を表す．QoS 機構において，パケットの優先度を決定するために用いられる．Time to Live（TTL）はパケットの寿命を表す．ネットワークでは，IP パケットがネットワーク機器を通過する度に TTL の値が 1 減算される．TTL が 0 になったパケットは破棄することで，ネットワークループに陥ったパケットを取り除く．TTL はルータのヘッダ修正機構において読み出され，減算される．プロトコル番号は上層のレイヤ 4 プロトコルを示す．TCP なら 6，UDP なら 17 となる．この値は，構文解析機構において，上層ヘッダのフォーマット特定に用いられる．チェックサムは IP ヘッダの整合性を示す．この値は，IP ヘッダ全体を 1 の補数演算というアルゴリズムにより計算すること

で得られる [56]．チェックサムは，ルータのヘッダ修正機構において計算がなされ，ヘッダへと書き込まれる．送信元 IP アドレスおよび宛先 IP アドレスは，通信の端点となる送信元ホストと宛先ホストのアドレスを示す．これらの値は．ルーティングテーブルやアドレス解決機構，QoS 機構，フィルタリング機構，ヘッダ修正機構といった様々な処理機構で用いられる．

TCP ヘッダ:
レイヤ 4 に位置する TCP ヘッダ [57] の各フィールドとデータサイズ，その役割，ルータ内においてそのフィールドを使用する処理機構を表 3.3 にまとめた．表 3.3 に示したフィールドは，Ethernet ヘッダと同様に，上から順にヘッダ内において当該フィールドが現れる順番を示している．以下では，ルータのパケット処理において用いられるフィールドに関して説明する．

送信元ポート番号および宛先ポート番号は，通信における送信元と宛先のプログラムを示す．Hypertext Transfer Protocol（HTTP）なら 80，Domain Name System（DNS）なら 53 となる．これらのフィールドは，QoS 機構，フィルタリング機構といったプロセッシング処理において用いられる．ヘッダ長フィールドは，TCP ヘッダ長を 32bit ブロック単位で表す．構文解析機構において TCP ヘッダ長を決定するために用いられる．チェックサムフィールドは，IP ヘッダのチェックサムと同様に，TCP ヘッダの整合性を示す値であり，1 の補数演算アルゴリズムにより計算される．この値は，ヘッダ修正機構において計算がなされ，ヘッダへと書き込まれる．

3.1.2 ルーティング

ルータが提供する最も根本的な機能が，パケットの経路を決定するルーティングである．ルーティングで行われる処理は，ルーティングテーブル検索によるネクストホップのアドレス決定と，ARP テーブル検索によるネクストホップのアドレス解決である．以降では，それぞれについて述べる．

3.1.2.1 ルーティングテーブル検索

IP を用いたパケットの経路決定では，従来からルーティングテーブル検索による経路決定手法が用いられてきた．ルーティングテーブルの例を表 3.4 に示す．ルーティングテーブルのエントリは宛先 IP アドレスとそれに対応するネクストホップアドレス，ネクストホップに到達するための出力インタフェース番号からなる．ネクストホップアドレスとはパケットが次に着くべきネットワーク機器の IP アドレスである．

ルーティングテーブル検索では，パケットの宛先 IP アドレスとルーティングテーブルに表記された宛先 IP アドレスとの最長一致検索（Longest Prefix Match: LPM）が行われる [58]．ルーティングテーブルにおける宛先 IP アドレスは，CIDR（Classless Inter-Domain Routing[59]）のプレフィックス長を伴って表記されるため，実際には表記されている宛先 IP アドレスの先頭からプレフィックス長分のデータとマッチングが行われる．そこで，LPM では各エントリの中で一致する宛先 IP アドレスの bit 数が最も長いエントリが採用される．

ルーティングテーブル検索における最も単純な LPM の決定方法は，プレフィックス長の長い順にエントリをソートし，最初にヒットしたエントリを採用する線形探索法である．しかしながら，

表 3.1: Ethernet ヘッダのフィールドと用途

フィールド	サイズ	役割	使用する処理機構
宛先 MAC アドレス	48 bit	次の機器のアドレス	ARP, QoS, Modification
送信元 MAC アドレス	48 bit	前の機器のアドレス	ARP, QoS, Modification
タイプ	16 bit	レイヤ 3 プロトコルの識別	Parsing

表 3.2: IP ヘッダのフィールドと用途

フィールド	サイズ	役割	使用する処理機構
バージョン	4 bit	IP のバージョン	Parsing
ヘッダ長	4 bit	IP ヘッダの長さ	Parsing
Type of Service(ToS)	8 bit	パケットの優先度	QoS. Scheduling
データグラム長	16 bit	IP ヘッダ + データ部の長さ	-
ID	16 bit	一連の分割パケットの識別子	-
フラグ	3 bit	パケット分割の有無	-
オフセット	13 bit	パケットの分割位置	-
Time to Live(TTL)	8 bit	パケットの寿命	Modification
プロトコル番号	8 bit	レイヤ 4 プロトコルの識別	Parsing
チェックサム	16 bit	IP ヘッダの整合性検査	Modification
送信元 IP アドレス	32 bit	送信元ホストのアドレス	Routing Table, ARP QoS, Filtering
宛先 IP アドレス	32 bit	宛先ホストのアドレス	Routing Table, ARP QoS, Filtering
オプション	可変長	様々な付加機能の実現	-

表 3.3: TCP ヘッダのフィールドと用途

フィールド	サイズ	役割	使用する処理機構
送信元ポート番号	16 bit	送信元ポートの識別	QoS, Filtering
宛先ポート番号	16 bit	宛先ポートの識別	QoS, Filtering
シーケンス番号	32 bit	一連のパケットの順序	-
確認応答番号	32 bit	受信を完了したデータサイズ	-
ヘッダ長	4 bit	TCP ヘッダの長さ	Parsing
予約	6 bit	予約済みビット	-
フラグビット	6 bit	URG/ACK/PSH/RST/SYN/FIN を表す	-
ウィンドウサイズ	16 bit	受信側のウィンドウサイズ	-
チェックサム	16 bit	TCP パケットの整合性検査	Modification
緊急ポインタ	16 bit	URG が 1 の場合のみ使用	-
オプション	可変長	様々な付加機能の実現	-

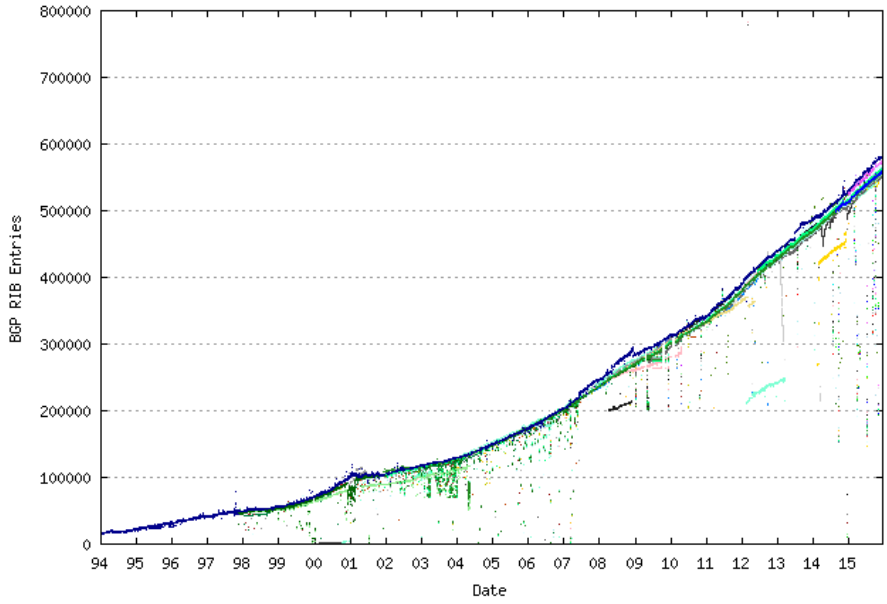


図 3.3: AS における BGP テーブルエントリ数の年間推移

表 3.4: ルーティングテーブルの例

Destination IP Address / Prefix	Next Hop Address	Interface
192.168.12/20	10.24.1.1	1
192.168.104/24	10.124.1.4	4
192.168.0/17	10.23.1.6	3
0.0.0.0/0	10.1.1.0	7

線形探索法によるエントリ探索速度はルーティングテーブルのエントリ数に反比例して遅くなる．図 3.3 はある AS（Autonomous System）における BGP（Border Gateway Protocol[60]）テーブルエントリ数の年間推移を示したグラフである．BGP テーブルとは，異なる AS 間のゲートウェイにおいてルータ同士が経路情報を交換し，その結果をアップデートすることで構築されるネットワークの全経路情報を示したテーブルである．ルーティングテーブルは BGP テーブルをもとに一つの宛先へ辿り着くための最短経路を集積することで構築される．図 3.3 に示されるように，近年の大規模ネットワークにおける全ネットワーク経路数は 60 万に届く．線形探索によりこのオーダの膨大なエントリを探索することは実用上不可能である．

そこで，実際には Trie[61] や，それを改良した Patricia Tree[62] などのツリー木を用いた手法，キャッシュや Ternary Content Addressable Memory（TCAM）[63] といった特殊なメモリ機構を用いた手法によってテーブル検索の高速化がなされる．テーブル検索の高速化手法については 3.1.5 項で詳述する．

表 3.5: ARP テーブルの例

Address	HWtype	HWaddress	Flags Mask	Iface
192.168.1.1	ether	00:20:88:D5:B4:2F	CM	eth0
192.168.1.13	ether	00:D0:18:AB:25:E1	C	eth1
192.168.23.2	ether	00:EF:A6:01:0C:63	C	eth1
192.168.106.12	ether	00:80:98:01:A7:C0	C	eth1

3.1.2.2 アドレス解決

パケットでは，ネクストホップ情報はレイヤ 2 で管理される．レイヤ 3 の IP アドレスは通信の始まりとなった送信元ホストと最終的な宛先ホストの管理を行うためである．よって，ルーティングテーブルからネクストホップの宛先 IP アドレスを得た後，得られた IP アドレスからネクストホップ機器の MAC アドレスを索引する必要がある．この処理はアドレス解決プロトコル ARP（Address Resolution Protocol）をもとに行われる．ARP では，宛先 IP アドレスの問い合わせパケットがネットワークにブロードキャストされる．該当した IP アドレスを持つ機器が自身の MAC アドレスを載せて応答することで，所望の MAC アドレスが得られる．

また，ルータは ARP の結果を格納した ARP テーブルと呼ばれるキャッシュを持ち，ARP テーブルを検索することによってこの処理を高速化している．ARP テーブルの例を表 3.5 に示す．表 3.5 において，Address は問い合わせ IP アドレス，HWtype は物理層プロトコル，HWaddress は Address を持つ機器の MAC アドレス，Iface は接続されているインタフェース情報を示す．また，Flags Mask の C は通常エントリであることを，更に M が加えられたエントリは永続エントリであることを示す．

3.1.3 プロセッシング

2.1 節で述べたように，初期のルータはルーティングおよびフォワーディングの性能を向上させることが至上命題であったが，ハードウェア技術の向上に伴い，パケット転送に要求される性能を満たすことは困難ではなくなった．これは即ち，ルータのハードウェアリソースをパケット転送処理以外に割くことが可能になったことを意味する．このような背景から，近年のルータには，パケットを指定された宛先へと転送するルーティングやフォワーディングの機能以外に，ネットワークの利便性を向上させるプロセッシングの機能を持つことが期待されるようになった．プロセッシングにおいて，パケットのフィルタリングや QoS 制御，暗号化，カプセリングといった様々な機能が実装されてきた．本節では，近年のルータが一般的に有するプロセッシング機能である，フィルタリング機構と QoS 制御機構について説明する．

3.1.3.1 フィルタリング機構

ルータにおいては，ACL（Access Control List）と呼ばれるテーブルリストによるパケットのフィルタリング機能が実装されている．ACL には標準 ACL と拡張 ACL が存在する．

表 3.6: 標準 ACL の例

action	Destination IP
permit	192.168.1.0.0.0.0.255
permit	192.168.10.1.0.0.0.0
permit	202.1.0.0.0.0.255.255
deny	any

表 3.7: 拡張 ACL の例

action	rule
permit	tcp host 192.168.1.1 host 10.0.1.1 ep 23
deny	tcp host 192.168.1.1 host 10.0.1.1 ep 80
permit	tcp host 192.168.1.0.0.0.0.255 host 10.0.1.1 ep 80
permit	ip any any

標準 ACL では，パケットの送信元 IP アドレスのみをチェックすることで，該当する送信元 IP アドレスを持つパケットに対しアクションが施される．表 3.6 に標準 ACL の例を示す．ACL では，ルーティングテーブルと同様に IP アドレスとプレフィックス長を合わせて記載することで，適用する IP アドレスの値の範囲を示している．そして，該当する IP アドレスのパケットに対し permit（許可）や deny（拒否）といったアクションを適用する．ACL は IP アドレスと併せてプレフィックス長を指定し，適用範囲を設定する都合上，ルーティングテーブル検索と同じく LPM によってエントリ検索を行う．

次に，拡張 ACL について説明する．拡張 ACL では，主にレイヤ 3 ヘッダの送信元 IP アドレス，宛先 IP アドレス，プロトコル番号，そしてレイヤ 4 ヘッダの送信元ポート番号，宛先ポート番号の 5 タプルを用いてパケットのマッチングが行われる．5 タプルを用いることで，通信の方向や使用されるアプリケーションを考慮したフィルタリングが実現される．拡張 ACL の基本的な構成は標準 ACL と同じである．表 3.7 に拡張 ACL のルール例を示す．例えば，表 3.7 の第一行目は，送信元ホスト 192.168.1.1 から宛先ホスト 10.0.1.1 への TCP ポート 23 番（SSH）の通信の許可を表す．

3.1.3.2 QoS 制御機構

近年のルータにおいて Quality of Service（QoS）は重要な機能の一つとなっている．QoS の実装において，ルータでは Diffserv モデルが一般的に用いられる．

Diffserv の処理はマーキングとスケジューリングに分割される．まず，ネットワークを流れるパケットはフローに分類され，フロー毎の優先度がパケットヘッダにマーキングされる．ここで，フローとはヘッダ情報のいくつかの値をもとにパケットをグループ化したものであり，設定者によりフローの定義は異なる．そして，Diffserv に対応したネットワーク機器は，パケットを転送する際，マーキングされた優先度をもとにキューイングし，スケジューラによって転送順序を制

御した上で転送することによりフロー単位の QoS を実現する．QoS のためのキューイング，スケジューリングに関しては，従来のフォワーディング機能の改善にあたるため 3.1.4 項で後述する．本項では，Diffserv におけるマーキング処理に関して説明する．

ルータにおけるマーキング方法は，レイヤ 2 ヘッダの IEEE802.1q Virtual LAN（VLAN）タグの中にある Class of Service（CoS）値を用いる方法，Multi-Protocol Label Switching（MPLS）ラベルの Exp ビットを用いる方法などがあるが，本論文では現在主流となっているレイヤ 3 ヘッダを用いた方法を説明する．この方法では，レイヤ 3 ヘッダにある ToS（Type of Service）フィールドに IP Precedence 値または DS Code Point（DSCP）値を格納することでマーキングがなされる．IP Precedence は ToS の先頭 3 ビットを用いて，0 から 7 の 8 段階の優先度を与える．一般的に，データトラフィックには 0，音声トラフィックには 5 が与えられる．DSCP は ToS の先頭 6 ビットを用いることで，IP Precedence との互換性を保ちながら 8 段階の優先レベルに加え 8 段階の破棄レベルを与える．一般的に，データトラフィックには 0，音声トラフィックには 46 が与えられる．

これらのマーキングはフロー単位で与えられる．ルータは到着したパケットをフロー単位で分類し，各フローにおける優先度を決定した上で ToS フィールドを更新し転送する．フロー分類には，送信元 IP アドレスや宛先 IP アドレス，送信元ポート番号，宛先ポート番号，プロトコル番号，入力インタフェース，パケットサイズ，CoS，ToS など様々な値が用いられてきた．現在最も主流となっているフロー分類方法は，今述べた値のうち最初の 5 項目を用いた方法である．

ルータでは，ワイルドカードを用いたフロー情報と，そこから決定される優先度を QoS テーブルに格納し，到着パケットに対して QoS テーブルの探索を行うことで所望の IP Precedence 値または DSCP 値を得る．ここで，QoS テーブルエントリのキーとなるフロー情報にはワイルドカードを指定できることから，エントリ検索は LPM によって行われる．

3.1.4 フォワーディング

ルーティングにおいて転送経路の選択がなされたパケットは，宛先に転送するためにいくつかの処理を施される．このフォワーディング処理を大別すると，ヘッダの内容を修正するモディフィケーション機構と，パケットをキューイングし転送制御を行うためのスケジューリング機構に分けられる．

3.1.4.1 モディフィケーション機構

ルーティングにおいて，パケットは宛先 IP アドレスを持つホストへ到達するためのネクストホップ MAC アドレスおよび出力インタフェース番号を得た．そこで，得られたネクストホップ情報をパケットに格納するために，レイヤ 2 ヘッダを修正する．具体的には，レイヤ 2 ヘッダの宛先 MAC アドレスフィールドをルーティングで得られた宛先 MAC アドレスに書き換え，送信元 MAC アドレスフィールドを現在処理中のルータの MAC アドレスに書き換える．

モディフィケーション機構では，その他にもレイヤ 3 ヘッダの TTL（Time to Live）と CRC チェックサム情報，場合によっては ToS（Type of Service）を更新する．TTL は，ネットワークにおいて宛先へ到達できずループしているパケットがネットワーク中に永久的に残る状態を防ぐため，通

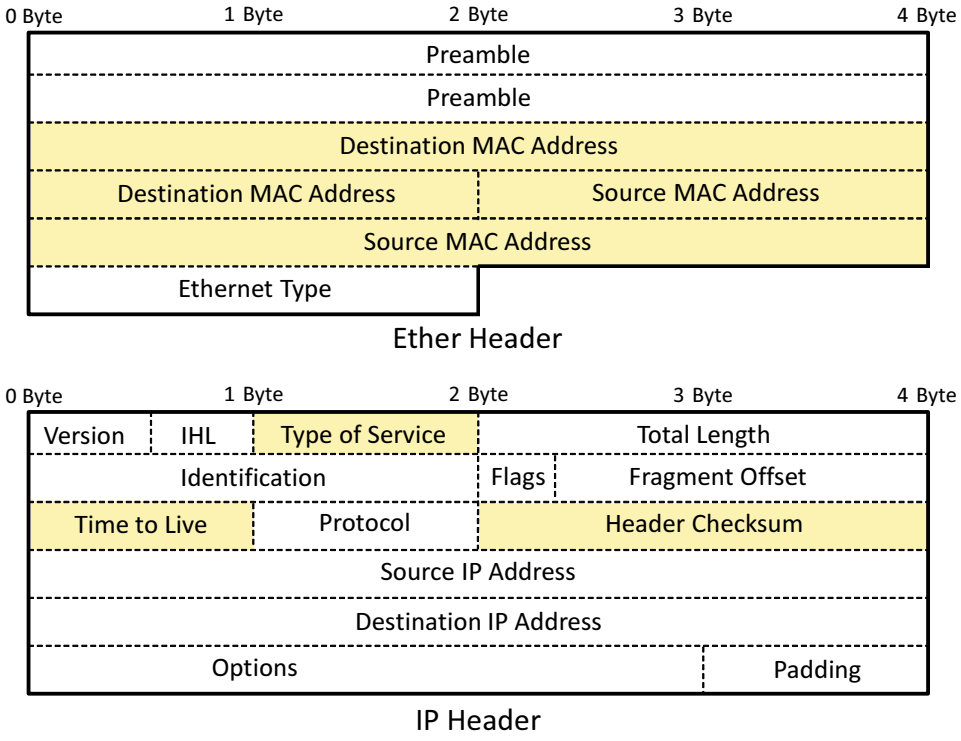


図 3.4: モディフィケーションにおけるヘッダ修正箇所

過するホップ数を制限するための値である．従って，ルータで処理されたパケットは TTL 値を 1 減算する必要がある．CRC チェックサムは，レイヤ 3 ヘッダの値が不本意に変更されてしまった場合にそれを検知するため，レイヤ 3 ヘッダ全体を CRC ハッシュ化した値である．ルータでは TTL の減算によりレイヤ 3 ヘッダの値が変化する．そのため，CRC チェックサムを再度計算しなおす必要がある．また，ルータは，3.1.3.2 項で述べたように，QoS 制御に用いる ToS 値を更新することがある．この場合も，モディフィケーションにおいて ToS を更新する．モディフィケーションにおける各ヘッダの更新箇所を図 3.4 にまとめた．

3.1.4.2 スケジューリング機構

モディフィケーションによって転送準備の整ったパケットは，ルーティングにおいて決定されたインタフェースから適切なネクストホップへと転送される．この時，帯域幅の差や Ethernet 回線の CSMA/CD 方式の転送制御，全二重通信の PAUSE フレームなどから，即座にパケットを回線に送り出すことができるとは限らない．このような場合に輻輳制御が行われる．輻輳制御にはポリシングとシェーピングの 2 種類がある．それぞれの概要を図 3.5 に示す．

ポリシングでは，データ転送レートに閾値を設け，閾値を超える場合にはパケットを破棄する．一方で，シェーピングでは閾値を超えるパケットを一旦キューに蓄え，後に転送する．ルータにおいては，制御の柔軟性からシェーピングが用いられることが一般的である．

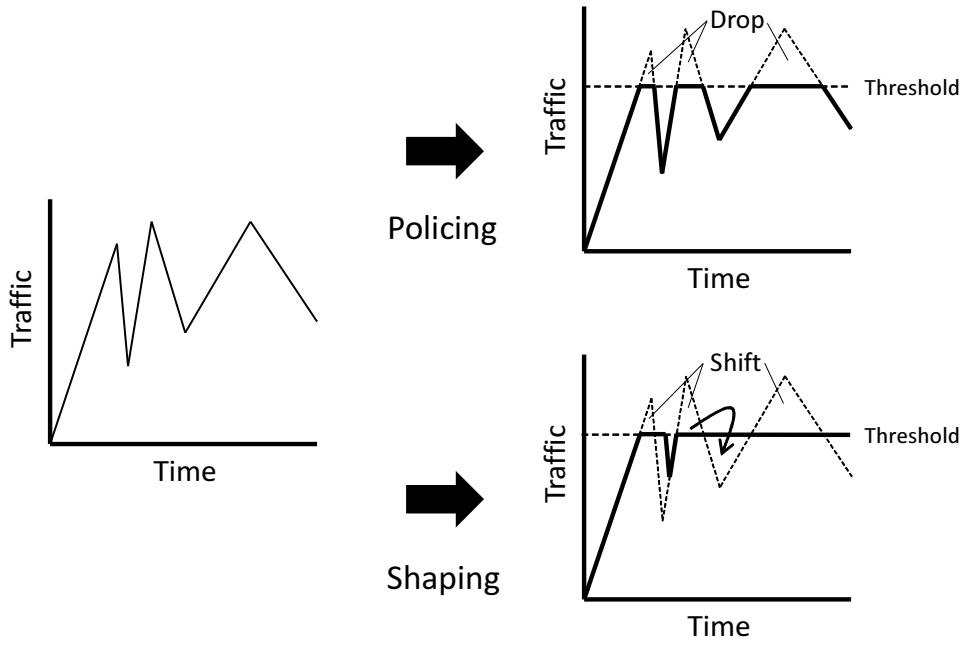


図 3.5: ポリシングおよびシェーピングの概要

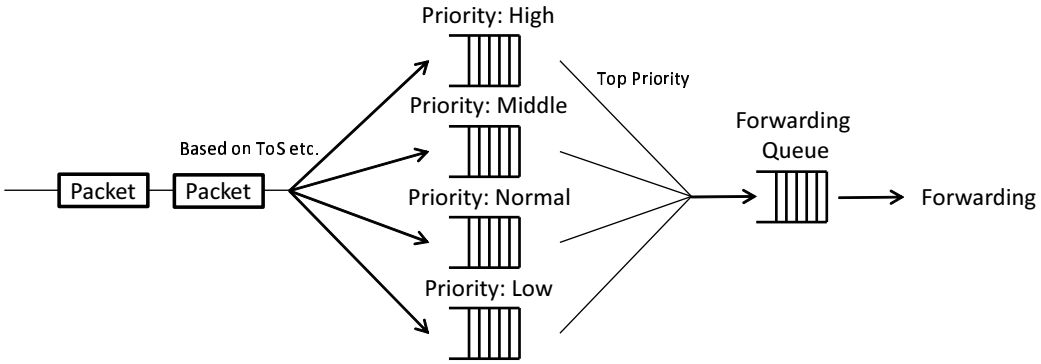


図 3.6: プライオリティキューイングの例

シェーピングでは，キューへと蓄えたパケットをスケジューリングのもとに転送する．この際，QoS テーブルで得られた優先度をもとにキューイング方法を工夫することで，QoS を実現する．このようなスケジューリング機構として最も基本的なプライオリティキューイングがある．プライオリティキューイングでは，優先度の異なるクラス単位でキューを用意する．プライオリティキューイングの例として，図 3.6 には四つの異なる優先度（high，middle，normal，low）を持ったキューによるスケジューリングの概要を示す．まず，各キューへのパケットの割り当ては 3.1.3.2 項で説明したレイヤ 3 ヘッダの ToS フィールドやレイヤ 2 ヘッダの CoS フィールド，MPLS ラベル中の Exp ビットなどをもとに決定される．そして，プライオリティキューイングでは，各キューへ蓄えられたパケットはより上位の優先度を持つキューが空になった場合にのみ，転送キューへ

と送られる．

このような手法では，上位のキューに蓄えられたパケットが消化されない場合，下位優先度キューにあるパケットが稀にしか転送されない事態が起こりうる．そこで，実際には重み付けキュー（Custom Queue）という手法が用いられている．重み付けキューでは優先度の低いキューであっても，一定の割合で転送キューへとパケットを送ることで上記の問題を解決する．

3.1.5 パケット処理高速化に関する研究

ルータのパケット転送処理において性能ボトルネックとなる部分は，主にテーブル検索であることが多くの研究者に指摘されている [64, 65]．これは，ルータにおける大部分のテーブル検索が LPM により行われるため，1 エントリの探索に多数のメモリアクセスを要することが原因である．各種テーブルを格納した主メモリである Dynamic Random Access Memory（DRAM）は，大容量だがアクセス遅延が大きく，1 アクセスあたりに数十 ns を要する．従って，LPM を行うためには 100ns 以上の遅延を要した．これに対し，100Gbps のネットワーク環境においてワイヤレートにパケットを処理するためには，最短パケット長のパケットを想定した場合，1 パケットあたり 5ns 程度でテーブル検索を完了しなければならない．

そこで，ルータにおけるテーブル検索を高速化する手法が数多く研究されてきた．本項では，このようなテーブル検索高速化に関する研究として，従来のルータで用いられており，なおかつ本論文と関わりのあるキャッシュ機構を用いた手法と，現在において一般的に採用されている Ternary Content Addressable Memory（TCAM）を用いた手法を紹介する．

テーブル検索の高速化を目的としたキャッシュは，対象とする処理によって幾つかのグループに分類できる．一つは，パケットのルーティングテーブル検索にのみ焦点を当てた，IP キャッシュである．もう一つは，パケットを複数のヘッダ情報からフローに分類し，フロー単位で等しくなる全テーブル検索結果をキャッシュするフローキャッシュである．そして最後に，テーブル検索に限らず，パケット処理全体の結果をキャッシュするパケット処理キャッシュがある．以降では，それぞれにわけて関連する研究を紹介しながら説明する．

3.1.5.1 IP キャッシュ

2.1 節で述べたように，初期のルータは全てのパケット転送処理をソフトウェアにより行っていた．1990 年代に入り，パケット転送機構のハードウェア化がなされ始めたが，まだハードウェア技術が未熟であり，テーブル全体をハードウェア化することは費用や回路規模の点から困難であった．そこで，ルーティングテーブルエントリの中でも再度参照される可能性の高いエントリを，小容量だがアクセスの高速なキャッシュメモリにコピーすることでテーブル検索を高速化する IP キャッシュが提案された．

IP キャッシュでは，同一の宛先 IP アドレスを持つパケット群の 1 パケット目を従来通りソフトウェアルーティングテーブルによって処理し，その結果をキャッシュへと保存することで，このようなパケット群の 2 パケット目以降のパケットに対してはキャッシュを参照することで高速にルーティングテーブル検索を完了する．IP キャッシュのエントリ検索は，IP アドレスとプレフィッ

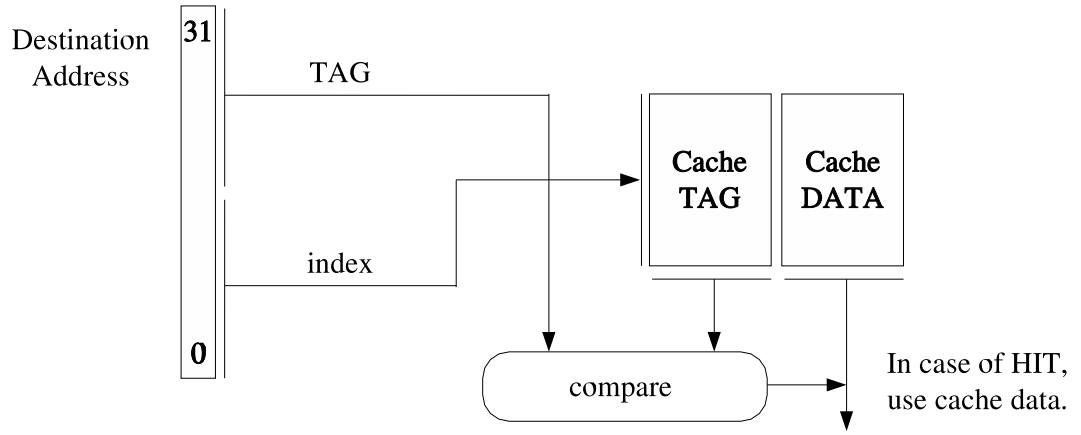


図 3.7: IP キャッシュのイメージ [3]

クス長を用いた LPM ではなく，IP アドレスの完全一致検索により行われる．完全一致検索では，後述するハッシュを用いることで 1 度のメモリアクセスによりエントリ決定が可能となる．また，キャッシュに用いられる Static Random Access Memory（SRAM）は 1ns 程度で高速にメモリアクセスが可能である．従って，キャッシュ内に該当する IP アドレスのエントリが存在する場合には，高速にルーティングテーブル検索を完了できる．

キャッシュでは，データ参照時に所望のエントリが存在した場合をキャッシュヒットと呼び，エントリが存在しなかった場合をキャッシュミスと呼ぶ．IP キャッシュでは，キャッシュヒットした場合にはキャッシュによって 1ns 程度で処理を完了できるが，キャッシュミスした場合には従来通り 100ns 程度の遅延が生じる．そこで，従来より，キャッシュミス率の改善を目的として，キャッシュインデックスの生成方法やキャッシュ構成，エントリ追い出しアルゴリズムなどが検討されてきた．

ミス率の改善 1: キャッシュインデックスの生成方法

IP キャッシュの多くでは，キャッシュのエントリ探索に用いるインデックスとして IP アドレスの一部を使用する．インデックスとは，キャッシュエントリを探索するための仮想アドレスのことである．カリフォルニア大学サンディエゴ校の Talbot らは，経路探索において IP アドレスをインデックスとキャッシュのタグに用いる IP キャッシュを提案した [3]．IP キャッシュの特徴は，キャッシュライン数が 1 エントリである点，IPv4 アドレスを二分割しインデックスとタグとして利用する点である．その理由として，キャッシュラインに複数のネクストホップ情報を格納してもアクセス時間を削減できないため，キャッシュラインには 1 個の結果だけしか入れる必要がないとしている．IP キャッシュは通常のプロセッサのキャッシュ同様，インデックスでキャッシュメモリにアクセスし，該当エントリのタグと IPv4 アドレス内のタグを比較して一致すればキャッシュヒットとなる．9 個のサイトから採取した実ネットワークトレースを利用した調査の結果，図 3.7 に示すように，IPv4 アドレスの上位をタグ，下位をインデックスとすることで良い性能が得られることを示した．

ニューヨーク州立大学の Chiueh らは Talbot らと同様にネットワークの時間的局所性に注目し，

IP アドレスをプロセッサのインデックスとして扱うことで，通常のプロセッサのキャッシュを利用しソフトウェアベースで経路検索を高速化する手法を提案している [66]．Chiueh らは入力パケットの宛先アドレスの 11bit 目から 5bit 目までをインデックスとして扱うことが最適であるとしている．また，宛先アドレスの 31bit 目から 12bit 目及び 4bit 目から 2bit 目をキャッシュタグとして用いている．

ミス率の改善 2: 最適なキャッシュ構成の検討

従来，IP キャッシュの最適なキャッシュ構成について多く検討がなされてきた．一般的にキャッシュのエントリ管理方法には以下の三つの方式がある．

1. **ダイレクトマップ方式**
   ダイレクトマップ方式では，図 3.8(a) に示すように一つのエントリに一つのインデックスが割り当てられる．従って，エントリ探索はインデックスにより一意に決定され，高速に行われる．しかしながら，エントリのインデックスが重複してしまった場合には既存エントリを追い出さなければならず，場合によっては有用なエントリを追い出すことでミスが増加してしまう．

2. **セットアソシアティブ方式**
   ダイレクトマップ方式の欠点を補うため，図 3.8(b) に示すように複数のエントリに一つのインデックスを割り当てたのがセットアソシアティブ方式である．ここで，割り当てられるエントリ数は連想度（way 数）と呼ばれる．セットアソシアティブ方式では，同一インデックスの各エントリが既に埋まっている状態で新規エントリを登録する場合，追い出すエントリを決定しなければならない．このような追い出しエントリの決定方法は追い出しアルゴリズムと呼ばれ，多くのキャッシュシステムで効果的な追い出しエントリの決定方法が研究されてきた．追い出しアルゴリズムについては後述する．また，エントリの探索はコンパレータを way 数分並列に配置しキャッシュタグ同時に比較することで，ダイレクトマップ方式同様，1cycle で探索可能である．そのため，連想度を大きくした場合にはコンパレータ数，追い出しアルゴリズムの複雑さが増し，ハードウェアコストが増大する．近年のプロセッサの L1 キャッシュ，L2 キャッシュを見ても，連想度は 4 か 8 であることが一般的である．

3. **フルアソシアティブ方式**
   フルアソシアティブ方式では，インデックスを用いず，図 3.8(c) に示すように全てのエントリを同時に探索することで最も効果的なエントリ管理を実現する．しかしながら，セットアソシアティブ方式でも述べたように，エントリの同時探索はハードウェアコストの増大を招くため，フルアソシアティブ方式が実際のキャッシュ機構に採用されることは少ない．エントリ数が少なくハードウェア資源を潤沢に用いることのできるキャッシュにおいては本方式が最も有効なエントリ管理手法となる．

IP キャッシュにおいては，多くの研究でセットアソシアティブ方式を採用することが効果的であると述べられている．Talbot らはキャッシュエントリ数を 4K〜256K に変化させ，ダイレクトマップ方式，2way と 4way のセットアソシアティブ方式のキャッシュ構成により，9 つのネット

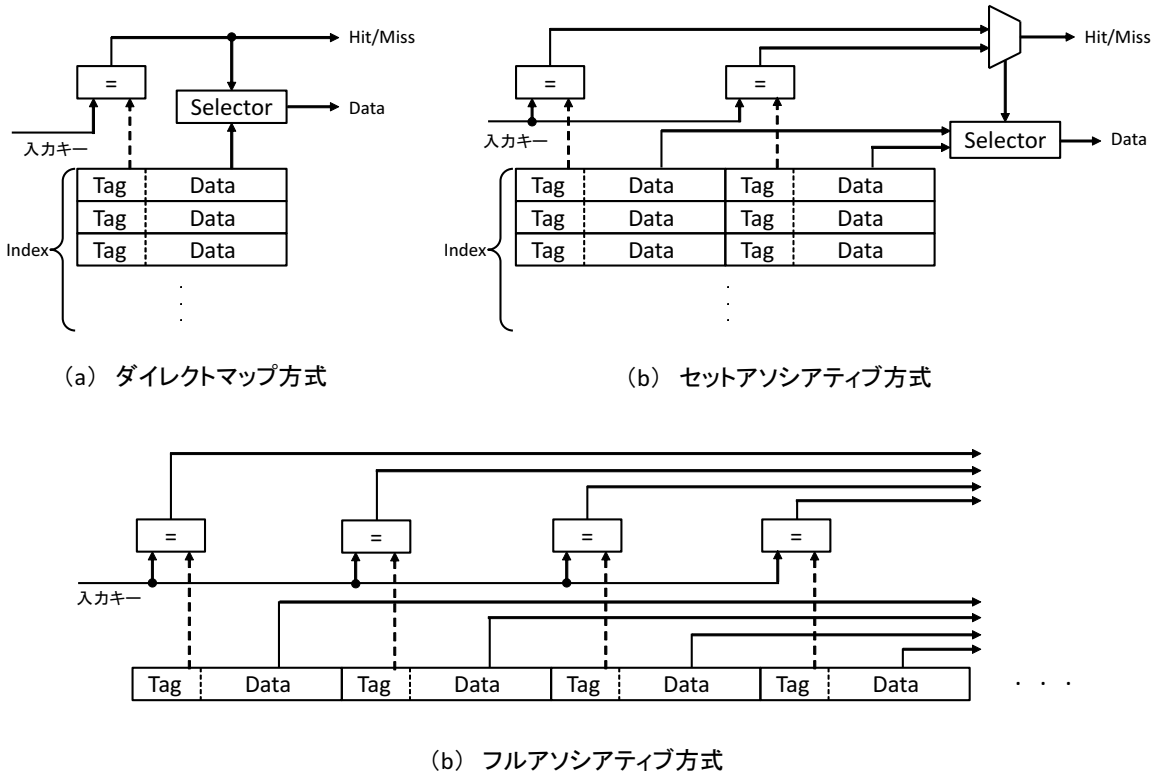


図 3.8: 各エントリマップ方式のアーキテクチャ

ワークトレースを用いた IP キャッシュのシミュレーションを行い，ミス率を比較した [3]．シミュレーションによれば，キャッシュを 4way セットアソシアティブにすることで，ダイレクトマップ方式と比べ 30%から 80%程度，2way アソシアティブ方式に比べ 10%から 50%程度ミスを改善できると述べている．

また，ニューヨーク州立大学の Chiueh らは，同様にルーティングテーブル検索におけるキャッシュ性能を，4K と 8K エントリのキャッシュにおいて連想度を 1，2，4 で変化させた場合のキャッシュミス率を比較することで検討している [67]．Chiueh らによると，4K エントリのキャッシュにおけるミス率は，ダイレクトマップ方式において 12.71%，2way セットアソシアティブ方式において 8.42%，4way セットアソシアティブ方式において 6.86%であり，4way セットアソシアティブ方式を用いることでダイレクトマップ方式に比べ 46%程度のミスを削減できている．また，2way セットアソシアティブ方式に比べると，4way セットアソシアティブ方式では 19%程度のミスを削減できている．8K エントリのキャッシュの場合，ミス率はダイレクトマップ方式において 7.57%，2way セットアソシアティブ方式において 4.59%，4way セットアソシアティブ方式において 3.29%であり，4way セットアソシアティブ方式を用いることでダイレクトマップ方式に比べ 57%，2way セットアソシアティブ方式に比べ 28%程度のミスを削減できている．これらの改善度合いは前述した Talbot らの研究結果とも一致しており，IP キャッシュにおける 4way セットアソシアティブ方式の有用性が示されている．

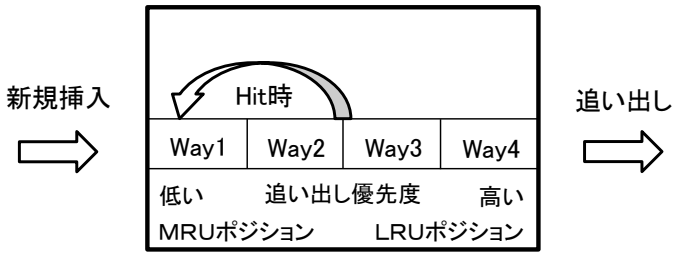


図 3.9: LRU におけるエントリ制御の概要

ミス率の改善 3: エントリ追い出しアルゴリズムの検討

前述したように，セットアソシアティブ方式のキャッシュでは追い出しエントリの決定がキャッシュミス率に大きく影響する．追い出しエントリをランダムに決定するランダム手法やエントリ登録した順に追い出す FIFO（First In First Out）手法など，キャッシュシステムにより異なるアルゴリズムが採用されてきた．IP キャッシュでは，追い出しアルゴリズムとして LRU（Least Recently Used）が有効であることが多くの研究により述べられている．LRU の概要を図 3.9 に示す．LRU ではエントリ衝突が起こった場合，参照された時間が最も古いエントリから順に追い出すことで，エントリ参照の局所性を考慮した追い出しエントリの決定がなされる．従って，ネットワークトラフィックの時間的局所性を利用した IP キャッシュにおいても高い効果が期待できる．

ニューヨーク州立大学の Rooney らは IP キャッシュにおける追い出しアルゴリズムとして，様々な用途のキャッシュにおいて一般的に用いられているランダム方式，FIFO 方式，LRU 方式をそれぞれ適用した場合のキャッシュミス率を比較している [68]．Rooney らによると，それぞれの追い出しアルゴリズムを適用した場合のキャッシュミス率は，ランダム方式を用いた場合が 7.55%，FIFO 方式を用いた場合が 7.53%，LRU 方式を用いた場合が 4.77%である．これより，ランダム方式と FIFO 方式ではキャッシュミス率が変わらないが，LRU を適用することで 37%程度のミスが削減でき，IP キャッシュにおいて LRU が効果的であることが示されている．

また，米ディジタル・イクイップメント・コーポレーション社の Jain らも同様に，宛先アドレス探索処理における IP キャッシュにおいて，同一フローに属するパケットの到着が時間的局所性を持つことから，適切な追い出しアルゴリズムを検討している [69]．論文 [69] では，IP キャッシュにおける追い出しアルゴリズムとしてランダム方式，FIFO 方式，LRU 方式に加え，千里眼アルゴリズムと呼ばれる最適な追い出しアルゴリズムを適用した際のキャッシュミス率を比較し，各追い出しアルゴリズムのミス削減性能を評価している．千里眼アルゴリズムとは，キャッシュ追い出しアルゴリズムにおいて理論的に最適な性能を有するアルゴリズムである [70]．千里眼アルゴリズムでは，将来に到着するデータ情報を用い，今後参照されないデータまたは再参照が最も遅いデータから順に追い出すことで，最適なエントリ追い出しを可能とする．しかしながら，実用において，将来到着するデータが分かっていることは稀であり，本手法を実用することは困難である．Jain らは，千里眼アルゴリズムを適用した際のミス率と，他のアルゴリズムによるミス率を比較することで，アルゴリズムがどの程度最適な性能に近いかを評価した．シミュレーショ

ンによると，容量が 10KB 以下と小さい，または 300KB 以上と大きいキャッシュでは，ランダム，FIFO，LRU に性能の差はなく，これらを適用した際には千里眼アルゴリズムを適用した場合の 3 倍程度のミスが生じる．一方で，容量が 50KB から 200KB 程度のキャッシュでは，LRU を適用した場合にミスが大きく削減されており，64KB キャッシュの場合のミス率は，千里眼アルゴリズムを適用することで 3%程度，LRU を適用することで 5%程度，ランダムまたは FIFO を適用することで 10%となっている．このことからも，IP キャッシュでは LRU の適用が効果的であることが示されている．

3.1.5.2 フローキャッシュ

IP キャッシュはルーティングテーブル検索に特化した高速化手法である．しかしながら，1990 年代中ごろからネットワークアプリケーションは多様化し，QoS テーブルやフィルタリングテーブルといったより複雑なキーを用いたテーブル検索が求められるようになった．ルーティングテーブル検索にのみ時間がかかっていた従来とは様相が異なる．このような背景から，ルーティングテーブルに限らず複数のテーブル検索の結果をキャッシュするフローキャッシュ機構が提案されてきた．

多くのテーブルでは，パケットヘッダにあるいくつかの値をキーとしてエントリが検索される．従って，これらの値が等しい，フローと呼ばれる一連のパケット群に対しては同じテーブル検索結果が返される．一般にフローは，パケットヘッダのフィールドから 3 タプルや 5 タプル，10 タプルなど独自の組み合わせにより定義される．例えばルーティングテーブルならば，宛先 IP アドレスの等しいパケット群をフローと呼ぶ．QoS テーブルにおいては，宛先 IP アドレス，送信元ポート番号，宛先ポート番号の 3 タプルの組み合わせをフローとすることで，ユーザおよびアプリケーションの識別を行ってきた．そこで，フローキャッシュではフローによって決定される全てのテーブル検索結果をフロー毎にキャッシュすることで，複数のテーブルを一度に検索する．一般的なフローキャッシュでは，多くのテーブル検索で用いられる 5 タプル（送信元 IP アドレス，宛先 IP アドレス，送信元ポート番号，宛先ポート番号，プロトコル番号）をフローと定義し，キャッシュタグに用いる．

ミス率の改善 1: キャッシュインデックスの検討

IP キャッシュでは宛先 IP アドレスのユニークとなる数 bit をそのままインデックスとして用いたが，フローキャッシュではそれは適さない．フロー情報は IPv4 パケットの 5 タプルでも 104bit あり，ユニークとなる数 bit を抽出するのが困難だからである．そこで，従来からフローキャッシュでは，フロー情報全体をハッシュ化し，そのハッシュ値をインデックスとして用いる方法が採用されてきた．ハッシュとは，任意長の入力に対し計算を施し，指定した bit 幅の出力を得るための方法である．

フローキャッシュでは，キャッシュエントリ数から必要な bit 幅を決定し，その bit 幅の出力を得るハッシュを実装する．例えば，エントリ数が 1,024（2^{10}）である場合には，10bit の出力を得るハッシュが必要である．ハッシュ値を得るためのハッシュ関数は様々なものが存在するが，一般的に巡回冗長検査（Cyclic Redundancy Check: CRC[71]）が用いられる．

表 3.8: CRC 生成多項式の例

エントリ数	ビット数	CRC 生成多項式
256	8	$X^8 + X^6 + X^3 + X^2 + 1$
512	9	$X^9 + X^5 + 1$
1K	10	$X^{10} + X^7 + 1$
2K	11	$X^{11} + X^2 + 1$
4K	12	$X^{12} + X^3 + 1$
8K	13	$X^{13} + X^4 + X^3 + X^1 + 1$

CRC は，データの bit 列の連続する誤りを検出するための誤り検出符号の一種であり，データ列の十分な撹拌能力を持つためデータ検索のキーとしても利用できる．出力したい bit 幅 n に合わせた n 次の生成多項式を用い，各次数の係数を並べた（n+1）bit のビット列を除数として XOR 演算を行い，最終的に得られる nbit のビット列がハッシュ値となる．従って，CRC を構成するための回路は複数段の XOR ゲートで容易に構成できるという利点も持つ．表 3.8 に各エントリサイズを達成するための CRC 生成多項式の一例を示す．

CRC によるインデックス生成においてハッシュ値の衝突は避けられない．これは，あるフローのハッシュ値と別のフローのハッシュ値が等しくなりうるということである．既存研究では，ハッシュ値衝突の可能性を削減できるとして，より最適なハッシュ関数の利用が検討された．例えば，カリフォルニア大学の Liao らは二つの異なるユニバーサルハッシュ関数をインデックス生成に用いる手法を提案している [4]．Liao らは，キャッシュバンクを二つに分割し，二つの異なるユニバーサルハッシュ関数を用いて各キャッシュバンクにアクセスすることで，ハッシュ値の衝突を改善できると述べている．図 3.10 に，提案されている二つのハッシュ関数を用いたキャッシュ機構のアーキテクチャを示す．

Liao らは，従来用いられてきた CRC と既存研究による Prime ハッシュおよび XOR ハッシュ，一つのユニバーサルハッシュと提案手法による二つのユニバーサルハッシュを用いた場合とで比較評価している．シミュレーションによると，二つのユニバーサルハッシュを併用することで CRC に比べ 5%から 9%程度ミスを改善できることが示されている．また，Prime ハッシュおよび XOR ハッシュではミスが増加してしまい，一つのユニバーサルハッシュでは大きな効果は得られていない．Liao らの研究では提案手法により数%ミスを削減できてはいるが，これはキャッシュバンクを二つにした恩恵が大きく，ハッシュのデータ攪拌性能に関しては一つのユニバーサル関数の結果が示すように CRC と同等であると考えられる．ハードウェア実装の容易さや高速さも考慮すると CRC を用いることが最適である．

ミス率の改善 2: 最適なキャッシュ構成の検討

フローキャッシュでは，IP キャッシュに比べタグとなるフロー情報サイズが大きく，1 エントリあたりのデータサイズは例えば 5 タプルを想定すると 4 倍程度となる．そのため，高いヒット率を得るために十分なエントリ数を確保することが困難であった．そこで，フローキャッシュにおいてもキャッシュの連想度や総エントリ数といったキャッシュの最適な構成が多くの研究により検

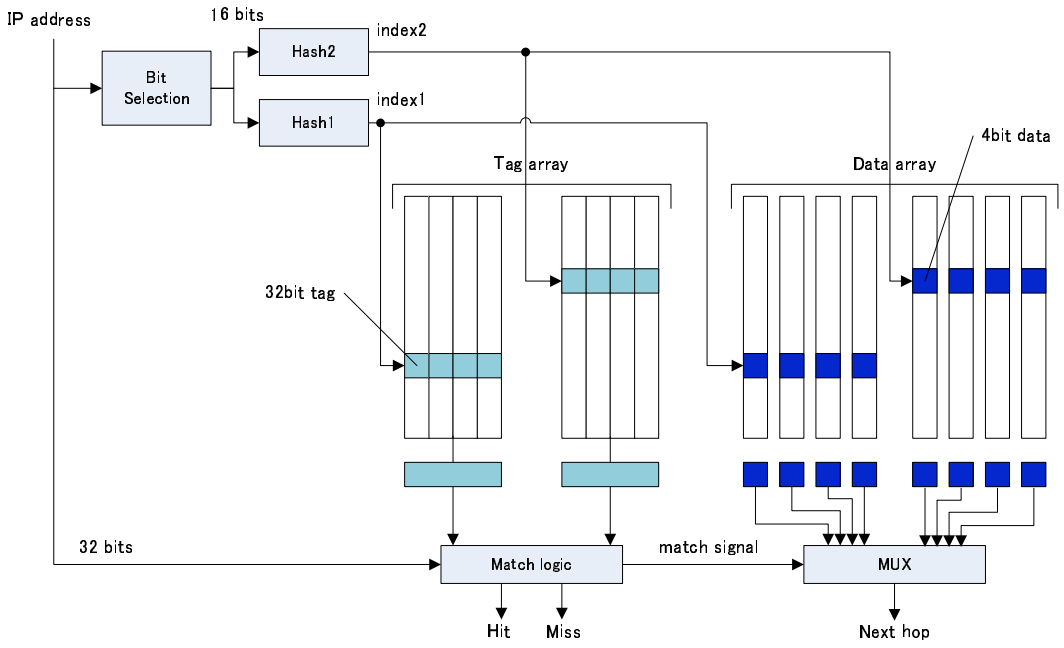


図 3.10: 二つのハッシュ関数を用いたキャッシュアーキテクチャ[4]

討されてきた．

ジョージア工科大学の Li らは，フローキャッシュにおける連想度の検討を行っている [72]．Li らは実装コストを考慮すると連想度は 4 までが限界であると述べ，ダイレクトマップ方式，2way，4way セットアソシアティブ方式によるミス率をフルアソシアティブ方式と比較している．その結果，連想度を増加することでミスは大きく削減されており，4way キャッシュにおいてはフルアソシアティブとほぼ同等のミス数となることを示した．

また，国立交通大学の Ho らは IPv6 パケットに対するフローキャッシュの最適な構成を検討している [73]．IPv6 では，特にフロー情報サイズが大きくなるため，キャッシュのエントリ数を確保することが難しい．そこで，フローキャッシュのエントリ数を 256 から 16K に設定した場合の連想度を 1 から 4 に変化させ，ミス率を測定している．Ho らによると，ダイレクトマップ方式と 2way セットアソシアティブ方式を比較した際にはミス数の改善は最小で 4.5%から最大で 20%となる．これに対し，ダイレクトマップ方式と 4way セットアソシアティブ方式を比べると，最小でも 24%，最大で 45%のミス数改善がなされる．従って，4way セットアソシアティブ方式を用いることが最も効果的であるとしている．

一方で，フローキャッシュにおけるフロー情報サイズを圧縮することでエントリ数を確保する手法も提案されている．大阪大学の宮原らは，一般的な 5 タプルによるフロー分類ではデータサイズが 104bit になってしまい，十分なキャッシュエントリ数が確保できない点を指摘している [74]．そこで宮原らはフロー分類には送信元 IP アドレス，宛先 IP アドレス，ポート番号の 3 タプルの 80bit で十分であるとしている．ポート番号は送信元ポート番号，宛先ポート番号のうち，小さいほうの番号だけでよいと述べている．これはポート情報は，1〜1023 までのウェルノウンポートとそのクライアント側の適当なポート番号の組み合わせが多く，小さいほうのウェルノウンポー

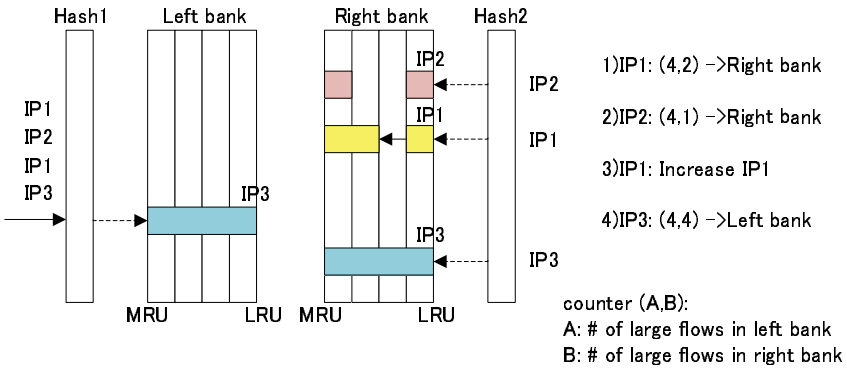


図 3.11: 二つのキャッシュバンクにおける追い出しアルゴリズムの概要 [4]

トによりアプリケーションを特定することが可能だからである．しかしながら，近年のパケットクラシフィケーションでは 3 タプルにより決定される処理が少ないことを考慮すると，やはり少なくとも 5 タプルには対応する必要があると考えられる．

ミス率の改善 3: エントリ追い出しアルゴリズムの検討

前述したように，フローキャッシュでは 1 エントリのサイズが大きく，十分なエントリ数を確保することが困難である．これに対しフロー情報を圧縮する手法は適していないことを述べた．そこで，追い出しアルゴリズムとして経験的に用いられてきた LRU を改良することで，ミスを削減する手法が研究されてきた．

例えば，Liao らは，インデックス生成に関するハッシュ関数の検討と同時に，追い出しアルゴリズムの改良を検討している [4]．Liao らの調査によるとトラフィック中のパケットの宛先 IP アドレスは，その多くが 1 パケットのみにしか現れず，全体としてジップの法則に従う．ジップの法則とは出現頻度が k 番目の要素が全体に占める割合が 1/k に比例するという法則であり，パケットの宛先 IP アドレスのみならずフローの分布にも当てはまると述べている．そこで，ジップの法則を反映し 1 パケットのみに現れる宛先 IP アドレスを優先的に追い出す手法を提案した．

図 3.11 に提案されている追い出しアルゴリズムの概要を示す．本アルゴリズムでは，キャッシュ内に一度しか現れない IP アドレスに焦点を当て優先的に追い出す．そのため，キャッシュエントリ内に extra bit と呼ぶ 1bit を設け，エントリが一度でもヒットした場合にこの bit を立てることで，一度しか現れていない IP アドレス（unpopular flow）と複数回現れた IP アドレス（popular flow）の区別を行っている．新しいパケットが到着した際には，左右のキャッシュバンクのうち unpopular flow の数が多い方のバンクが選択され，最も追い出し優先度の高いキャッシュエントリを追い出す．そして，キャッシュヒットをした場合には，該当するエントリの追い出し優先度が 1 つ低くなるように入れ替える．このような追い出しアルゴリズムでは，新規エントリは LRU より優先的に追い出されるようになり，一度しか現れない IP アドレスも追い出されやすくなる．左右のキャッシュバンクに存在する unpopular flow の数が同じ場合には左側のバンクからエントリを追い出す．

Liao らは二つのメモリバンクに対し，提案したハッシュ関数によるアクセスと追い出しアルゴリズムを併せて用いることで 15%程度キャッシュミス率を改善できるとした．しかしながら，提案されている追い出しアルゴリズムは複雑であり，ハードウェア実装する上では実装コストを検

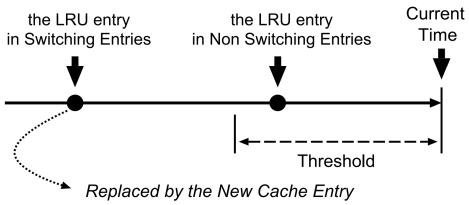


図 3.12: Weighted Priority LRU の概略 [5]

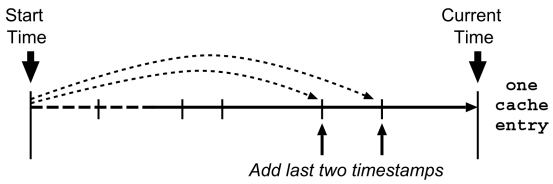


図 3.13: L2A scheme の概略 [5]

討しなければならない．まず，左右のバンクで unpopular flow 数を管理しなければならない．これには別途メモリを設けるか，キャッシュアクセスを事前に一度行い unpopular flow 数を数える必要がある．どちらの場合であっても，エントリ探索に 2 サイクルを要するためスループットは大幅に低下すると考えられる．

また，Kim らはパケット転送にラベルスイッチング技術を用いた場合の処理性能を向上させるため，テーブルキャッシュと同様の観点から MPLS キャッシュに適した追い出しアルゴリズムを提案した [5]．ラベルスイッチングでは全てのフローパケットはスイッチング処理で転送され，フローではない単体パケットのみ，ルーティング処理で転送されることが望ましい．Kim らはキャッシュエントリをノンスイッチングエントリとスイッチングエントリに分けることで，これらのパケットの扱いを区別している．初めて挿入されるエントリはノンスイッチングエントリに，ノンスイッチングエントリのパケットがフローと分類された場合にスイッチングエントリへと変える．ノンスイッチングエントリとスイッチングエントリの LRU エントリの参照された時間が共に同じだった場合，前述したように重要度の差からノンスイッチングエントリが追い出されるべきである．だが，ノンスイッチングエントリを優先的に追い出すことは新たにスイッチングエントリになるエントリを減らしてしまうため，場合によってはスイッチングエントリに追い出し優先度を持たせるべきである．そこで本論文ではスイッチングエントリとノンスイッチングエントリの追い出し優先度を適切に制御する，二つの追い出しアルゴリズムを提案している．図 3.12 および図 3.13 に提案手法の概略を示す．

Kim らの提案手法はともに LRU を改良し，ノンスイッチングエントリとスイッチングエントリで別々の優先度設定を与える手法である．一つは，閾値を設け LRU エントリの参照されてからの経過時間が閾値以内の場合と閾値を超えた場合で，別々の追い出し優先度を与える Weighted Priority LRU Scheme である．キャッシュ追い出し時に，まずノンスイッチングエントリの LRU エントリを検索し，そのエントリが参照されてからの経過時間を見る．経過時間がある閾値以内であった場合，次にスイッチングエントリ内の LRU エントリを参照する．そして，二つのエントリの内，参照されてからの経過時間が長いほうのエントリを新規エントリと入れ替える．ノンスイッチングエントリにおける LRU エントリの参照されてからの経過時間が閾値を超えていた場合は，ノンスイッチングエントリから入れ替える．挿入して間もない新規エントリはフローとなりスイッチングエントリへと移行する可能性が高いため，このようにすることで新規エントリに対しては追い出し優先度が低く，古いスイッチングエントリを優先的に追い出せるようになる．閾値を 0.5 秒に設定したシミュレーションによると，キャッシュエントリ数が少ないときにはキャッシュミス

率はほぼ LRU と変わらないが，エントリ数が 5K 以上のときには LRU に対し最大で 23%程キャッシュミス率を改善することが可能である．

もう一つの手法は，フローパケットの時間的局所性を考慮し，エントリの過去二つ分のタイムスタンプ情報を用いた L2A（Last 2 Add） scheme である．LRU では一つ前の参照があった時間をタイムスタンプとして保存し，タイムスタンプの値が小さい程，昔に参照されたエントリであるとしている．しかし，LRU ではフローパケットが短時間に連続して来ているのか，長い時間間隔で来ているのか等のフロー特性まで考慮することができない．そこで，L2A scheme では図 3.13 に示すように，過去二つ分のタイムスタンプを足し合わせ，この値が小さいエントリから順に入れ替える．L2A scheme では最近 2 個分のパケットが長い間送られていないエントリから優先的に追い出されるようになる．本手法ではキャッシュのエントリ数によらずキャッシュヒット数を上げることができ，0.5K エントリ，1K エントリにおいて最大で 31%程キャッシュミス率を改善することが可能である．

論文 [5] で提案された二つの手法は，ハードウェアとして実装する上ではコストが大きい．まず Weighted Priority LRU では各エントリが参照されてからの経過時間を情報として持つ必要がある．キャッシュ上でこのような時間情報を管理し，更新するには 1 エントリあたりに多大な bit を追加しなければならず，更にキャッシュアクセス回数が増えるため，大容量のトラフィックを高速に処理しなければならないフローキャッシュには向かない．また，論文中で，キャッシュエントリ数が少ない場合にキャッシュミス率は LRU と同程度となると述べられていることからも，高速なメモリアクセスを要することから小規模メモリとなるフローキャッシュ機構においては，LRU を用いた方が効率的である．L2A scheme はエントリ数の少ないキャッシュであっても有効な手法であるが，過去二つ分のタイムスタンプ値を持たなければならず，実装コストが Weighted Priority LRU よりも高くなることからフローキャッシュに適しているとは言えない．

3.1.5.3 パケット処理キャッシュ

日立製作所の奥野らは，高スループットなパケット処理を可能とするパケット処理キャッシュ搭載ネットワークプロセッサ P-Gear を提案した．パケット処理キャッシュでは，フローキャッシュと同様に特定のヘッダフィールドの値の組み合わせ（主に送信元 IP，宛先 IP，送信元ポート番号，宛先ポート番号，プロトコル番号の 5 タプルの値）が等しいパケット群を 1 つのフローと定義する．フローキャッシュと異なる点は，同一フローパケットに対して等しい結果となる全ての処理（テーブル検索結果のみならずヘッダ書き換えやカプセリングなども含む）をキャッシュに保存することで，パケット転送処理全体を高速化する点である [75]．P-Gear はハードウェアアーキテクチャや必要資源に関して考察が精練されており，本研究において参考となる部分が多い．そこで，以降では主に P-Gear のパケット処理キャッシュ機構周辺に焦点を当て，詳述する．

P-Gear のデータ構造:

P-Gear では，受信パケットから処理に必要な情報を抽出し，トークンとして保持する．トークンとして，U-Info，A-Info，E-Info，R-Info，P-Info の 5 つの情報を生成する．5 つの情報それぞれの概要を以下に示す．

表 3.9: E-Info の代表例

名称	抽出フィールド	必要 bit 数
IPv4 フォワーディング	宛先 IPv4 アドレス	32
IPv6 フォワーディング	宛先 IPv6 アドレス	128
優先度考慮 IPv4 転送	送信元・宛先 IPv4 アドレス	64
5tuple（IPv4）	送信元・宛先 IPv4 アドレス 送信元・宛先ポート番号，プロトコル番号	104

- U-Info：パケット固有情報
  U-Info（Unique Information）は，受信パケットを解析し得られるそのパケット固有の情報である．20〜30bit 程度の情報を想定している．U-Info には P-Gear 内でのパケットの整理番号やそのパケットを一時的に保存しておくためのパケットバッファのアドレス，またエラー情報や例外情報等を記録する．U-Info はパーサ部によりパケットヘッダの解析と共に生成される．個々のパケットの識別に使われるため，主にキャッシュ内容を元のパケットに適用する際に用いられる．

- A-Info：プロトコル解析情報
  A-Info（Analysis Information）は，受信パケットのプロトコルに関する情報である．10〜30bit 程度の情報を想定している．A-Info はネットワーク通信における各レイヤヘッダのプロトコル（IPv4，IPv6，VLAN，MPLS 等）に関連する情報を必要な分だけ解析した情報である．P-Gear の振る舞いを決定するコントロールレジスタにより，どのレイヤのどのようなプロトコルまで解析するかを変化させることができる．

- E-Info：フロー抽出情報
  E-Info（Extracted Information）は，受信パケットの所属フローを分類するための情報である．100〜300bit 程度の情報を想定している．情報抽出するヘッダフィールドは A-Info に従って選択される．抽出するフィールドについて，いくつかの代表例を表 3.9 に示す．最も基本的に用いられるフロー分類は，レイヤ 3 の送信元 IP アドレスおよび宛先 IP アドレス，プロトコル番号，レイヤ 4 の送信元ポート番号および宛先ポート番号の 5tuple を用いた分類であり，この場合に必要となるデータは 104bit となる．IPv6 の場合には，IP アドレスが 32bit ではなく 128bit となるため，296bit となる．VLAN タグの VLAN ID や，MPLS のシムヘッダ情報の抽出なども行われる．E-Info は主にキャッシュのタグとして用いられる．また，テーブルルックアップやヘッダ生成処理等にも使用する．

- R-Info：ヘッダ置換情報
  R-Info（Replaced Information）とは，パケット処理を行う際のパケットヘッダ置換情報やカプセリング化用の追加情報であり，PLC が記録する処理結果の一部である．300〜400bit 程度の情報を想定している．R-Info には，例えば通常の IP フォワーディングの場合，置換用の送信元および宛先 MAC（Media Access Control）アドレスを記録する．パケットが VLAN

や MPLS 等のネットワークプロトコルを利用している場合には，カプセリング化するための新規ヘッダや置換用のシムヘッダ等を記録する．

- P-Info：パケット処理情報
  P-Info（Processing Information）とはパケットの処理情報であり，キャッシュが記録する処理結果の一部と処理の適用方法である．20〜100bit 程度の情報を想定している．P-Info はパケットの処理結果として，ルーティングテーブルや ACL 等，各種テーブルの検索結果等を含む．また，パケットの処理結果を適用する方法として，パケット修正や廃棄処理命令等の指示子を含む．例えば，ヘッダに対するビット列操作の方法（操作種類，操作位置，操作長），CRC 演算やチェックサム演算の有無，パケット廃棄の有無，優先度処理の指定等である．

C-Engine の構成:

C-Engine は P-Gear の主要な機能を司り，パケット処理のスループットを決定づけるパケット処理キャッシュ機構を含む．図 3.14 に C-Engine のアーキテクチャを示す．C-Engine では，A-Engine においてパケットから抽出されたトークンをもとに，パケットの処理結果とその適用方法を格納したキャッシュの検索を行う．キャッシュ検索がミスした場合には，P-Engine へとトークンを転送し，PE を介したパケット処理を行った後に処理トークンを C-Engine へと戻す．キャッシュ検索がヒットした場合のトークンは，キャッシュに格納された処理情報と共に R-Engine へと転送され，元パケットの適用がなされる．また，P-Engine での処理がなされたトークンも C-Engine へと戻した後，キャッシュ内容の更新と同時に R-Engine へと転送される．A-Engine と P-Engine，R-Engine の詳細については，本研究との関連が低いことから割愛する．

C-Engine はパケット処理キャッシュを構成する Process Learning Cache（PLC）と，PLC がミスした際に，PE を割り当てるためにパケットがブロッキングされるのを防ぐ Cache Miss Handler（CMH）により構成される．以降ではそれぞれの機能について詳述する．

Process Learning Cache (PLC):

Process Learning Cache（PLC）はパケット処理キャッシュ機構として，各種テーブル検索結果やカプセリングにおけるヘッダ生成といった R-Info と，その結果の適用方法である P-Info を格納する．PLC におけるキャッシュエントリの登録は，C-Engine へ転送されたトークンがキャッシュミスした際に P-Engine へと送られ，P-Engine において処理のなされたトークンが再び C-Engine へと転送された後となる．そのため，パケットを受信してからエントリ登録がなされるまでに若干のタイムラグが発生するが，この点については後述する CMH で補うことが可能である．PLC にエントリ登録後は，後続の同一フローのパケットトークン（A/E-Info が同一）については P-Engine を一切利用せずに PLC の内容をそのまま適用することでヘッダ処理を行うことができる．ネットワークトラフィックの時間的局所性が強く期待できる場合には，大多数のパケットに対して PLC から直接 R/P-Info を与えることができるため，P-Engine（通常のネットワークプロセッサの PE 集積部分）のスループットが低くても P-Gear は高スループットなパケット処理を実現できる．

PLC は通常のキャッシュメモリと同様に，タグメモリである PLC タグメモリとデータメモリである PLC データメモリにより構成される．図 3.15 に PLC のアーキテクチャを示す．A-Engine から転送されたトークンは C-Engine においてハッシュ化され，生成されたハッシュ値を PLC タグ

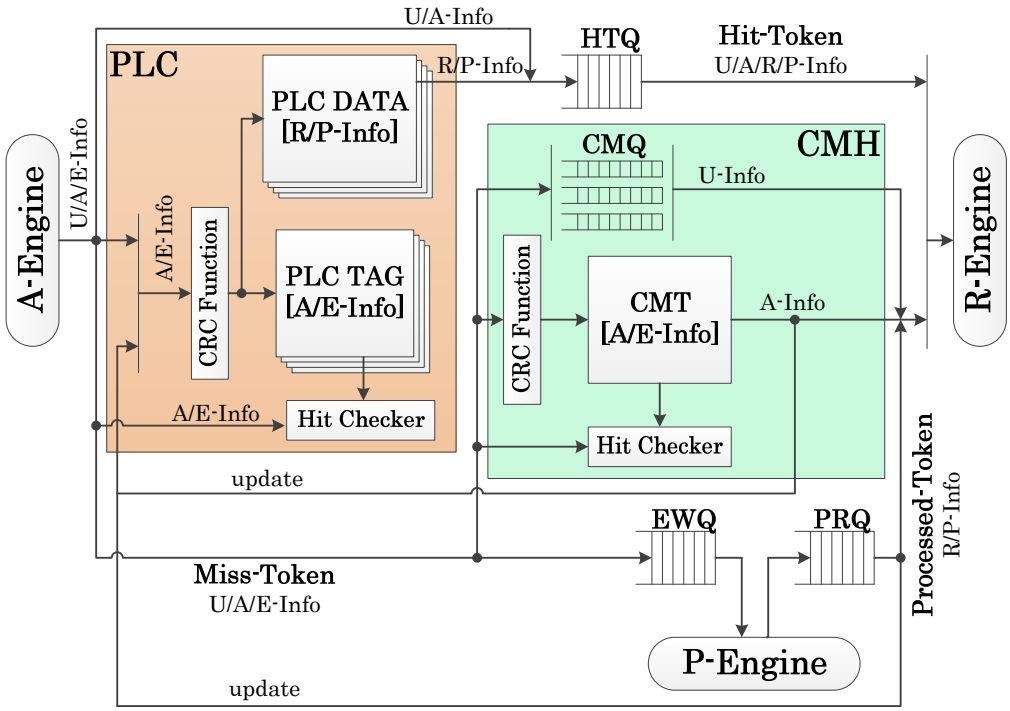


図 3.14: C-Engine のアーキテクチャ

メモリおよび PLC データメモリにおけるインデックスとして利用する．ハッシュについてはこの後に述べる．

まず，PLC タグメモリは受信したトークンを元に，そのパケットが所属するフローのエントリが PLC 中に存在するかどうかの判定を行う．トークンの A/E-Info と 1bit の Valid bit（タグエントリが有効ならば 1）が格納されている．従って PLC タグは 110〜330bit 程度であると予想される．トークンから得られたハッシュ値をアドレスとしてアクセスしたエントリの A/E-Info と，トークンの A/E-Info が等しく，かつ Valid bit が 1 の時に ”キャッシュヒット ”となる．エントリの A/E-Info とトークンの A/E-Info が異なるか，Valid bit が 0 であったときは ”キャッシュミス ”となる．この判定は Hit Checker により行われる．ここでキャッシュミスした場合，受信トークンは通常通り PE を用いて処理を行うために CMH へと送られる．また，PLC タグメモリのアクセスと同様に，トークンから得られたハッシュ値をアドレスとして PLC データメモリへのアクセスも並列に行われる．PLC データメモリは各フローにおける R/P-Info を格納している．従って，PLC データサイズは 320〜500bit 程度であると予想される．PLC データメモリから得られる R/P-Info は，Hit Checker の出力を受け，PLC タグメモリがキャッシュヒットだった場合に Hit Queue（HTQ）へと転送される．これらのデータは HTQ に入る前に受信トークンから U/A-Info を加え，Hit-Token として蓄えられる．スケジューラにより HTQ 中のトークンは R-Engine へと適時転送される．

従来の P-Gear の研究によると，PLC のキャッシュ方式は 4way セットアソシアティブにおいて最も性能の効率が良くなることがわかっている [6]．よって図 3.14 および図 3.15 では 4way セットアソシアティブ方式の PLC を表記している．

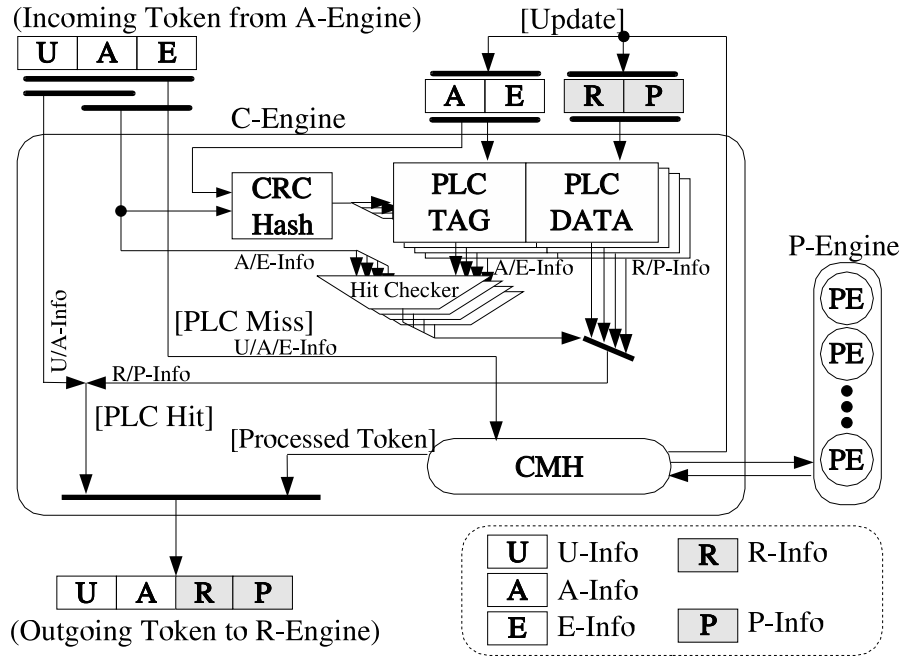


図 3.15: PLC のアーキテクチャ[6]

Cache Miss Handler (CMH):

マルチスレッド環境において，あるフローのパケットを PE で処理中に同一フローの後続パケットが到着した場合，後続パケットは再度キャッシュミスを起こし，複数の PE に同一処理が割り当てられるという問題点がある．このような PE での処理から PLC エントリが更新されるまでのタイムラグによる，処理の重複化は PE のリソースを無駄に使用し，後続パケットの処理をブロッキングすることで致命的な性能低下を引き起こしうる．そこで，CMH では PLC においてキャッシュミスしたトークンに対して，PE で処理中のトークンを管理し，処理中のフローの後続トークンをキューへ蓄え，PE での処理後に処理内容をキューへと適用することで，PLC エントリが更新されるまでの遅延を隠蔽する．図 3.16 に CMH のアーキテクチャを示す．

CMH は，PLC ミスしたトークンを連続的に受信し PLC 参照をノンブロッキング化する機能，受信したトークンを P-Engine の PE へと割り当てる機能，P-Engine から処理済みのトークンを受信し PLC を更新する機能，処理済みトークンを R-Engine へと転送する機能を有する．各機能においてテーブルやキューを持ち，これらの待ち行列はスケジューラにより管理される．表 3.10 にそれぞれのテーブル，キューの役割と格納するトークン情報を示す．

PLC においてキャッシュミスしたトークンの A/E-Info は，Cache Miss Table（CMT）へと送られる．CMT では，P-Engine で処理中のフローの管理を行う．まず受信トークンの A/E-Info のハッシュ値をインデックスとして CMT を参照し，A/E-Info が一致するかどうかをヒットチェッカで判定する．CMT ミスであった場合には当該フローの A/E-Info を CMT に登録し，CMT エントリと対応する Cache Miss Queue（CMQ）に U-Info を登録する．ここで登録されたエントリは，P-Engine での処理済みトークンが C-Engine へと戻ってきた際に解放される．よって，CMT での検索ミスは P-Engine で処理中の同一フローが存在しないということを示す．CMT ミスしたトークンの A/E-info

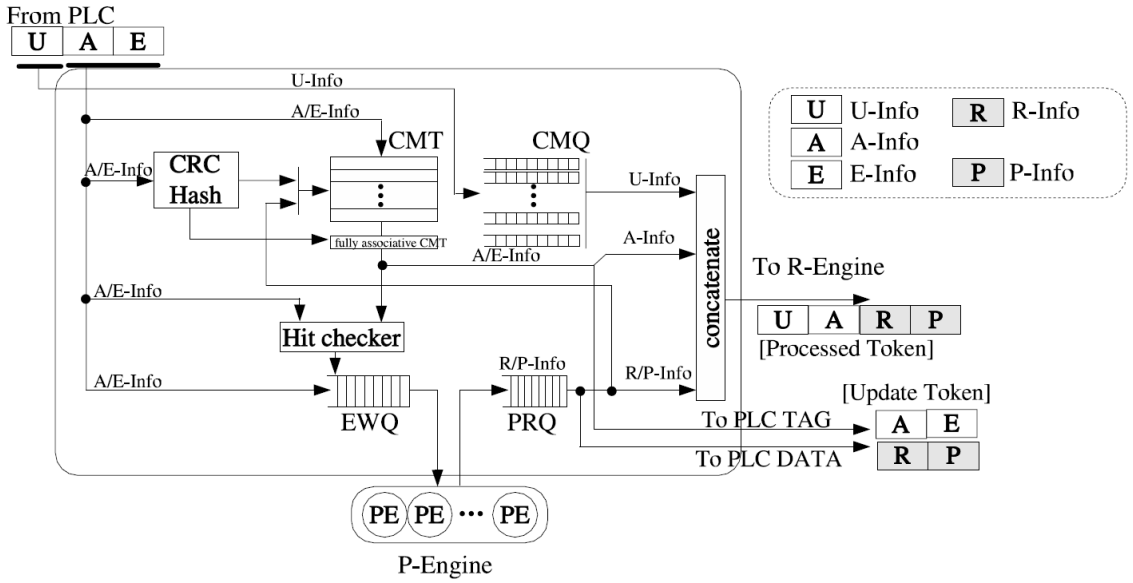


図 3.16: CMH のアーキテクチャ[7]

表 3.10: CMH における各キューの役割

名称	格納トークン	機能の説明
CMT （Cache Miss Table）	A/E-Info	PLC ミス中の全フローを管理. 先着トークンのみ P-Engine で処理.
CMQ （Cache Miss Queue）	U-Info	処理中フローの後続パケットを蓄える. CMT のエントリに対する CMQ が必要.
EWQ （Execution Waiting Queue）	U/A/E-Info	P-Engine へ送るトークンを蓄える.
PRQ （Processing Result Queue）	R/P-Info	PE での処理済みトークンを蓄える.

は P-Engine で処理を行うため，Execution Waiting Queue（EWQ）へと送られ，スケジューラの管理のもと P-Engine へと転送される．検索結果がヒットであった場合には，ヒットした CMT エントリと対応する CMQ の末尾に U-Info を格納する．ヒットしたトークンは，先行する同一フローのトークンが P-Engine で処理中であるため，P-Engine へと転送する必要はない．従って，P-Gear におけるキャッシュヒット数とは PLC と CMH におけるキャッシュヒット数の和であり，最終的に CMH でキャッシュミスしたパケットが P-Engine へと割り当てられる．

P-Engine で処理がなされ R/P-Info が得られたトークンは，C-Engine の Processing Result Queue（PRQ）へと蓄えられる．出力処理では，この PRQ をチェックし，CMQ に有効なエントリがある場合先頭から一つ取り出し，対応する CMT エントリの A/E-Info を読み出す．そして，PLC のインデックスを CRC ハッシュにより再計算し，PLC タグを A/E-Info で，PLC データを R/P-Info で

表 3.11: CMT 及び各キューのハードウェア構成

名称	項目	構成
CMT	連想度	4way セットアソシアティブ + 小規模フルアソシアティブ
	エントリ数	1,024（256 エントリ/way）+ 8（フルアソシアティブ）
	CMT 容量	44.0KByte（A/E-Info，valid bit 格納，349bit×1K+8）
CMQ	本数	1,024 + 8（CMT の各エントリに対応）
	エントリ数	16
	総 CMQ 容量	44.4KByte（U-Info 格納，$22bit \times 16 \times 1,032$）
EWQ	エントリ数	256
	EWQ 容量	11.6KByte（U/A/E-Info 格納，$370bit \times 256$）
PRQ	エントリ数	256
	PRQ 容量	11.6KByte（P/R-Info 格納，$366bit \times 256$）
HTQ	エントリ数	128
	HTQ 容量	6.25KByte（U/A/P/R-Info 格納，$400bit \times 128$）

更新する．更に，対応する CMQ に保持されている U-Info，CMT の A-Info，PRQ の R/P-Info を結合して処理済みトークンとして R-Engine に発行する．当該 CMQ が空になるまで発行を繰り返した後，CMT の当該エントリを解放して CMH 出力処理を終える．

C-Engine では，CMQ から発行される処理済みトークンと PLC ヒットし HTQ から発行されるトークンがある．これらが無秩序にトークンを R-Engine へと発行すると，同一フローのパケットの順序が入れ替わる可能性がある．ルータにおいて順序の入れ替わったパケットの順番整列操作は受信ホストにおいて CPU の浪費と遅延増加に繋がるため，望ましくない．そこで，CMH では HTQ よりも CMQ のトークン発行優先度を高くし，スケジューラに管理させることでパケット順序の入れ替わりを防ぐ．

奥野らの研究では，CMH や各キューのメモリサイズの妥当性に関するシミュレーションがなされた [7]．表 3.11 にそれぞれの必要メモリサイズ，最適なメモリの構成について示す．既存の研究によると，CMT の構成は 1,024 エントリ ×4way のセットアソシアティブと 8 エントリのフルアソシアティブが良く，CMQ は 1 つの CMQ あたり 16 エントリ格納できれば十分であるとしている．また，その他のキューに関しても EWQ，PRQ は 256 エントリ，HTQ は 128 エントリ格納できれば十分であると述べている．

最終的に奥野らは提案機構を FPGA 上へと実装し，実ネットワークトレースを用いてキャッシュヒット率の測定を行っている．シミュレーションの結果，概ね 70%程度のヒット率が得られており，これにより，従来の 1/3 程度のパケット処理性能を有したパケット処理機構で従来と同等のスループットが得られることを示した．

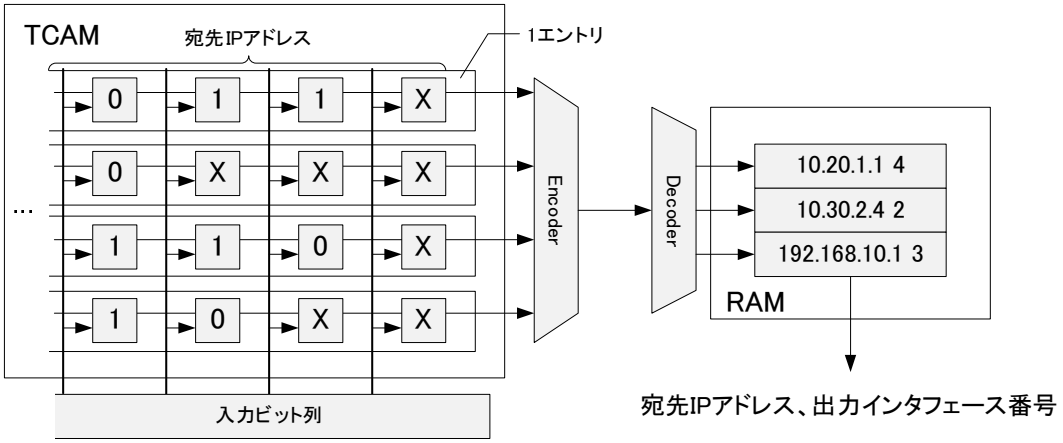


図 3.17: TCAM の構成および TCAM を用いたルーティングテーブル検索の概要

3.1.5.4 TCAM を用いたテーブル検索

Content Addressable Memory（CAM）はデータ探索の高速化に特化したメモリで，全てのデータ bit 毎に比較器を用意することにより，1cycle でエントリマッチングを完了する．CAM のエントリアクセスは，一般的に用いられてきたアドレス指定ではなく，格納されたデータとのマッチングにより行う．CAM はマッチングしたデータの格納されたアドレスを返す．特に，Ternary CAM（TCAM）はデータとして 1 と 0 のみならず don't care まで対応することにより，ワイルドカードの用いられたデータであっても 1cycle で探索を完了する．

本項ではルーティングテーブル検索を TCAM により行う場合について説明する．TCAM の構成および TCAM を用いたルーティングテーブル検索の概要を図 3.17 に示す．TCAM 方式では，TCAM のメモリサイズを圧縮するために，TCAM と RAM の 2 回に分けてデータを探索する．まず，宛先 IP アドレスをキーとして TCAM のエントリが探索される．そして，一致したエントリの中で最もプレフィックス長の長いエントリが格納されたメモリアドレスから RAM のインデックスが得られる．ここで得られたインデックスをもとに，RAM からネクストホップアドレスおよび出力インタフェース番号が読み出される．従って，TCAM 方式は 2cycle により所望のルーティングテーブル検索結果を得ることができる．

このように，TCAM を用いたテーブル検索は高速であることから，近年のルータではルーティングテーブルに限らず多くのテーブルに用いられている [76]．TCAM 方式におけるルーティングテーブル検索のスループットは，エントリ数によらず TCAM のアクセス遅延により決定される．各社ルータベンダが用いている TCAM に関して詳しい情報は公開されていないが，既存の TCAM のアクセス遅延は 5ns 程度である [77]．TCAM のエントリ検索性能はアクセス遅延の逆数で計算でき，200Mpps（Packet per Second）となる．このことから，最短パケット長 64Byte のパケット処理におけるスループットを概算すると，TCAM 方式におけるテーブル検索スループットは 100Gbps 程度となる．

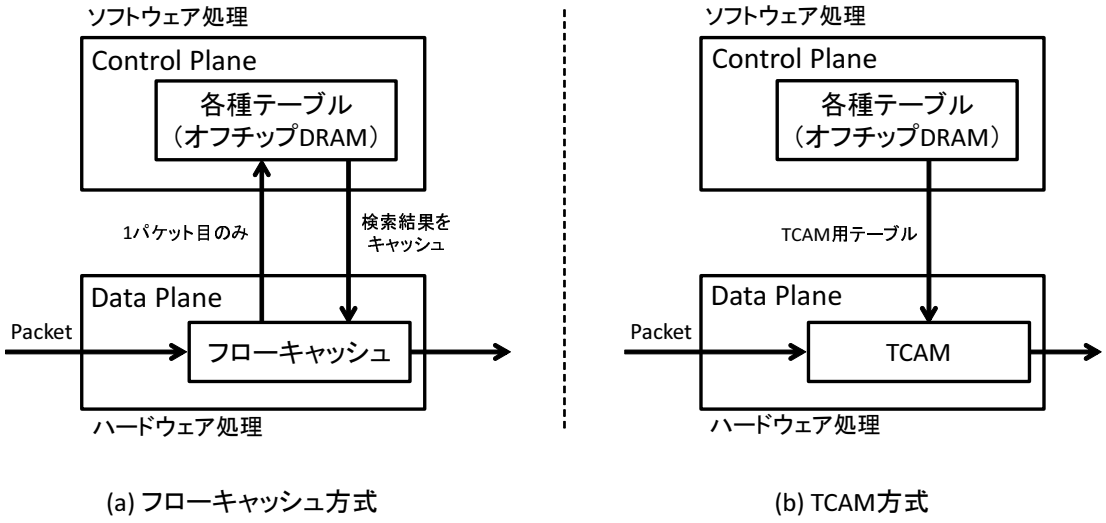


図 3.18: 各テーブル検索方式の概要

3.1.5.5 キャッシュから TCAM への変遷

現在のコアルータにおけるテーブル検索手法はフローキャッシュ方式から TCAM 方式へと移り変わった．この理由として，フローキャッシュのミスペルティや，ネットワーク帯域の向上によるフローの増大などが挙げられる．フローキャッシュ方式が用いられてきた背景には，それまでのソフトウェアによるテーブル検索遅延の大きさがある．一般にソフトウェア処理で用いられる主メモリは，オフチップの DRAM として実装されており，50ns から 100ns 程度の大きいアクセス遅延が生じる．これに対し，フローキャッシュは高速なオンチップ SRAM で実装可能であり，そのアクセス遅延は数 ns 程度であった．そのため，キャッシュヒット時には数 ns で処理が可能であった．しかしながら，キャッシュミス時にはソフトウェアで処理する必要があり，特に 1 パケット目は必ずミスが生じるため，その度にミスペナルティとして大きい遅延が生じる．2000 年以降のフロー数の増加は顕著であり，これによってキャッシュミス数が増え，ミスペナルティが無視できない．

これに対して，TCAM 方式は近年の進歩したメモリ技術を潤沢に用いる．フローキャッシュ方式と TCAM 方式の概要を図 3.18 に示した．TCAM 方式は，全てのエントリを 1cycle で探索可能な TCAM メモリに，主メモリのテーブルから予め必要となる全てのエントリを格納することで，フローの 1 パケット目であっても高速に処理することが可能となる．従って，TCAM 方式ではミスペナルティを気にすることなく，常に高速にテーブル検索が行える．

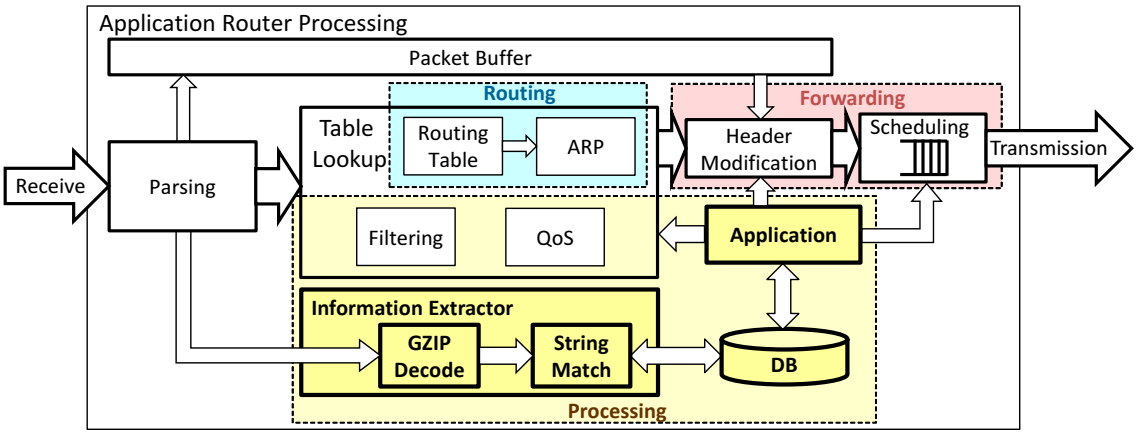


図 3.19: アプリケーションルータにおけるパケット処理プロセス

3.2 アプリケーションルータのパケット処理機構

はじめに，一般的なルータに焦点を当て，そのパケット処理機構について述べた．本節では，アプリケーションルータに焦点を当て，アプリケーションルータが持つ独自の処理機構に関して述べる．

図 3.19 に，本論文で想定するアプリケーションルータのパケット処理プロセスを示す．一般的なルータにおけるパケット処理プロセスを示した図 3.1 と比べ，アプリケーションルータではプロセッシング機能における処理が増す．具体的には，情報抽出機構として，GZIP 圧縮された HTTP パケットを解析するための GZIP 展開処理とペイロードから必要な情報を抽出するための文字列探索処理の 2 つの処理機構，文字列探索処理結果を保存するためのデータベースエンジン，データベースに蓄えられた情報を用いるアプリケーションが追加される．

情報抽出機構における GZIP 展開処理は，多くのアプリケーションルータでは考慮されていない．しかしながら，3.2.2 項で後述するように近年のネットワークではパケットの GZIP 圧縮化が普及しはじめており，このようなパケットに対応するためにも GZIP 圧縮パケットの展開機構は必須であると考える．情報抽出機構の文字列探索処理は，アプリケーションルータが一般的に持つ機能である．CCN ルータにおいてコンテンツを抽出する機構 [34] や，サービス指向ルータにおいて抽出ルールとのマッチングを行う機構 [78] がこれに該当する．また，データベースエンジンも多くのアプリケーションルータが持つ機構である．CCN ルータにおいて Content Store と呼ばれるコンテンツを格納する機構 [34] や，サービス指向ルータにおいて抽出ルールとのマッチング結果を格納する機構 [79] がこれに該当する．アプリケーション機構は，アプリケーションルータがサービスを提供するために当然持つべき機能である．以下では，それぞれの処理概要と既存研究について詳述する．

3.2.1 情報抽出機構

本項では，アプリケーションルータにおいて必要となる情報抽出機構を説明する．まず，ネットワーク経路上での情報抽出を行うには，パケット断片化の問題を考慮しなければならない．

3.2.1.1 ネットワーク経路上における情報抽出の概要

ネットワークにおいて，通信データは MTU（Maximum Transmission Unit）で指定されたサイズに分割され，パケットへと成形される．断片化されたパケットは，エンドホストでパケットヘッダの 5 タプル（送信元 IP アドレス，宛先 IP アドレス，送信元ポート番号，宛先ポート番号，プロトコル番号）情報およびシーケンス番号などをもとに一つのストリームとして再構築されることで，データとして読むことが可能となる．ここで，ストリームとは上記の 5 タプルにより決定される片方向のデータ通信を意味し，コネクションとは 2 つのストリームにより構成される双方向の通信を意味する．エンドホストでパケットの情報抽出を行う場合には，分割されたパケットが全部揃ってから情報を抽出することで，パケット断片化の影響を受けることなく処理が可能であった．しかしながら，ネットワーク経路上でパケットの情報抽出を行う場合には，パケットの断片化が問題となる．なぜならば，抽出対象となるデータ列が，分割されたパケットの境界上にまたがっている可能性があるからである．従来では，このようなパケット境界での情報抽出を可能とする手法が，特に Deep Packet Inspection の研究において多く提案されてきた．これらの研究手法をまとめると，ストリームの全パケットを保存する方式とストリームの一部を保存する方式の 2 種類に分けることができる．

3.2.1.2 ストリームの全パケットを保存する方式

第 2 章で述べた NIDS を実現するソフトウェアに Snort がある [80]．Snort は IDS ソフトウェアとしてパケットとシグネチャの照合を行い，アタックを検知する．一方で，Snort はプリプロセッサと呼ばれる強力な前処理機能を有しており，このプリプロセッサにおいてパケットの断片化を考慮した TCP 再構築機能が提供されている．Snort の提供する TCP 再構築機能の概要を図 3.20 に示す．

図 3.20 において，A3，C1 といったブロックはパケットを表す．ここで，A や C といったアルファベットはストリーム ID を，1 や 3 といった番号はストリーム中のパケットの番号を示す．従って，A3 のブロックはストリーム A の 3 番目のパケットであることを表す．Snort プリプロセッサは，FIN や RST によって TCP ストリームが終了するまでストリームの全パケットを保存しておく．後に到着するストリームの後続パケットは，保存された同ストリームのパケットと合わせ，ストリームとした上で処理される．保存されたパケットは TCP ストリームの終了と同時に解放される．

Snort と同様にパケット断片化に対応した情報抽出方法に，Paxson らの提案する侵入検知ソフトウェア Bro がある [20]．Bro もパケットの TCP フラグによって TCP ストリームが終了するまでパケットを蓄えることで，パケット境界にまたがる情報抽出を可能とする．また，花岡らは TCP

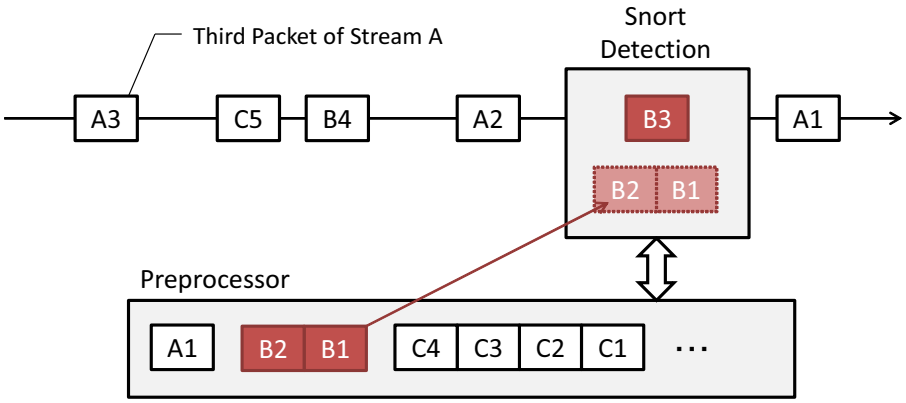


図 3.20: Snort による TCP 再構築機能

フラグに加え，パケットの識別子やフラグメント・オフセット，シーケンス番号を考慮することで，アウトオブオーダのパケットにも対応している [22].

このような方式による情報抽出は，全パケットを保存するためメモリ使用量が膨大となる．例えば 100Gbps の広帯域ネットワークでは，1 秒に 100Gbit 程度のデータを保存する必要がある上，動画などのマルチメディア通信では長い時間ストリームが続くことも考えられるため，実用性に乏しい．更には，毎回パケットを繋ぎ合わせてストリームとした上で処理するため，抽出対象となるデータサイズが大きく処理負荷が増大する．このようなことを考慮すると，本方式は広帯域ネットワークにおけるパケット情報抽出には向かない．

3.2.1.3 ストリームの処理途中状態を保存する方式

ストリームの全パケットを保存する方式に対して，ストリームの処理途中状態を保存する方式が提案されている．石田らは，パケットの断片化に対応したパケット毎の逐次情報抽出手法としてコンテクストスイッチを提案している [2]．コンテクストスイッチの概要を図 3.21 に示す．

コンテクストスイッチでは，ストリームの全パケットが保存されることはなく，文字列探索といった処理の途中状態がコンテクスト情報としてストリーム毎に管理される．パケットが到着した際には，まずパケットが属するストリームを 5 タプルにより決定し，コンテクスト情報を読み込む．そして，コンテクスト情報より得られた処理の途中状態から処理を再開することで，パケットの断片化に対応した情報抽出が可能となる．更に，正規の手順で終了しない TCP コネクションによって，コンテクスト情報が蓄積されていくことを問題点としている．そこで，コンテクスト情報の管理にタイムアウト値を設けることで，コンテクスト管理に要するメモリ使用量を削減している．論文 [2] によると，タイムアウトとして 300 秒が最適であるとしている．この理由として，Web サーバソフトウェア Apache の KeepAliveTimeout 値が 15 秒であること，Snort のタイムアウト値が 30 秒であること，300 秒のタイムアウトによって情報抽出の精度が低下しないことを挙げている．

菅原らは，石田らと同様に，パケット処理の途中状態を保存する情報抽出手法を提案している

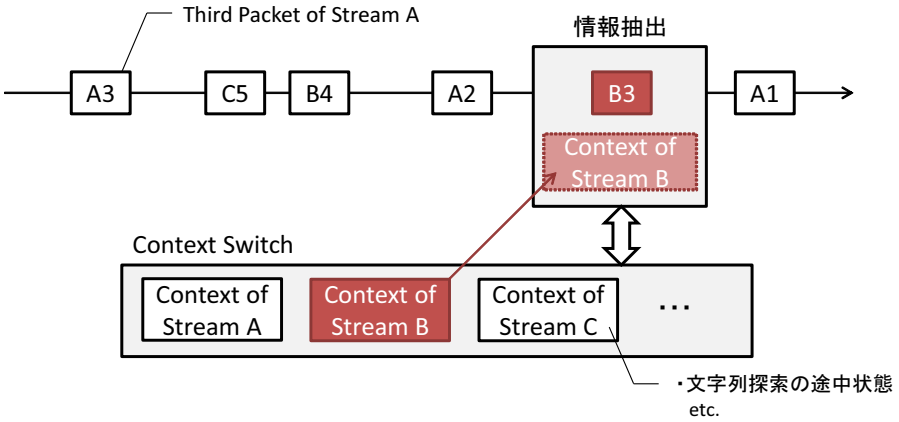


図 3.21: コンテクストスイッチによる TCP 再構築機能

[81]．菅原らの提案手法は，文字列探索処理の途中状態に加えて，パケット末尾の数オクテットをストリーム毎に保存する．更に，シーケンス番号を考慮することで，アウトオブオーダのパケットに対してはパケット先頭の数オクテットも保存する．菅原らは，このような情報を加えることで，アウトオブオーダのパケットに対応した情報抽出が行えると述べている．

このような，処理の途中状態を保存する方式は，全パケットを保存する方式に比べて格段にメモリ使用量が小さい．石田らは，アプリケーションにもよるが，100 のルールを抽出する場合には 1 ストリームに 100Byte 程度のコンテクスト情報を管理できれば十分であるとしている．アプリケーションルータの広帯域ネットワークへの対応を考えると，情報抽出方法は本方式が適していると言える．

3.2.2 GZIP 展開機構

アプリケーションルータにおける主な解析対象は，現在のネットワークにおいて大多数を占めている HTTP（Hypertext Transfer Protocol[82]）である．HTTP は web ブラウザと web サーバ間の通信に用いられるプロトコルであり，HTML（Hypertext Markup Language）などによって作成された，画像や音声を含む web ページを転送するために用いられる．特に近年では，その汎用性の高さからセンサデータの転送といった様々な用途にも用いられている．Borgnat らによると，2000 年から 2008 年において HTTP トラフィックがネットワークに占める割合は年々増加しており，2008 年において 60%程度を占めている [19]．また，我々が 2013 年に行ったトラフィック解析によっても，トラフィック全体のおよそ 78%のデータ量が HTTP トラフィックであることがわかっている [83]．

HTTP は 1999 年に提案されたバージョン 1.1[84] から，zlib[85] や GZIP[86]，DEFLATE[87] といったアルゴリズムにより転送データを圧縮することに対応している．HTTP データを圧縮し転送することで，ユーザは回線の帯域制限に対して効率よくデータの送受信を行うことが可能となった．タグ等の繰り返しが多い HTML 文書やプログラム，ピクセルの連続する画像等は圧縮の恩恵を受けやすく，平均して 30 %程度までデータを圧縮することが可能である [88]．

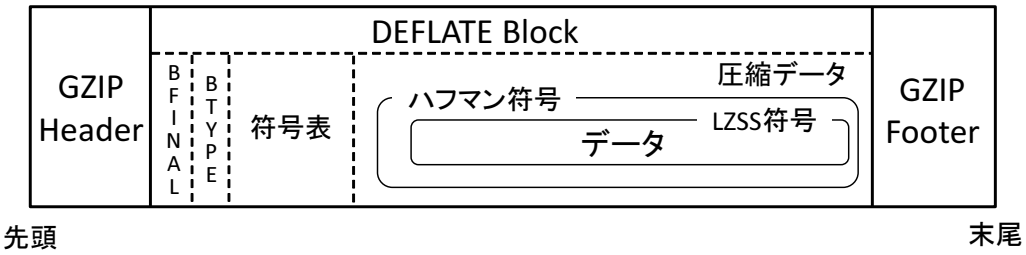


図 3.22: GZIP 圧縮データの全体像

このような HTTP データの圧縮プロトコルとして GZIP（GNU ZIP[86]）がデフォルトで用いられている．ネットワーク経路上で GZIP 圧縮された HTTP パケットから情報を抽出するには，圧縮データを一旦展開した上で情報を抽出する必要がある．そこで，アプリケーションルータには GZIP 展開機構が必要とされる．しかしながら，ネットワーク経路上において GZIP 圧縮データはパケット化されていることから，断片化された GZIP 圧縮パケットを逐次に展開することは困難であった．更に，処理の複雑さからスループットの獲得が困難であることも相まって，既存のアプリケーションルータに関する研究の多くでは GZIP 展開機構について議論されることが少なかった．トラフィック圧縮の需要は高まっており，更なる GZIP 圧縮パケットの普及を考えるとアプリケーションルータが GZIP 展開機構を持つことは重要である．本項では GZIP の基本的な処理について述べる．

3.2.2.1 GZIP 展開処理の概要

GZIP は誰でも使うことのできるフリーのアルゴリズムであり，古くから UNIX システムなどで用いられてきた．図 3.22 に GZIP 圧縮データの構造を示す．GZIP では，まず LZSS 符号 [89] を用いて入力データを非負整数に変換した後，ハフマン符号 [90] を用いてこれを二値符号化する．このように 2 種類の符号法を併用した DEFLATE アルゴリズム [87] を用いることで，圧縮速度および圧縮率の点でバランスの良い圧縮を実現している．2 種類の符号化と圧縮時の情報を組み込むことで，GZIP 圧縮データの構造は複雑化している．GZIP 圧縮プロセスは以下に従う．以降では，GZIP 圧縮に伴う各処理について順を追って述べる．

1. データの LZSS 符号化
2. LZSS 符号のハフマン符号化
3. ハフマン符号における符号表の添付
4. GZIP ヘッダの添付

GZIP 圧縮処理 1: LZSS 符号化プロセス

LZSS 符号は J.Ziv と A.Lempel が 1977 年に発表した LZ77 符号 [91] および 1978 年に発表した LZ78 符号 [92] を改良した圧縮符号である．これらの符号では，圧縮対象の直前数 KB 分のデー

タを辞書として用いて繰り返し文字列を検出する．LZ77 符号では辞書の中から圧縮対象データ列と同じデータ列を発見した場合，圧縮対象データ列を（一致位置，一致長，次の不一致データ）という 3 つの値に置き換えていた．しかしこの方法では一致がなかった場合には（0，0，不一致データ）と一致位置，一致長の分だけ冗長となる．そこで LZSS 符号では，一致があった場合（1，一致位置，一致長）とし，一致がなかった場合（0，不一致データ）と符号語を最適化することで圧縮率の向上を図っている．つまり，最初に一致したかどうかの判定に 1bit を用いる．以下では厳密な形でアルゴリズムを記述する [93]．まず，入力データの部分系列を（3.1）式で表す．

長さ M の入力データ系列 $x(1, M)$ の中で，既に $x(1, s-1)$ の符号化が終了しているとして，$x(s, T)$ の符号化を考える．この時大きさ N のバッファには，辞書として x_s の直前のデータ系列 $x(s-N, s-1)$ が格納されている．一般に GZIP では圧縮対象サイズ $T-s$ は 258Byte，辞書の大きさ N は 32KB となる．LZ77 符号では以下により入力データ系列の符号化を行う．

1. 繰り返しの探索:
   （3.2）式で表される，$x(s, T)$ に最も長く一致する部分データ系列を辞書内で探索する．

2. 符号語への置き換え:
   上記の一致長と戻り距離を用いて，$x(s, s+t)$ に対する符号語を $< s-m, l_{st} >$ とする．

LZSS 符号では，上記の 2 に変更を加えて一致長 l_{st} に応じて符号化モードを選択する．$l_{st} \geq \alpha$ のとき，$< 1, l_{st}-\alpha, s-m >$ を符号語とする．$l_{st} < \alpha$ のとき，< 0，先頭の一文字 $>$ を符号語とする．前者をコピーモード，後者をリテラルと呼ぶ．α は定数であり，GZIP では $\alpha = 3$ としている．以下では具体例を示す．

図 3.23 のように，文字列”HTML5HTML”を想定する．後に出現する文字列”HTML”は文字列の長さが 4 であり，また 5 文字前に同じ”HTML”という文字列が出現している．このとき，後に出現した”HTML”は符号語 < 1, 1, 5 > と置き換えられ,「長さ (1+3)」の同一文字列が辞書中の「戻り距離 5」の位置に存在することを示す．

$$x(i, j) = x_i x_{i+1} \cdots x_j \tag{3.1}$$

$$x(s, s+t) = x(m, m+t) \quad (1 \leq m \leq s-1) \tag{3.2}$$

GZIP 圧縮処理 2: LZSS 符号のハフマン符号化プロセス

ハフマン符号は 1952 年に D.Huffman によって開発された圧縮符号である．対象文字列の文字出現頻度を用いて出現頻度の高い文字を短いビット列に，出現頻度の低い文字を長いビット列に変換することで文字列を圧縮する．このハフマン符号化プロセスにおいて，文字列の変換は，文字とハフマン符号の対応を表した符号表を用いて行われる．符号表は，文字列に出現する文字頻度をもとにハフマン木と呼ばれるバイナリツリーを作成し，更にハフマン木を変換することで作成される．以下では，GZIP におけるハフマン符号化プロセスについて述べる．

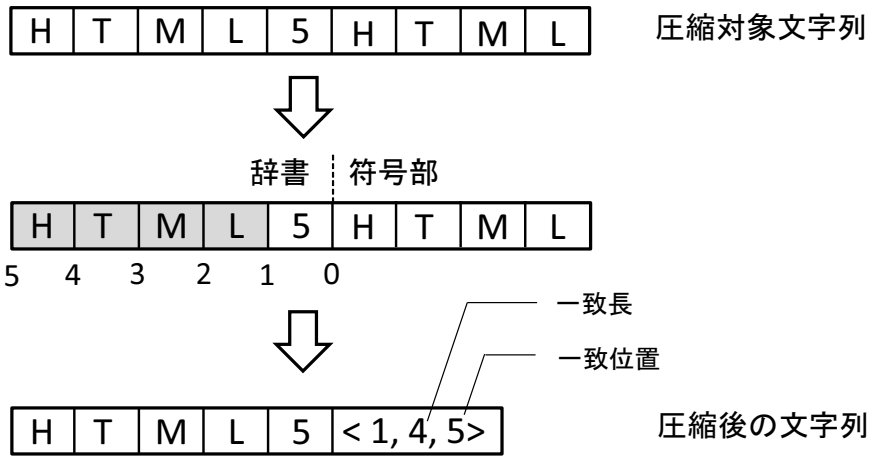


図 3.23: LZSS 圧縮の例

GZIP の LZSS 符号化プロセスにおいて出力される符号語は，コピーモードの場合 < 1，一致長 ，戻り距離 > であり，リテラルの場合 < 0，一文字 > である．これらは，中間バッファに蓄えられると同時に，データ出現頻度の計測に用いられる．

初めに，ハフマン木が作成される．ハフマン木は，データ列に出現する記号（GZIP においては文字か一致長か戻り距離）の出現頻度をもとに作成される．まず，各記号を出現頻度の高い順に，また同じ出現頻度では辞書の降り順となるように左から右へと並べ節点とする．その中から，出現頻度が最小の節点と 2 番目に小さい節点を葉として，新たな節点を設ける．この際，新たな節点には上記二つの記号の出現頻度を足し合わせた値を与える．以上を繰り返して根節点まで到達することで木が完成される．次に，根から順に左右に 0 と 1 の値を割り振っていく．GZIP では左側が 0 となる．すると，それぞれの葉に対して一意にビット列が与えられる．このビット列をハフマン符号とし，記号とハフマン符号を対応させた符号表を作成する．そして，もとのデータ列をハフマン符号に変換していくことでハフマン符号化プロセスが完了する．

例えば，文字列”westtitle”をハフマン符号化する場合，アルファベットの出現頻度からハフマン木は図 3.24 のように作成され，各アルファベットのハフマン符号は表 3.12 となる．そして，最終的に元の文字列”westtitle”は，”111 01 110 00 00 100 00 101 01”と符号化される．

ハフマン符号の復号においては，符号を前方から検索していき，最初に完全一致したハフマン符号が変換対象となる．更に，ハフマン符号は，全ハフマン符号の長さのみを知ることができれば，一意に符号表を復元することが可能であるという特徴的な性質を持つ．”westtitle”の例で，ハフマン符号の長さが”2 2 3 3 3 3”であるとわかれば，一意に”00 01 100 101 110 111”へと変換できるということである．

GZIP における LZSS 符号では，一致長の最大値は 258Byte，戻り距離の最大値は 32KB である．これをそのままハフマン符号化すると，符号表サイズが大きくなりすぎ，また現れない一致長や戻り距離のハフマン符号が無駄になる．そこで，GZIP では一致長を 29 種の 5bit コードで，戻り距離を 30 種の 5bit コードで圧縮し，後続 bit によって詳細を示すことで，ハフマン符号の作成を効率化している．一致長と戻り距離の値による範囲を表 3.13 および表 3.14 に示す．例えば，一致

表 3.12: 出現頻度とハフマン符号

文字	出現頻度	ハフマン符号
t	3/9	00
e	2/9	01
i	1/9	100
l	1/9	101
s	1/9	110
w	1/9	111

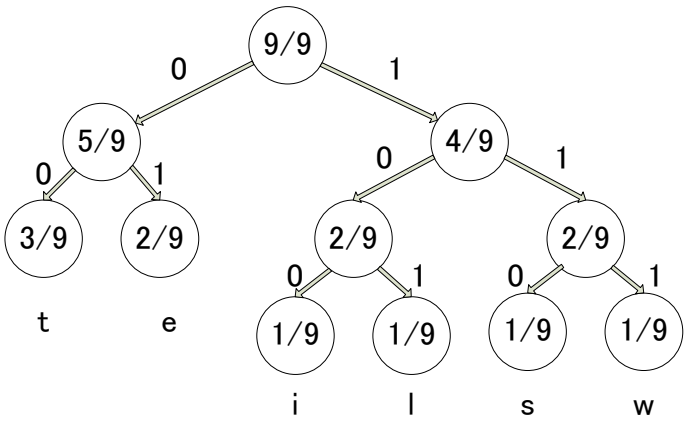


図 3.24: ハフマン木の例

長が 20 ならば，一致長コード 13 と後続 bit を用いて”01101 01”と表せる．

GZIP では，ASCII コード 256 種と LZSS 符号の一致長コード 29 種の値，それに終端文字を表す 1 種の値を合わせた 286 種の値（9bit）が一つのハフマン木で符号化される．これにより一致不一致を示すモードフラグを考慮せずとも，復号後の値が 0 から 255 なら単純な文字コードであり，256 から 287 なら LZSS 符号の一致長コードであることが検知できる．また，LZSS 符号の戻り距離コード 30 種の値（5bit）は別のハフマン木で符号化する．ここで，GZIP におけるハフマン符号の最大値は 15bit と決められている．

GZIP 圧縮処理 3: ハフマン符号における符号表の作成

ハフマン符号化後のデータを出力する際，復号のために符号表を添付する必要がある．DEFLATE ブロックでは，符号表の添付サイズを削減するために，複雑なフォーマットに基いて符号表が添付される．ここまで，GZIP におけるハフマン符号の符号表は，文字/一致長の符号表と戻り距離の符号表の 2 種類があることを前述した．これらの符号表は，実際には記号とハフマン符号の対応表ではなく，記号とハフマン符号長の対応表とすることでデータサイズが削減されている．DEFLATE ブロックでは，更にこの符号長 1 から 15（ハフマン符号の最大長は 15bit）をハフマン符号化することで符号表サイズの削減を計っている．従って，DEFLATE ブロックには，符号長の符号表と文字/一致長の符号表，戻り距離の符号表という 3 種類の符号表が添付される．図 3.25 に DEFLATE ブロックにおける符号表の構造を示す．

符号長の符号表は，1 から 15 の符号長と 4 つの繰り返し記号を合わせた 19 種類のコードをハフマン符号化し，そのハフマン符号長を 3bit としてコード順に添付することで，3bit×19 により 57bit が必要となる．なお，使われていない符号長や繰り返し記号を省くことで，57bit 以下とすることが可能である．次に，文字/一致長の符号表は，符号長をハフマン符号化により平均 3bit 程度まで圧縮できるため，これを文字/一致長コードの順に添付することで，3bit×286 により 850bit 程度必要となる．戻り距離の符号表も同様に，平均 3bit となった符号長を戻り距離コード順に添付することで，3bit×30 により 90bit 程度必要となる．更に，DEFLATE ブロックでは，符号表の先頭に 5bit の HLIT および HDIST，4bit の HLEN が付加される．HLIT および HDIST は，一致長コードの数，戻り距離コードの数-1 を示すもので，GZIP ではどちらも 29 となることが一般的で

表 3.13: 一致長コードと一致長範囲の対応

コード	後続 bit	範囲	コード	後続 bit	範囲	コード	後続 bit	範囲
0	0	3	10	1	15-16	20	4	67-82
1	0	4	11	1	17-18	21	4	82-98
2	0	5	12	2	19-22	22	4	99-114
3	0	6	13	2	23-26	23	4	115-130
4	0	7	14	2	27-30	24	5	131-162
5	0	8	15	2	31-34	25	5	163-194
6	0	9	16	3	35-42	26	5	195-226
7	0	10	17	3	43-50	27	5	227-257
8	1	11-12	18	3	51-58	28	0	258
9	1	13-14	19	3	59-66			

表 3.14: 戻り距離コードと戻り距離範囲の対応

コード	後続 bit	範囲	コード	後続 bit	範囲	コード	後続 bit	範囲
0	0	1	10	4	33-48	20	9	1025-1.5K
1	0	2	11	4	49-64	21	9	1537-2K
2	0	3	12	5	65-96	22	10	2049-3K
3	0	4	13	5	97-128	23	10	3073-4K
4	1	5-6	14	6	129-192	24	11	4097-6K
5	1	7-8	15	6	193-256	25	11	6145-8K
6	2	9-12	16	7	257-384	26	12	8193-12K
7	2	13-16	17	7	385-512	27	12	12289-16K
8	3	17-24	18	8	513-768	28	13	16385-24K
9	3	25-32	19	8	769-1K	29	13	24577-32K

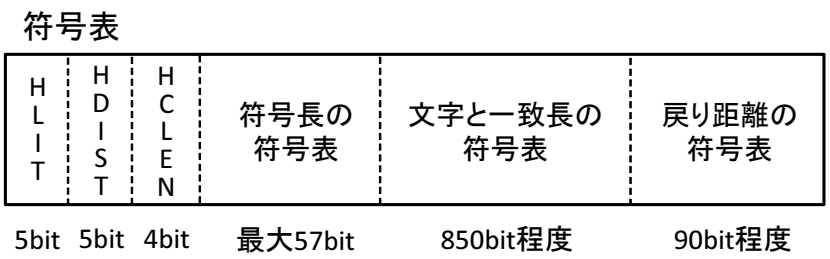


図 3.25: DEFLATE ブロックにおける符号表の構造

表 3.15: FLG フィールドの名称と役割

ビット数	名称	役割
bit0	FTEXT	圧縮データが ASCII テキストであるかどうか
bit1	FHCRC	圧縮データの直前に CRC 値が挿入されているかどうか
bit2	FEXTRA	オプションの拡張フィールドを使用する
bit3	FNAME	元ファイルの名前が記載されているかどうか
bit4	FCOMMENT	コメントが存在するかどうか
bit5-7	予約	予約ビット

ある．また，HLEN はハフマン符号として使われている符号長の個数を示す．ハフマン符号が 1bit から 15bit まで全て存在する場合には，HLEN は 15 となる．これらを全て併せると，DEFLATE ブロックの符号表サイズはおよそ 120Byte 程度となる．

符号表の添付後，図 3.22 に示したように，更に先頭に 1bit の BFINAL および 2bit の BTYPE を付与することで DEFLATE ブロックが完成する．BFINAL は，そのブロックが最終ブロックであることを表す 1bit である．また，BTYPE はハフマン圧縮の方式を表す．BTYPE が 00 ならば無圧縮データであることを，01 ならば固定ハフマン方式により，10 ならばカスタムハフマン方式により圧縮されていることを示している．

GZIP 圧縮処理 4: GZIP ヘッダの添付

GZIP 圧縮データには GZIP ヘッダやフッタが付加される．GZIP ファイルフォーマットの詳細を図 3.26 に示す．まず，GZIP 圧縮データの先頭には 10Byte のヘッダが付与される．その後に 10Byte の拡張ビットが存在することもある．ヘッダには GZIP 圧縮の有無やデータ圧縮時の環境などを記録する．最初の ID1 および ID2 がそれぞれ 0x1f，0x8b であるならば，そのデータは GZIP 圧縮データであることを示す．次の CM は GZIP の圧縮方法を規定する．CM = 0-7 は予約，CM = 8 は ”deflate” 圧縮方式であることを表す．通常，GZIP は deflate 圧縮方式が用いられる．FLG は GZIP 圧縮に適用されている様々なオプションを示す．表 3.15 に FLG の各ビットの役割を示す．これらの FLG の後，次の 4 バイトは MTIME と呼ばれ，その GZIP ファイルの最終更新日が記録される．9 番目のバイト（XFL）は DEFLATE 方式における圧縮の方法を示す．XFL=2 のときは圧縮速度が遅い代わりに圧縮率の高いアルゴリズムを，XFL=4 のときは反対に速いアルゴリズムを用いたことを示す．最後のバイトは GZIP 圧縮を行ったオペレーティングシステムを表す．

これらの GZIP ヘッダの後に圧縮されたブロックが続く．圧縮されたデータブロックの後には，データの最後を表す 8 バイトの GZIP フッタが存在する．はじめの 4 バイトには圧縮されたデータの CRC32 の値が，残りの 4 バイトには非圧縮時のオリジナルのデータ長のモジュロ 2^{32} の値が入る．

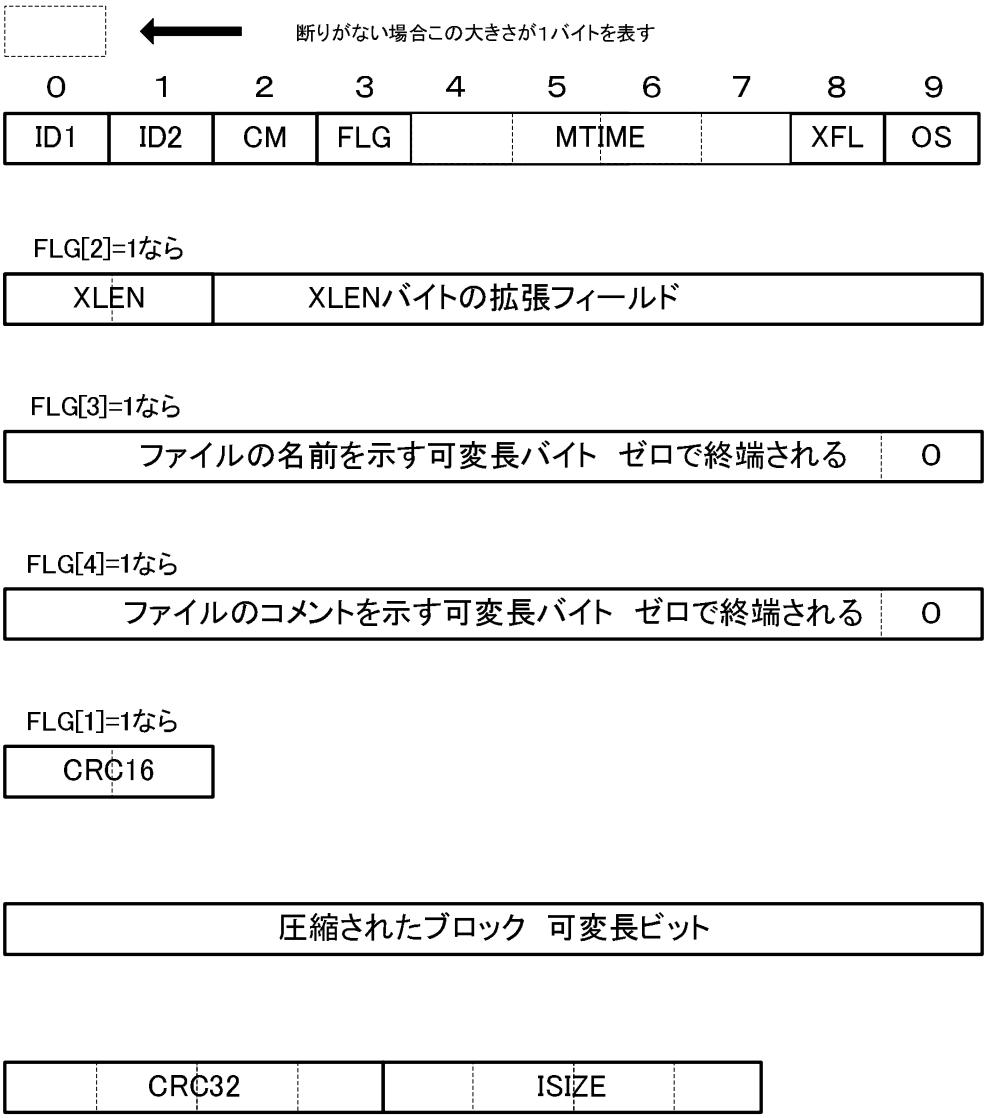


図 3.26: GZIP のフォーマット

3.2.2.2 GZIP 展開に関する既存研究

従来，データの GZIP 圧縮展開に用いられる手段は，GZIP の作者らによって提供された zlib ライブラリを用いることが一般的であった [85][94]．GZIP の用途は，ストレージ容量の圧迫に対するデータサイズの削減が主であり，処理に速度が求められることは少なかった．これに対し，近年では前述したように，主に通信分野においてデータ転送量を削減するために GZIP が用いられる．このような用途では，データの圧縮展開はデータをネットワークに転送する前後で即座に行われる必要があるため，GZIP 圧縮展開の処理速度が重要な要素となる．そこで，近年は GZIP 圧縮展開速度の向上に関する研究が行われてきた．以降では，主だった研究について説明する．

既存研究 1: ソフトウェアライブラリ zlib

GZIP を圧縮展開するソフトウェアライブラリとして zlib がある．zlib は GZIP の作者である Mark Adler と Jean-loup Gailly によって作られた，データの圧縮および伸張を行うためのフリーのライブラリである．多くのプログラミング言語では zlib をライブラリで提供しており，例えば Java プラットフォームからも利用できる．また，zlib は商用ソフトを含む多くのソフトウェアで採用されている．画像フォーマットの PNG が Deflate の実装を必要とするため，データ圧縮用途だけでなく，画像を表示するソフトウェアでも一般的に使われている．パソコン・サーバー・携帯電話など，非常に多くの OS で使われているライブラリのため，2002 年と 2005 年にセキュリティ問題が発見された際，広範囲のシステムに影響が及んだ．

zlib のインターフェイスは数種類の単純な関数呼び出しに制限されている．全体の圧縮展開セッションにおける状態は C 構造の z_stream というタイプによってカプセル化されている．圧縮と展開を行う際，zlib はそれ自体 I/O としては働かない．そのかわり，z_stream オブジェクトに存在する入力バッファポインタよりデータを読み出している．next_in に入力データの次のブロックへのポインタをセットし，avail_in に処理可能なバイト数ををセットする. 同様に，zlib は出力データを next_out にセットしたメモリバッファに書き込む. 出力バイトを書き込む際に，zlib は count が 0 になるまで avail_out からデータを読み出す．上述した I/O 手法を用いることで，zlib に独立した読み書きのコードを実装する手間を省き，入力ストリームの種類によらず対応できる．

このような zlib ライブラリを用いた GZIP 圧縮展開について，4Gamer.net では，Sandy Bridge プロセッサにおける zlib のスコアを測定している [8]．図 3.27 に，Web ページ [8] で公表されている zlib のスコアを示す．この研究による評価では，最新の Sandy Bridge プロセッサを用いることで，最も速いもので 250MB/s が得られている．従来のプロセッサに比べ高速であるとはいえ，アプリケーションルータにおける 100Gbps ネットワーク処理に要する GZIP 展開スループット 5.5Gbps には不足している．

既存研究 2: 複数 FPGA を用いた GZIP 圧縮展開ハードウェア

インターネットデータセンター (IDC) 向けに FPGA を用いて低コストに GZIP 圧縮展開を行うハードウェアアーキテクチャを提案する論文 [9] が存在する．データ圧縮技術はインターネットやデータベースストレージの領域において幅広く使われており，その中でも GZIP は最も広く用いられる．従来のアプリケーションでは，データの圧縮展開にソフトウェアベースの手法を用いることが一般的であった．しかしながらソフトウェアベースのデータ圧縮展開は，大量の CPU 資

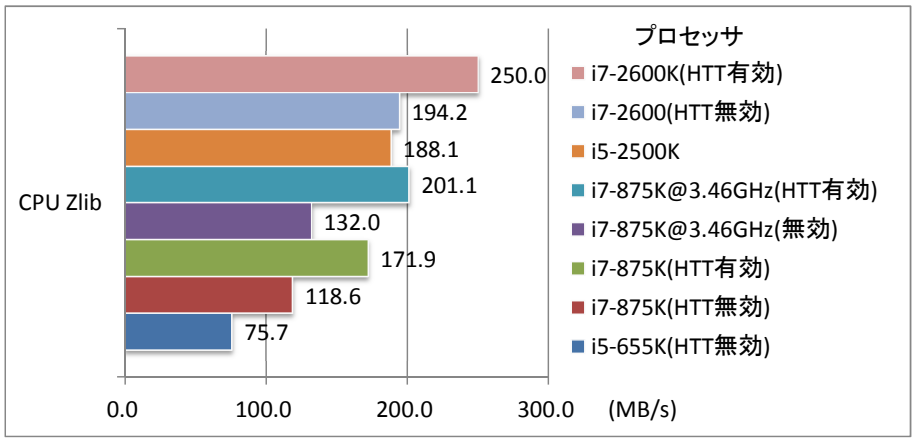


図 3.27: 最新のプロセッサにおける zlib のスコア [8]

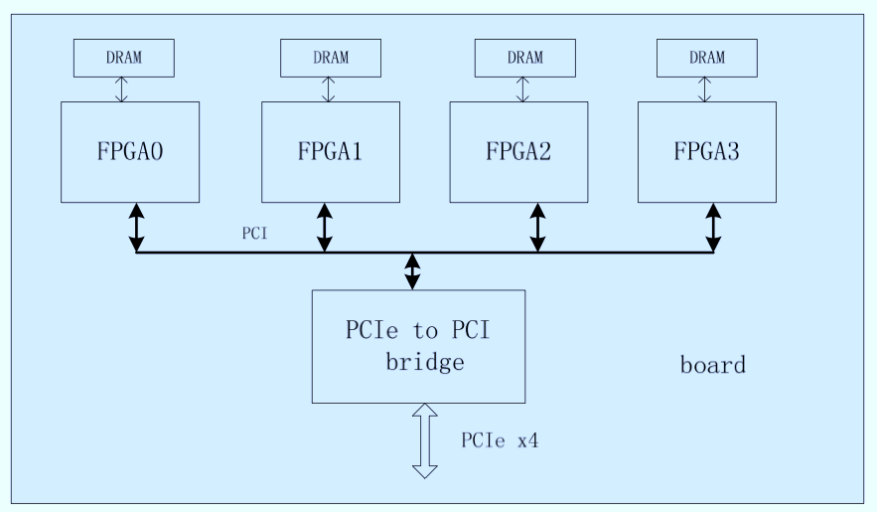


図 3.28: FPGA を４つ用いたアーキテクチャ[9]

源とメモリ資源を消費し，達成できるスループットもそれに応じて限定的なものとなってしまう．

IDC は膨大な量のデータを扱う必要から，一つのコンピュータが複数のタスクを同時に処理することがあり負荷が大きい．そこで，この研究では FPGA チップを４つ用いてタスクレベルでの並列性を達成し，CPU の負荷をオフロードするとともにシステムが処理できる同時トランザクションの数を増やしている．図 3.28 のように 4 つの cyclone-III チップを PCI を介してブリッジ接続し，ホストコンピュータと PCI-eX4 インターフェイスによって相互接続している．それぞれの FPGA チップは圧縮エンジンとしても展開エンジンとしても働く．

この研究では，FPGA の 16 個の分散 RAM を用いることで，履歴の保存管理およびデータの読み出しを並列化している．これは，ソフトウェアがシーケンシャルにしかデータを読み書きでき

表 3.16: ソフトウェアとハードウェアの展開に関する比較

Compression Ratio(%)	CPU Bandwidth (MB/s)	FPGA Bandwidth (MB/s)
21.17	241.1	306.1
23.67	218.3	300.0
27.94	213.8	286.8
34.48	164.3	254.8
34.67	160.7	254.5
35.22	224.7	266.3

表 3.17: PowerEN における GZIP 圧縮展開のパフォーマンス

Algorithm	# of Engines	Bandwidth (Typical)	Bandwidth (Peak)
gzip (input bandwidth)	1	8 Gbps	9.2 Gbps
gunzip (output bandwidth)	1	8 Gbps	16 Gbps

ないことに対し，効果的なデータの読み書きを可能とする．また，FPGA チップを 4 つ用いることで，提案されているハードウェアアーキテクチャでは 4 つの探索処理を同時に行うことができる．

この研究では 2.66GHz で動作する CPU コアと，132MHz で動作する FPGA アクセラレータとのスループットの比較結果を示している．表 3.16 は圧縮に関する CPU と FPGA のスループットの比較である．どちらも FPGA の方が圧縮率，バンド幅ともに上回っているが，この研究では回路規模や CPU の詳細スペックが表示されていない．この比較では不十分であり，圧縮率やバンド幅が向上していても消費電力やコストなどが非効率であれば実用性は低い．また，この手法における GZIP 圧縮展開スループットは 2Gbps 程度であり，アプリケーションルータに搭載する上では FPGA への実装は適していない．

既存研究 3: ハードウェアアクセラレータ PowerEn

ネットワークの成熟に従って，より多くの機能がネットワークそれ自体に埋め込まれることが多くなってきており，ネットワークのエッジに相当するアプリケーションはますます多くの負荷に対応する必要がある．例えば，サイバーセキュリティやネットワーク侵入検知システム，データベース高速化，オンライントランザクション処理など，サービウ思考アーキテクチャの高速化が求められている．

このような要求に対し，パケットの暗号化，XML 解析，文字列探索，GZIP 圧縮展開といった様々な処理をネットワーク上で高速に行うコプロセッサとして，IBM の PowerEN[10] がある．PowerEN におけるハードウェアアクセラレーションの概要を図 3.29 に示した．PowerEn の半導体テクノロジは，IBM の 45nm SOI(Silicon on Insulator) を用いた SoC(System on a Chip) である．表 3.17 に PowerEN における GZIP 圧縮展開ハードウェアアクセラレータのパフォーマンスを示した．

PowerEN は次世代のネットワーク要求を満たすように設計されたプロセッサであり，16 の汎用

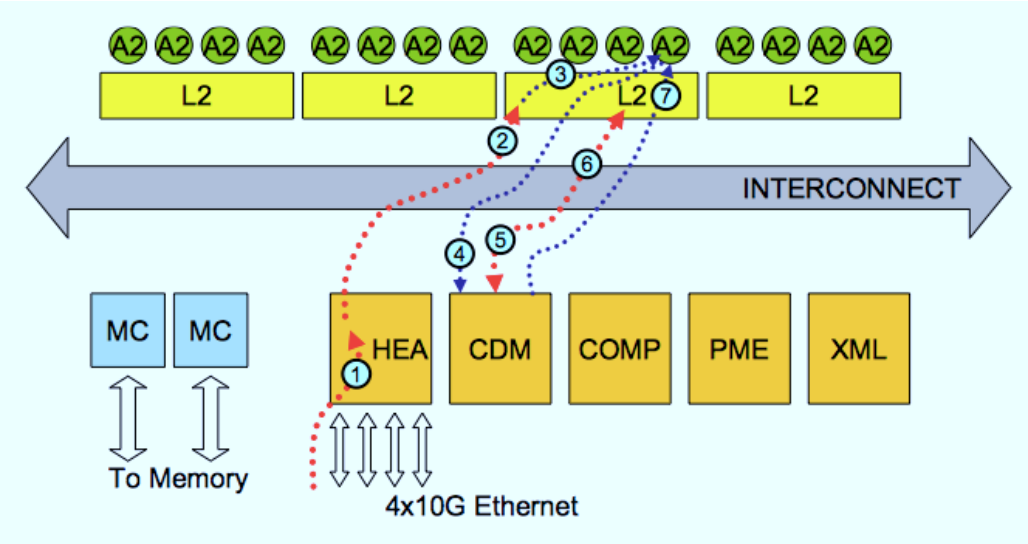


図 3.29: PowerEN におけるハードウェアアクセラレータ [10]

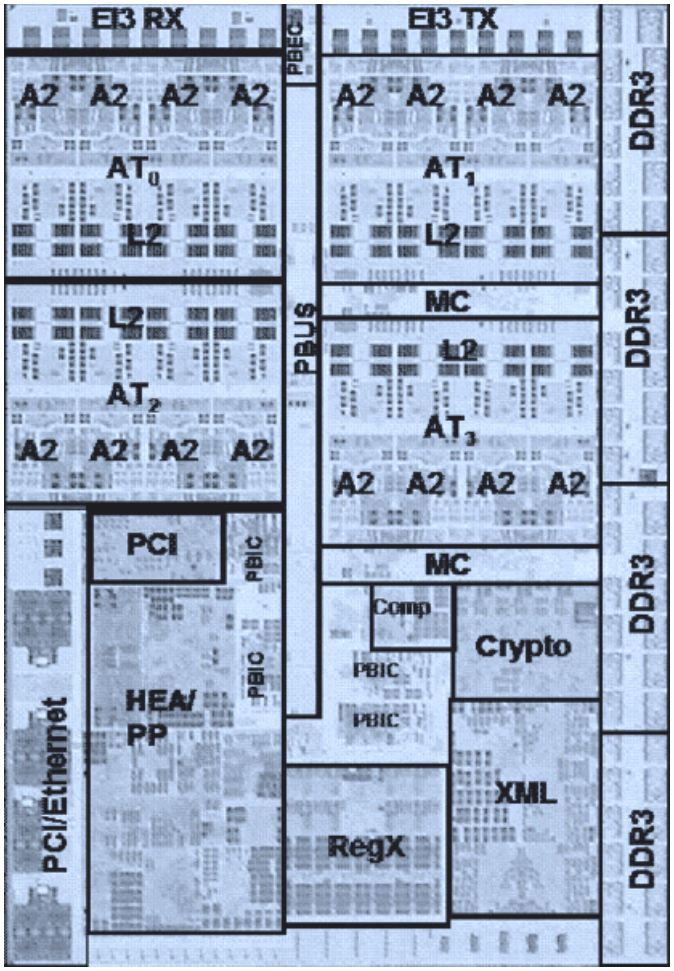


図 3.30: PowerEN 試作チップの概要 [11]

マルチスレッドプロセッサコアを備えている．このアーキテクチャはアクセラレータフレンドリーであり，ソフトウェアが簡単に効率よくハードウェアアクセラレータを使用できるような機能を備えている．本項では，論文 [10, 11] に公表されている範囲で GZIP 圧縮展開部分のハードウェアアクセラレータについて紹介し，実用性を検討する．

論文 [10] では，PowerEn の GZIP 展開スループットが 8Gbps を達成可能であることをシミュレーションにより示しているが，これは一つの GZIP ファイルをそのまま展開した際の結果から求められたスループットであり，ネットワーク上でのパケット分割を考慮した結果ではない．そのため，ネットワーク経路上での GZIP 展開においてはスループットがこの値より大きく低下するはずである．また，論文 [10] における記述とアーキテクチャ図を見る限りでは，PowerEn はコンテクストスイッチ手法のようなパケット分割に対応した機構を持たないと考えられる．従って，ネットワーク経路上での GZIP 展開においては，パケット分割を考慮した手法を実装する必要があると考えられる．

また，PowerEN はパケットの情報を 4KB 分キャッシュするヒストリーキャッシュという仕組みを持つ．これは，LZSS 符号に用いられる 32KB の辞書のうち，頻繁に参照される 4KB をキャッシュすることで LZSS 符号の展開を高速化する機構だと考えられるが，詳しいアーキテクチャや，GZIP 展開処理スループットへの貢献度合いが具体的に示されておらず，本機構が有用であるという根拠は示されていない．

PowerEn における最大の問題点は高いハードウェアコストである．図 3.30 に PowerEn のチップを示した．この図から，全体のダイサイズとその一部である GZIP 圧縮展開部分の面積を比較することにより，GZIP 圧縮展開部分に関するダイサイズを大まかに見積もることができる．PowernEn 全体のダイサイズが $428mm^2$ であることから，GZIP 圧縮展開部のダイサイズはおおよそ $4.15mm^2$ 程度だと考えられる．更に，後述する既存研究 4 の論文 [12] における GZIP の圧縮モジュールと展開モジュールの回路規模を比較すると，おおよそ 9:13 であることから，PowerEn における展開モジュールのダイサイズは $2.45mm^2$ 程度であると見積もられる．このダイサイズは一般的な使用においては比較的大きい回路規模であり，コストパフォーマンス的に不十分である．

既存研究 4: ハフマン符号化処理の高速化手法

GZIP 圧縮展開処理全体の高速化ではなく，ボトルネックに焦点を当てた研究もある [12]．Akil らは，GZIP 展開におけるハフマン符号の符号表作成および復号が性能ボトルネックとなると考え，これらの処理の高速化およびメモリ使用量の削減を検討している．展開機構は高速化のためにハードウェアにより実装されており，高速化が必要であるという点で本研究と類似している．論文 [12] では三つの手法が提案されているが，本論文ではその中で最もパフォーマンスの高いパラレルデコーディングを紹介する．

この方法の最も重要な点はハフマン符号の長さを決定し，その後，文字/一致長コードのアドレスと距離のアドレスを決定しているところである．この手法では，復号する符号が実際には何 bit なのか計算を先に行うことで，テーブルから読み出すアドレスを一つに絞っている．パラレルデコーティングのアーキテクチャを図 3.31 に示す．

ハフマン符号の bit 数を決定してから文字を読み出す方法を以下に示す．この方法は論文 [95] に詳しく説明されている．まず minimum codes と呼ばれるテーブルを参照し，復号しようとしてい

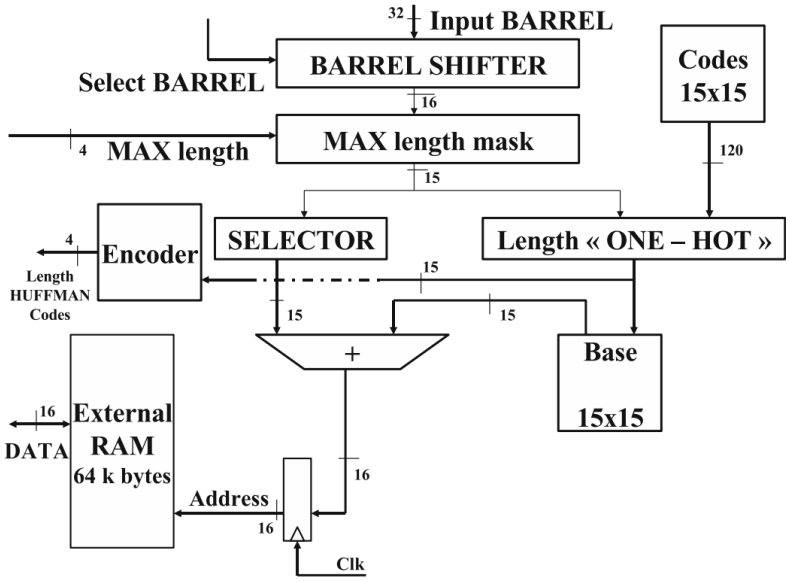


図 3.31: パラレルデコーティングのアーキテクチャ[12]

る符号が何 bit なのかを計算している．minimum codes は与えられた bit（1bit〜15bit）に対する最小ハフマン符号を示すテーブルである．ハフマン符号は 1bit から 15bit の長さしかとり得ないため，復号される bit 列と最小ハフマン符号を 15 個並列に比較することで，1cycle により符号長を決定できる．minimum codes は，図 3.31 に Codes として示されており，符号長 15 個 × 最小ハフマン符号 15bit を要する．

次に，得られた符号長から，対応する文字/一致長コード，もしくは距離のアドレスを決定する．事前に，各符号長に対して，

$$V_a = (Address_{code_{Min}}) - Code_{Min} \tag{3.3}$$

を計算し，V_a を”Base Memory”と呼ばれるテーブルに挿入してある．ここで，Address_Code_Min は，その符号長における最小のハフマン符号が格納されたアドレスであり，Cod_Min は最小のハフマン符号を表す．Chd を復号したいハフマン符号とすると，このハフマン符号の表す文字を参照するアドレス e は

$$e = C_{hd} + V_a \tag{3.4}$$

により計算される．Base Memory は，図 3.31 に Base と示されているブロックであり，15 個の符号長 ×15bit を要する．

この方法では，一つのハフマン符号の復号が 2 サイクルで済み，また一定の速度で文字が出力されるため安定したパフォーマンスで処理することが可能である．パラレルデコーディングの手順をまとめると以下となる．

1. 復号する bit 列のハフマン符号長の計算

2. 文字/一致長コード，もしくは距離を格納してある外部 RAM のアドレスの決定

3. 決定したアドレスの値の読みだし

この論文では圧縮機構と展開機構に分けてアーキテクチャの評価が行われている．ハードウェアとして Virtex XCV4000 が搭載された ADM-XRC ボードを用いている．圧縮機構は 450CLB 使用し，最大動作周波数 50MHz での動作を可能としている，また展開機構については 650CLB，外部メモリとして 64KB のメモリを使用し，最大動作周波数 50MHz での動作を可能としている．ソフトウェアベースの従来手法とパラレルデコーディング手法において展開処理の性能比較を行った結果，この手法により 2，3 倍高速化されるとしているが，詳細な結果は述べられていない．

パラレルデコーディングはメモリ使用量，復号における速さ共にアプリケーションルータに用いる GZIP 展開機構として高い親和性があると考えられる．しかし上記に挙げた手法をそのまま用いることはできない．まず，画像印刷の GZIP 展開においてはハフマン符号表の作成がボトルネックになると述べているが，ネットワーク上における GZIP 圧縮データの展開においては，同様に復号処理自体や前述したコンテクストスイッチ手法におけるコンテクストの読み出しがボトルネックとなることが考えられる．また，論文 [12] によると，復号を 2 クロックで完了しメモリとして 64KB 用いると記述されているが，これらに関しては改善の余地があり，更に，ネットワーク上での処理を想定する上ではマルチストリームへの対応を考慮しなければならない．そこで，我々はパラレルデコーディングを改良することで，よりハフマン符号表の作成を高速化した．本手法については第 5 章で後述する．

既存研究 5: ネットワーク経路上におけるパケットの逐次的な GZIP 展開手法

ここまで，一般的な GZIP ファイルの展開を目的とした既存研究について紹介してきた．一方で，ネットワーク経路上におけるパケットの GZIP 展開は，パケット断片化の影響を大きく受けるため，GZIP ファイルの展開とは様相が異なる．GZIP 展開では，3.2.1 で述べたように，ハフマン符号を復号するために符号表が，LZSS 符号を復号するために辞書がそれぞれ必要となる．ここで，符号表は GZIP ヘッダから，辞書は復号対象の直前 32KB のデータから作成される．ネットワークにおいて GZIP 圧縮データは，図 3.32 に示したように，パケット断片化によって分割されることがある．そのため，符号表および辞書を分割された以降のパケットにも引き継がなければならない．

従来の GZIP 展開手法では，ストリームの全てのパケットが到着してから GZIP 展開を開始することで，上記の問題を解決していた．しかしながら，このような手法は 3.2.1 項でも述べたように，ストリームの全パケットを保存しなければならず，広帯域ネットワークにおける処理には不向きである．そこで，石田らはアプリケーションルータの GZIP 展開機構において，zlib およびコンテクストスイッチを用いることでパケットを逐次的に処理する手法を提案している [2].

石田らは，3.2.1 項で紹介したコンテクストスイッチ手法を拡張することで，GZIP 圧縮パケットの逐次展開を可能とした．石田らの提案するコンテクストスイッチの拡張手法を図 3.33 に示す．拡張されたコンテクストスイッチでは，GZIP 展開に必要となる符号表と辞書，パケット末尾で復号できなかったデータが従来のコンテクスト情報に加えられる．そして，パケットの GZIP 展開時にこれらのコンテクスト情報を読み出すことで，ストリームの途中のパケットであっても GZIP

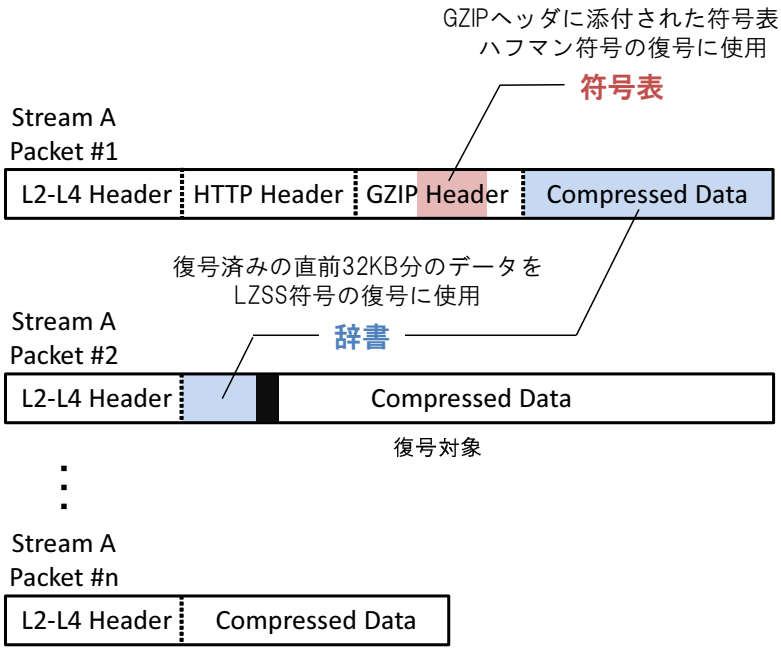


図 3.32: パケット分割された GZIP データ

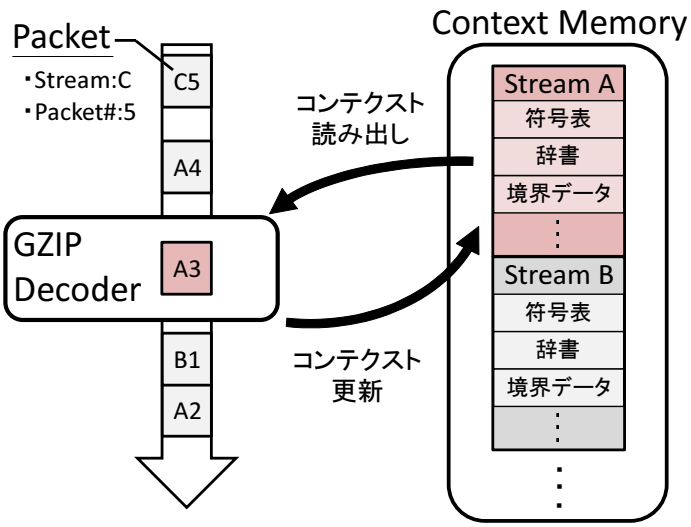


図 3.33: コンテクストスイッチの GZIP 展開への拡張

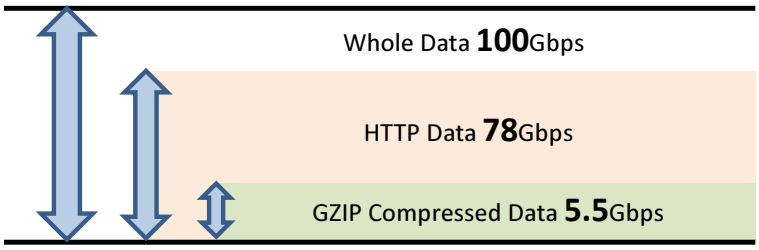


図 3.34: トラフィックにおける HTTP および GZIP 圧縮トラフィックのデータ割合

展開が再開可能となる．パケットの GZIP 展開後は，更新された辞書およびパケット末尾のデータをコンテクストメモリに書き込むことで，コンテクスト情報を更新する．

石田らのシミュレーションによると，40Gbps の仮想トラフィックに対して情報抽出を行う場合に必要となるコンテクストメモリサイズは，300 秒のタイムアウトを設定した場合に 5GB 以下となる．石田らは GZIP 展開に zlib を用いており，zlib の仕様により 1 ストリームあたり 64KB のメモリスペースをコンテクスト情報として確保しているが，実際に必要なメモリ量は符号表数百 Byte と辞書 32KB の合わせて 33KB 程度である．これを考慮して，100Gbps を想定したコンテクストメモリサイズを予測すると 6.4GB 程度となり，アプリケーションルータであっても十分に実装できるサイズであることがわかる．一方で，スループットに関しては情報抽出機構全体で 1.12Gbps 程度が達成可能であると述べられている．論文 [2] では GZIP 展開と文字列探索のどちらがボトルネックとなっているか分析はなされていないが，既存研究の zlib によるスループットを考慮すると，やはり zlib ではスループットが不十分であると考えられる．

3.2.2.3 アプリケーションルータにおける GZIP 展開機構

GZIP 圧縮された HTTP トラフィックに対する展開機構は，アプリケーションルータに関する既存研究の多くでは考慮されてこなかった．しかしながら，近年の GZIP 圧縮データの普及を考えると搭載すべき機能であると言える．実際に，慶應義塾大学理工学部西研究室の 1Gbps ネットワークにおける，日常的なネットワーク使用状況下にあった 2011 年 12 月 5 日から 8 日間のトラフィックを解析し，ネットワーク中における HTTP データ量の割合及び GZIP 圧縮された HTTP データ量の割合を測定したところ，図 3.34 に示すように，トラフィック全体の 5.5%程度が GZIP 圧縮トラフィックであることがわかった．これは，100Gbps での情報抽出を想定した場合，GZIP 展開機構に 5.5Gbps 以上のスループットが求められるということである．前述したように，石田らによると，最も一般的な GZIP 展開手法である zlib を用いた GZIP 展開機構を搭載したアプリケーションルータでは，1Gbps 程度が限界であると述べられている．従って，GZIP 展開処理が一つのボトルネックになっていると言える．アプリケーションルータにおいては，今後の GZIP エンコードの普及も予想し，より高スループットを達成する GZIP 展開機構が必要となる．

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS

(msg:"POLICY Google Desktop initial install - installer request";
flow:to_server, established;
uricontent:"/installer?"; uricontent:"action=install"; uricontent:"version=";
uricontent:"id="; uricontent:"brand=GGLD"; uricontent:"hl=";
content:"User-Agent|3A|"; nocase; content:"Google"; distance:0; nocase;
content:"Desktop"; distance:0; nocase;
pcre:"/User-Agent¥x3A[^¥n¥r]+Google[^¥n¥r]+Desktop/smi";
classtype:policy-violation; sid:7859; rev:1;)
```

図 3.35: Snort の抽出ルール例

3.2.3 文字列探索機構

パケットのペイロードは文字列探索機構へと転送され，抽出ルールにもとづいて情報が抽出される．この際，ペイロードに対して文字列探索が行われる．抽出ルールはアプリケーションルータが提供するサービスに依存して決定される．

図 3.35 に，例として NIDS アプリケーション Snort の抽出ルールを示す．Snort ルールの構成要素は大きく分けて先頭行のルールヘッダとそれに続くルールオプションの 2 つである．ルールヘッダでは送信元，宛先のアドレスおよびポート番号やプロトコルといった探索パケットの指定が行われる．図 3.35 の例では，内部ネットワークから外部ネットワークに向けた宛先ポート 80 番の TCP パケットが対象となる．ルールオプションでは，実際にペイロードから抽出すべき文字列が指定される．この文字列をパターンと呼ぶ．ルールによっては，一つのルール内に複数のパターンが指定されている場合もある．

アプリケーションルータのサービス例は 2.2.2 節で述べたが，例えば NIDS ならば図 3.35 に示したようなシグネチャ数千程度が抽出ルールとして考えられる．また，web レコメンデーションにおいては＜ title ＞といった HTML タグやコンテンツを特定するためのキーワードが，QoS 保証においてはアプリケーションの特徴が抽出ルールになると考えられる．

3.2.3.1 文字列探索処理の概要

文字列探索の方法は様々であるが，まず基本となる単純な文字列探索方法について説明する．図 3.36 に力任せ法と呼ばれる最も単純な文字列探索の概要を示す．まず入力データの先頭に対し，抽出文字列との完全一致検索を行う．ここで一致しなかった場合には比較対象となる入力データを 1Byte ずらし，再度完全一致検索を行う．この操作を繰り返すことで，入力データ全体に対する抽出文字列の一致を検索する．実装方法によって，マッチングがあった場合に探索処理を終了し最初のマッチング位置を返り値とするか，入力データの最後まで探索を行い複数の一致があった場合に複数のマッチング位置を返り値とするかは異なる．

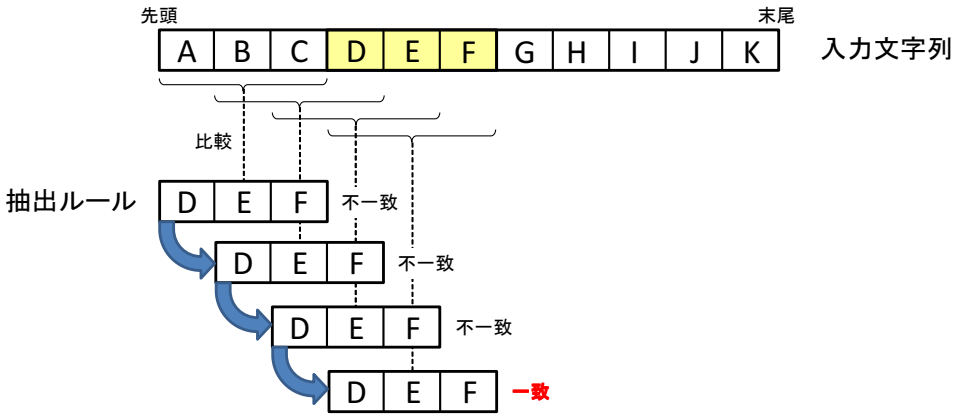


図 3.36: 単純な文字列探索処理の概要

3.2.3.2 文字列探索処理に関する既存研究

力任せ法は 1Byte ずつ入力データをシフトさせながら一致を検索するため，スループットが低い．文字列探索処理は従来，多くのシステムにおいて性能ボトルネックとなっており，処理高速化に関する研究が多くなされてきた．特に，近年の Snort[80] や Bro[96] の普及により，文字列探索エンジンの研究は活発化している．文字列探索の既存研究におけるアプローチは，主に 3 種類に分類される．以降では，3 種類のアプローチの概要とその既存研究について紹介する．

既存研究 1: Boyer-Moore 法

力任せ法による 1 文字ずつのシフトは効率が悪く，文字列探索処理のスループットが獲得できない要因の一つとなっている．そこで，Boyer-Moore 法は，完全一致検索処理を工夫することにより，より多くの文字数のシフトを可能とする [97].

まず，パターンの文字長を m とし，入力文字列の先頭から m 文字とパターンを照合することから始める．Boyer-Moore 法では，入力文字列とパターンの照合を後方から行う．従って，まず入力文字列の m 文字目とパターンの m 文字目が比較される．ここで，文字が一致した場合は，入力文字とパターンそれぞれの m-1 文字目を更に照合していく．この操作によって，パターンの先頭文字と入力文字が一致した場合に，完全一致が検出できる．一方で，文字が一致しなかった場合は，次に入力文字列 m 番目の文字がパターン内に存在しないか検査される．文字がパターン内に存在した場合は，一致したパターンの文字位置と入力の m 文字目を重ね合わせ，再度後方から照合を開始する．文字がパターン内に存在しなかった場合は，パターンの先頭と入力の m+1 文字目を重ね合わせ，再度後方から照合を開始する．このようにして，入力文字列の末尾まで照合したところで文字列探索処理が完了となる．

Boyer-Moore 法の例を図 3.37 に示す．図では，入力文字列”aliceinwonderland”に対してパターン”wonder”を探索している．まず，入力文字列”alicei”とパターン”wonder”の照合を後方から行っている．i と r は一致しないため，次に，入力文字 i がパターン中に存在しないか探索する．ここで，wonder に i は含まれないため，入力文字列をパターン長である 6 文字分シフトする．そして，

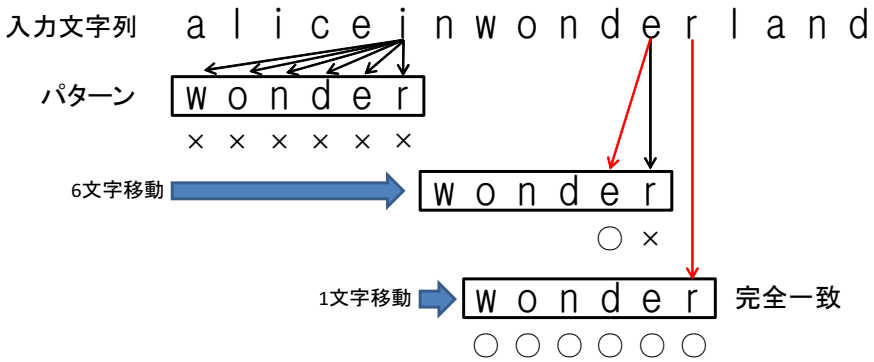


図 3.37: Boyer-Moore 法における一致探索例

シフト後の文字列”nwonde”とパターン”wonder”の照合を後方から行う．ここでも末尾の文字は一致しないが，入力文字の末尾 e はパターン中にも存在する．そこで，入力文字の e とパターンの e の位置を合わせ，すなわち入力文字列を 1 文字シフトさせ，再度後方から照合を行う．この結果，入力文字”wonder”とパターン”wonder”の一致が検出される．

このように，Boyer-Moore 法では後方から照合を開始することで，無駄な完全一致検索処理を省略し，文字列探索を高速化する．その平均計算量は $O(n/m)$ であり，平均計算量 $O(n)$ の力任せ法よりも基本的には高速である．Boyer-Moore 法は実用的であり，多くのシステムで採用され，また最適化が提案されてきた．Boyer-Moore 法の改良手法として性能が良く，有名なアルゴリズムに Sunday アルゴリズムがある [98]．Sunday アルゴリズムは，Boyer-Moore 法における注目文字を末尾の文字から末尾の次の文字に変更することで，単純ながらシフト量を 1 文字増やすことが可能となっている．

Boyer-Moore 法は，1 パターンの探索における処理の高速化を目的としており，複数パターンに対する応用は考えられていない．特に，本手法は，事前にパターンから一致文字とシフト量の対応表を作成しなけれならない．そのため，パターン数に応じて対応表作成遅延や一致検索遅延，メモリ使用量は線形に増加する．アプリケーションルータにおけるサービスとして，例えば前述した snort を考えると，数千以上のパターンが存在する．このような膨大なパターンの探索に対し，Boyer-Moore 法は適していない．

既存研究 2: Aho-Corasick 法

近年最も盛んに研究されている文字列探索の高速化手法として，Aho-Corasick 法がある [99]．Aho-Corasick 法は，全てのパターンを合わせて一つの決定性有限オートマトン（DFA）を作成し，そのオートマトンに沿って状態遷移を行うことでパターンの一致を検索する．オートマトンを用いることで，Aho-Corasick の計算時間は理論的にはパターン数によらず入力文字列長にのみ依存する．

Aho-Corasick 法によるオートマトンの作成例を図 3.38 に示した．図では he，his，him，she，her，hers の 6 種類のパターンをオートマトン化した例を示している．図中の実線は遷移関数を，点線

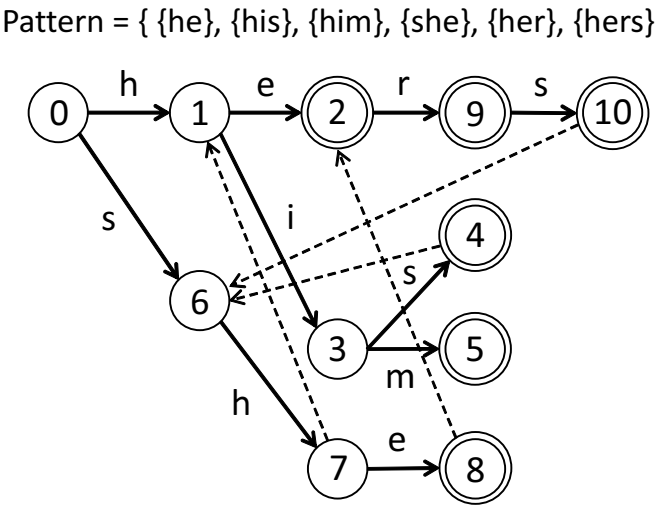


図 3.38: Aho-Corasick 法におけるオートマトンの作成例

は失敗関数を表している．Aho-Corasick 法は，このようなオートマトンを最初に作成し，入力文字の先頭からオートマトンに照らし合わせることで一致検索を進める．

本手法は，ハードウェアやマルチコアによる並列化，パイプライン化の恩恵を受けやすく，近年の FPGA や GPU を用いた高速化手法が多く研究されてきた．Alan Kennedy らは，Deep Packet Inspection に向けた，ハードウェアベースの並列化を用いた Aho-Corasick の改善手法を提案している [100]．Kennedy らは，65nm プロセスの Cyclone3 FPGA ボードに 4 並列度の提案手法を実装しシミュレーションを行った結果，7.5Gbps のスループットが得られたと述べている．また，同 65nm プロセスの Stratix3 FPGA ボードに 6 並列度の提案手法したところ，22.1Gbps が達成できたと述べている．

Tuck Nathan らは，侵入検知システムのための高速な Aho-Corasick アーキテクチャを提案している [101]．オートマトンによる状態遷移を，Bitmap 化手法を用いて最適化することで，ハードウェア実装に適したアーキテクチャとした．Tuck らのシミュレーションによると，ASIC 実装による文字列探索シミュレーションでは 7.8Gbps のスループットが達成可能となっている．

Aho-Corasick 法で用いられる DFA は状態遷移先が必ず一つに決まる特徴をもつ．しかしこの特性により複雑な正規表現では指数関数的な状態数の増加が生じ，メモリ使用量が増大することが以前から問題として指摘されてきた [102]．前述したようにアプリケーションルータのサービスが数千以上のパターンを持つとすると，最初の DFA 作成にかかるオーバーヘッドやメモリ使用量が問題となることが予想される．論文 [102] では，NIDS を想定し，パターン数を 5000，パターンの平均長を 20Byte とした場合，単純計算すると 2560 万の遷移状態が存在することになると述べている．また，アプリケーションルータでは，例えば NIDS シグネチャの更新や QoS アプリケーションの変更といったパターンの更新が柔軟に行われる．これに対して，Aho-Corasick 法はパターン更新毎にオートマトンを作り直さなければならず，このような状況に最適であるとは言えない．

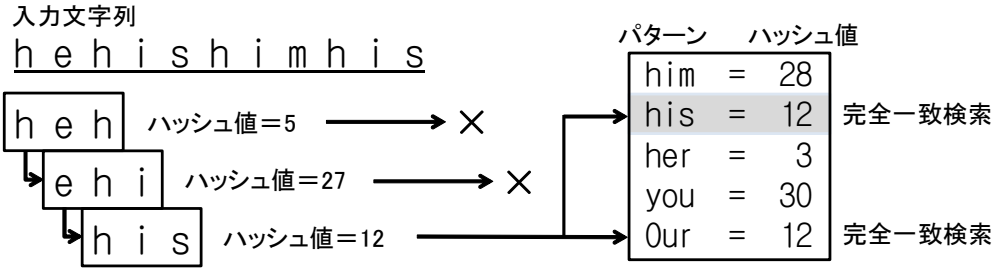


図 3.39: Rabin-Karp 法の処理概要

既存研究 3: Rabin-Karp 法

文字列探索を高速化するのではなく，事前処理によってオフロードする手法が研究されている．このような文字列探索におけるオフロードを目的とした代表的な手法として，ハッシュをベースにした Rabin-Karp 法がある [103].

Rabin-Karp 法では，入力文字列に対するパターンの探索はハッシュ値をもとに行う．パターンのハッシュ値と，同文字数の入力文字列のハッシュ値を比較することで，入力文字列がパターンである可能性があるかどうかわかる．ここで，ハッシュ値が一致しない場合は，入力文字列はパターンではない．一方で，ハッシュ値が一致した場合は，入力文字列がパターンかハッシュ値の等しくなる他の文字列ということになる．従って，ハッシュ値が一致した場合にのみ完全一致を検索すればよく，従来ボトルネックとなっていた完全一致検索処理の負荷を削減できる．

Rabin-Karp 法の概要を図 3.39 に示した．まず，事前準備としてパターンそれぞれのハッシュ値が計算され，パターンとハッシュ値の対応が管理される．そして，入力文字列をパターン長で切り出し，ハッシュ化する．ここで，パターン長が複数ある場合には，全てのパターン長分の入力文字列が切り出されるが，図 3.39 の例ではパターン長が 3 のパターンのみ用いられている．次に，得られた入力文字列のハッシュ値と同じハッシュ値を持つパターンの検索が行われる．ハッシュ値の等しいパターンが存在しなかった場合，入力文字列はパターンではないことが判別できる．ハッシュ値の等しいパターンが存在した場合には，そのパターンと入力文字列で完全一致検索を行う．ここで一致した場合に，入力文字列がパターンであることが判別される．図のように，ハッシュ値の等しいパターンが複数存在する場合には，そのパターンの数だけ完全一致検索が行われることになる．以上の操作を一文字ずつシフトさせながら行い，入力末尾まで繰り返すことでパターンの抽出がなされる．

Rabin-Karp 法では，ハッシュ値の一致があった場合にのみ完全一致検索を行うことで，文字列探索処理負荷を削減している．また，複数パターンであっても一度にハッシュ値が比較でき，計算量はパターン数によらない．そのため，マッチングすることの少ないパターンの検索や膨大なパターンの検索を行う場合に，特に性能が向上できる．Rabin-Karp 法では，完全一致検索処理部とハッシュによるフィルタリング処理部を分けて考られることから，完全一致検索モジュールには既存の研究，例えば Aho-Corasick 法などを適用することも可能である．そこで，従来では，ハッシュによるフィルタリング処理部に焦点をあて，これを改善する研究が行われてきた．

一般的に Rabin-Karp 法では，ハッシュ関数としてローリングハッシュをベースとした関数が用

いられるローリングハッシュは，入力データ列をシフトさせながら順にハッシュ化する場合に，前のハッシュ値を用いて次のデータ列のハッシュ値を高速に計算する，特殊なハッシュ関数である．例えば，1 文字目から 5 文字目までの文字列のハッシュ値が与えられた時，その値を用いて 2 文字目から 6 文字目までの文字列のハッシュ値を 1clock により計算できる．以下ではローリングハッシュに関してより詳しく説明する．

n 文字の文字列 s[1..n] が与えられている時に，m 文字のハッシュ値を 1 文字ずつシフトさせながら計算する場合を考える．ローリングハッシュでは（3.5）式によりハッシュ値が計算される．ここで，式中の s[m] は m 番目の文字の文字コードを表し，e はある素数を基数として用いたものである．このようなローリングハッシュでは，s[2..m+1] のハッシュ値は,（3.6）式により計算できる．一般的なハッシュ関数では，ハッシュ値の計算は文字列長 m に比例し，O(m) オーダとなるが，ローリングハッシュでは文字列長によらず O(1) により次のサブ文字列のハッシュ値が計算ができる．また，1 文字シフトさせたサブ文字列のハッシュ値を高速に計算できるだけでなく，ある文字から 1 文字のハッシュ値，2 文字のハッシュ値，3 文字のハッシュ値，と m 文字目までのハッシュ値を一つの計算の中で一度に得ることができる．

$$hash(s[1..m]) = s[1] \times e^{m-1} + s[2] \times e^{m-2} + ... + s[m] \times e^{0} \tag{3.5}$$

$$hash(s[2..m+1]) = (hash(s[1..m]) - s[1] \times e^{m-1}) \times e + s[m+1] \tag{3.6}$$

また，Rabin-Karp 法に Bloom filter を応用したアプローチ [104] がある．3.2.1 項で紹介したハードウェアコプロセッサ PowerEn は，論文 [10] によると，Bloom Filter String Matching（BFSM）[105] をベースに 8 個の文字列探索エンジンを搭載することで，トータル 20Gbps から 40Gbps のスループットが獲得できると述べている．Bloom filter は，メモリ空間を効率的に使うことのできる特徴をもつ．しかし，複数の hash 関数が必要となる．この複数の hash 関数を取り扱う方法は２つある．１つの hash メモリを複数回に分けてアクセスする方法と，メモリを hash 関数分用意する方法である．しかし，どちらも問題がある．前者は，bloom filter の高い空間効率を享受できるものの，処理ステップ数がかさむ．後者はハードウェアの並列度を利用して処理ステップ数は削減できるものの，ハードウェアコストがかさむ．Bloom filter にはこれらの欠点があり，G.Papadopoulos ら [106] も同様な欠点を指摘している．

Perfect hashing function を用いたアプローチもある [107]．Perfect hashing function は，候補パターンの hash 値が必ず異なるように調整した hash 関数である．このため，必ず，verify 候補を 1 つに絞ることができる．この hash 関数を用いて，hash table の代わりに，hash tree を生成する．Hash tree はメモリ上ではなく，ユニークな回路として構成される．verify 候補を 1 つに絞るユニークなアプローチだが，ハッシュツリーの再構築に回路の再構成が必要であり，パターンの迅速な更新ができない欠点がある．また，そもそも verify 候補を１つに絞る必要性が乏しい．ヒット時の verify を Verify が不要なヒットしなかった時の時間も利用して行えば，verify に数ステップを要してもよい．特に NIDS のパターンが頻繁にヒットする状況は通常ないため，verify に数ステップかけることは問題にならない．Verify を 1 ステップで終了させる必要がなければ，一般的なメモリを利用した構造で済み，回路の再構成が不要なため迅速なパターン更新が可能となる上，シ

ンプルな構成でエリアコストも削減できる．

一般的なメモリを用いた hash のアプローチとして，hashMem[106] というアーキテクチャが提案されている．しかし，パターン長毎に hash テーブルが必要であり，ハードウェアコストが大きい．また，パターン長毎に verify 用のコンパレータを持つ機構だが，各コンパレータはパターン長の異なる verify には用いることができず，非効率である．長いパターンを分割することでハードウェアコストを削減するタイプも hashmem にはあるが，分割された hash 値を統合してマッチを判断する回路が別に必要となる．

アプリケーションルータにおける文字列探索処理:

ここまで紹介したように，近年の研究による文字列探索処理は，Boyer-Moore 法，Aho-Corasick 法，Rabin-Karp 法のいずれかの手法を，ASIC や FPGA といったハードウェアに実装することで高速化を図っている [108, 109]．しかしながら，これらの研究により達成されるスループットは数 Gbps から数 10Gbps 程度であり，100Gbps の文字列探索スループットを獲得することは困難であった．従って，文字列探索処理もボトルネックの一つであると考えられ，処理の高速化，あるいはオフロードが必要となる．

3.2.4 データベースエンジン

文字列探索処理における抽出結果や抽出ルールといった様々な情報はルータ内のデータベースに格納される．ここで，データベースへのデータ挿入や抽出，更新，削除といったエントリを管理する機構がデータベースエンジンである．

3.2.4.1 アプリケーションルータにおけるデータベースエンジンの概要

アプリケーションルータにおいてデータベースのセレクション処理は，アプリケーションからの要求があった場合のみであるため，そこまでの頻度を要しない．それに対して，データベースへのインサーション処理では，ネットワークトラフィックに対して情報抽出した結果をワイヤレートに挿入する必要があるため，スループットが求められる．このような要求に対し，近年では In-Memory Database（IMDB）と呼ばれるデータベースエンジンが提案，実用化されている [110]．IMDB はデータをディスクではなくメモリ上に格納することで，高速なデータベース操作を可能とする．更に，西田らはアプリケーションルータにおけるデータベースエンジンとして，論文 [111] において，IMDB と従来の Storage Database を併用したハイブリッドメモリデータベースエンジンを提案している．以降では，アプリケーションルータにおけるハイブリッドメモリデータベースエンジンについて説明する．

3.2.4.2 ハイブリッドメモリデータベースエンジン

アプリケーションルータでは，前述したように，データベースからの高速なデータ読み出し処理よりも高速なデータ書き込み処理が重要となる．そこで，ハイブリッドメモリデータベースエンジンでは，IMDB として機能するメモリにデータを書き込む．このメモリは一次データベース

として機能し，付随するメモリ内にあるインデックス情報により管理される．インデックス情報には保持期限，パケットが到着した時間，所有者情報などが管理される．これらは，格納されたデータの自動削除および強制削除，データベース容量制限の設定，セキュリティ管理などにおいて頻繁に使用される情報である．

IMDB はハードディスクと比較して容量が小さいため，限られたリソースを有効活用しなければならない．現存のサーバにおける搭載メモリの最大容量は 512GB 程度でありアプリケーションによっては十分とはいえない．そこで，ハイブリッドメモリデータベースエンジンでは，すべてのデータについて保持期限と優先度の設定を義務付け，インサートされた情報は保持期限に従って自動的に削除または Storage Database への永続化がなされる．Storage Database はディスクベースのデータベースであり，ディスクの容量を活かして IMDB に蓄えきれなくなった高優先度のデータを蓄える二次データベースとして働く．Storage Database は選択演算の高速化などを考慮し，B+-tree を用いた一般的なデータベースソフトウェアの管理下におかれる．

アプリケーションルータではこのように二階層のデータベース機構を採用することで，IMDB によってデータベースアクセスを高速化しつつ優先度の高いデータを Storage Database で永続化する．優先度が高く設定されたデータは永続化が完了した際に IMDB メモリ上から削除される．ただし，永続化が行われるかどうかはアプリケーションに依存し，永続化が行われなければ保持期限により削除される．さらに，限られたデータベース容量を効率良く利用するために複数のクエリ間で重複するデータは一つにまとめて書き込む．

近年では，IMDB と Storage DB を組み合わせたハイブリッドメモリデータベースエンジンが数多く提案され，実用化されている [112, 113]．更には，上述したようなアプリケーションルータに特化したデータベースエンジンの研究も進められてきた．論文 [114] によると，IMDB を用いたデータベースインサーションの処理速度は 200Ktps（Transaction per second）程度である．一方で，100Gbps ネットワークのパケット数は，パケットの平均データサイズ 526Byte[4] を用いると 20Mpps 程度と計算される．ここで，アプリケーションルータにおけるデータベースインサーションの発生は，パケットが抽出ルールに一致した場合のみであることを考慮すると，20Mtps を下回るはずである．そこで，石田らの公開するサービス指向ルータのソフトウェアシミュレータを用いて，実ネットワークトレースに対し抽出ルールに一致するパケットの抽出を行い，その抽出頻度を測定した．抽出ルールとしては，ルールにマッチングするパケットの割合が高いと考えられる，web ページタイトルを抽出するための文字列 <title> および <TITLE> を設定した．まず，ある国内の 10Gbps コアネットワークで取得した 2012 年 9 月 13 日のトレースを用いてパケットを抽出したところ，およそ 253 パケットに 1 パケットの割合で web ページタイトルが抽出された．更に，2015 年に開催されたある大規模展示会における 40Gbps ネットワークのトレースを用いてパケットを抽出したところ，およそ 277 パケットに 1 パケットの割合で web ページタイトルが抽出された．このことから，データベースへのインサーションは，抽出率の高いルールであっても 250 パケットに 1 回程度の発生であり，100Gbps ネットワーク環境で考えても 80Ktps 程度である．従って，現状でも 100Gbps ネットワークにおいて情報抽出を行うのに十分なデータベースインサーションの処理速度は得られる．また，本項の初めに述べたように，アプリケーションルータのデータベースエンジンにおけるセレクション操作に関しては，アプリケーションからの要求があった場合のみであるため，インサーション程の処理速度を求められない．これらのことから，本論文で

はアプリケーションルータにおけるデータベースエンジンはボトルネックとはならないと考える．

3.2.5 アプリケーション

図 3.19 に示したアプリケーション機構では，データベースに蓄えられた情報をサービスとして使用できる形に整形する．例えば，NIDS ならばルールに一致したパケットのフロー情報とそのフローに対するアクションを決定し，ACL テーブルに追加する．場合によっては，精査やログのためにルーティングテーブルの内容を変更する．アプリケーションルータを用いた QoS 保証サービスならば，抽出された情報から当該パケットの優先度を決定し，そのパケットの属するフロー情報と優先度情報を QoS テーブルに追加する．また，web 行動解析ならば得られた情報から web ページの滞在時間やアクセス回数といった統計情報を解析する [45, 46]．このように実装するアプリケーションにより処理負荷は異なるが，ネットワークのワイヤレート処理に関係する部分ではないため，基本的にはソフトウェア処理で十分であると考えられる．

3.3 アプリケーションルータにおける性能ボトルネック

ここまで，アプリケーションルータの各処理機構とその既存研究について詳細を述べてきた．そこで本節では，前節から導かれるアプリケーションルータの性能ボトルネックについて分析する．これまで述べてきた各処理機構の達成スループットを，参考となる文献および研究と共に表 3.18 にまとめた．

ルーティングおよびフォワーディングに関しては，近年の商用パケット転送ルータにおいて 100Gbps のスループットが得られていることを 3.1.2 項および 3.1.4 項で述べた．一方で，アプリケーションルータのプロセッシングに関しては，3.1.3 節で述べたように，GZIP 展開処理および文字列探索処理がボトルネックとなり 100Gbps ネットワークを処理するために十分なスループットが得られていない．まず，既存の GZIP 展開手法では，ネットワーク経路上でのパケット断片化の問題を考慮した場合，1Gbps 程度のスループットを得ることが限界であり，100Gbps ネットワーク下でのワイヤレートな GZIP 展開処理に要する 5.5bps のスループットを達成することは困難である．そこで，本論文では高速な GZIP 展開アーキテクチャを提案する．

また，文字列探索処理において，既存の手法による処理速度は数 Gbps から数 10Gbps 程度であり，100Gbps にはおよばないことを説明した．しかしながら，アプリケーションルータでは，実際に 100Gbps の文字列探索性能を確保する必要はない．なぜならば，文字列探索が不要な部分（例えばレイヤ 4 以下のヘッダ情報）や不要なパケット（例えば web レコメンデーションサービスにおける HTTP 以外のパケット）を除くことでオフロードが可能だからである．このようなネットワーク経路上での情報抽出に特化した文字列探索の最適化は従来研究されてこなかった．そこで，ネットワーク経路上での情報抽出に特化した文字列探索オフローダの提案を行う．

一方で，アプリケーションがルーティングテーブルや ACL，QoS テーブルといった多くのテーブルに対して追加操作を行うことを 3.2.5 項で述べた．これによって，アプリケーションルータではテーブル検索処理の負荷が増大することが考えられる．これまで述べてきたように，ルータにおける各種テーブル検索はワイルドカードを用いた検索や LPM が行われるため処理負荷が高

く，近年では TCAM を潤沢に用いることで検索を高速化してきた．しかしながら，TCAM により得られるテーブル検索スループットは 100Gbps 程度であり，アプリケーションルータにおける 100Gbps ネットワークのパケット処理を想定した場合，テーブル検索負荷の増大に対処できないことが危惧される．

更に，TCAM の高い消費電力と実装コストは以前から問題点として指摘されている [115, 116, 117, 118]．TCAM はメモリ内のデータに対し，1cycle で全ゲートに通電する．従って，その探索方式の違いから RAM に比べて消費電力が高い．また，データ 1bit 毎に比較器を用意することから回路規模が増大し，それに伴い製造コストの削減も困難である．Weirong らは論文 [119] で，参考文献をもとに同規模の TCAM と SRAM のコストを比較している．それによると，消費電力は 120 倍程度，1bit あたりのトランジスタ数は 2.7 倍程度 TCAM のほうが高くなる．このように消費電力および実装コストが大きい TCAM を多用することは，ルータにとって最適な実装であるとは言えない．

阿多らは，総務省の公表した「環境負荷低減に資する ICT システム及びネットワークの調査研究会報告書」[120] を基に，2010 年のルータによる消費電力が国内総発電量の 0.4 から 1.7%に達することから，大きな社会問題へ発展していると言及している [116]．更には，経済産業省の公表したエネルギー消費量予測 [121] によると，国内の ICT（Information and Communication Technology）機器消費電力は 2006 年から 2025 年にかけておよそ 5.2 倍に増加する見込みであり，国内総発電量の 20%に達する恐れがあると述べられている．これは，同報告書で述べられている 2006 年から 2010 年にかけての消費電力増加予測が 1.3 倍であるのに対し，2010 年以降は加速度的に通信機器の需要が増える見込みであることを意味する．このような問題に対して，2006 年以降，総務省による「低消費電力型通信技術等の研究開発（エコインターネットの実現）」[122] や，経済産業省による「グリーン IT イニシアティブ」[121] といった，ICT 機器，特にルータに関する消費電力削減への取り組みが活発に行われている．しかしながら，データセンタでの消費電力は 2010 年に 100 億 kWh 程度であったのに対し，2014 年度に 140 億 kWh 程度と，年 10%程度の増加傾向にある [123]．従って，ルータ消費電力の低減はいまだ重要な課題である．

ルータの消費電力が増大する要因として，図 3.3 で示したように，近年のネットワーク経路数の加速度的な増加に伴うルーティングテーブルエントリ数の増加が挙げられる．特に，今後の IPv6 への対応を考慮すると，TCAM に格納すべきデータサイズが増えることで，TCAM 容量が増大することが懸念される．TCAM では，前述したように，メモリ内の全ゲートに一度に通電するため，メモリ容量と比例して消費電力が増大する．このように，消費電力の観点から見ても，アプリケーションルータにおけるテーブル検索処理はボトルネックになると考えられ，TCAM のみに頼らない新たなテーブル検索機構を併用する必要がある．

そこで本論文では，上述した三つのボトルネックである GZIP 展開処理，文字列探索処理，テーブル検索処理に焦点を当て，それぞれを解決するアーキテクチャを提案することで，100Gbps ネットワーク下におけるアプリケーションルータでの情報抽出およびテーブル検索を実現する．

表 3.18: アプリケーションルータにおける各処理のスループット

分類	処理機構	実効スループット
ルーティング	ルーティングテーブル検索	100Gbps [32]
	アドレス解決	100Gbps [32]
プロセッシング	ACL	100Gbps [32]
	QoS	100Gbps [32]
	GZIP 展開（パケット断片化考慮）	18Gbps [2]
	（パケット断片化未考慮）	36Gbps [8]
		36Gbps [9]
		145Gbps [10]
	文字列探索	7.5Gbps [100]
		7.8Gbps [101]
		20-40Gbps [10]
	データベースエンジン	100Gbps [111, 114]
		（1 挿入/100 パケット）
フォワーディング	モディフィケーション	100Gbps [32]
	スケジューリング	100Gbps [32]

第4章　逐次展開ハードウェアによるGZIP展開の高速化

4.1　本研究の動機

第 3 章では，近年の HTTP 通信においてデータの GZIP 圧縮化が普及しはじめていることについて述べた．これに対して，ネットワーク経路上における情報抽出に関する既存研究の多くでは，必要となるメモリ量や処理レイテンシの問題から，GZIP 圧縮された HTTP トラフィックの展開は考慮されなかった．そのため，GZIP 圧縮された HTTP トラフィックに対して情報抽出を行うことが困難であった．このような背景に加え，関連研究で述べたように従来提案されてきた GZIP 展開機構を用いて 100Gbps 以上の高速なネットワーク処理を達成することが困難であったため，アプリケーションルータに実装可能なアーキテクチャの提案もなされなかったと考えられる．そこで，本研究では 100Gbps ネットワークを想定し，アプリケーションルータにおいて GZIP 圧縮された HTTP トラフィックに対し，ワイヤレートな展開処理を可能とするアーキテクチャの提案を行う．

4.2　高速な GZIP 逐次展開ハードウェアの提案

3.2.2 項で述べたように，近年のネットワークにおける GZIP 圧縮データの割合はおよそ 5.5%である．このことから，100Gbps ネットワーク下におけるワイヤレートな GZIP 展開には 5.5Gbps のスループットが求められる．また，今後更に GZIP 圧縮の需要が高まることを考慮すると，5.5Gbps 以上のスループットを達成できることが望ましい．これに対して，3.2.2 節では，一般的な zlib ライブラリを用いて最新のプロセッサにより展開した場合や複数の FPGA を用いて展開した場合でも，2Gbps 程度の処理性能しか得られないことを示した．論文 [10] では，ハードウェアコプロセッサ PowerEn により 8Gbps の展開性能を獲得できると述べられているが，回路規模が格段に大きいといった問題があることを述べた．更には，多くの既存研究における GZIP 展開は，ネットワーク経路上での処理を想定しておらず，パケット断片化の影響を考慮していない．このことから，本論文では，パケット断片化に対応し，なおかつ高スループットな処理性能を達成できるハードウェア GZIP 逐次展開アーキテクチャを提案する．

まず，アプリケーションルータにおける GZIP 展開のガイドラインとして，GZIP 逐次展開プロセスのフローチャートを図 4.1 に示す．なお，パケット断片化への対応には，3.2.2 項で述べた石田らの研究における拡張されたコンテクストスイッチ手法を用いる．本論文では，図 4.1 のフローチャートに従う GZIP 展開機構をハードウェア化することで高スループットを達成する．更に，GZIP 逐次展開における性能ボトルネックを改善する手法を併せて提案する．GZIP 逐次展開の性能ボトルネックとしては以下の三つが考えられる．

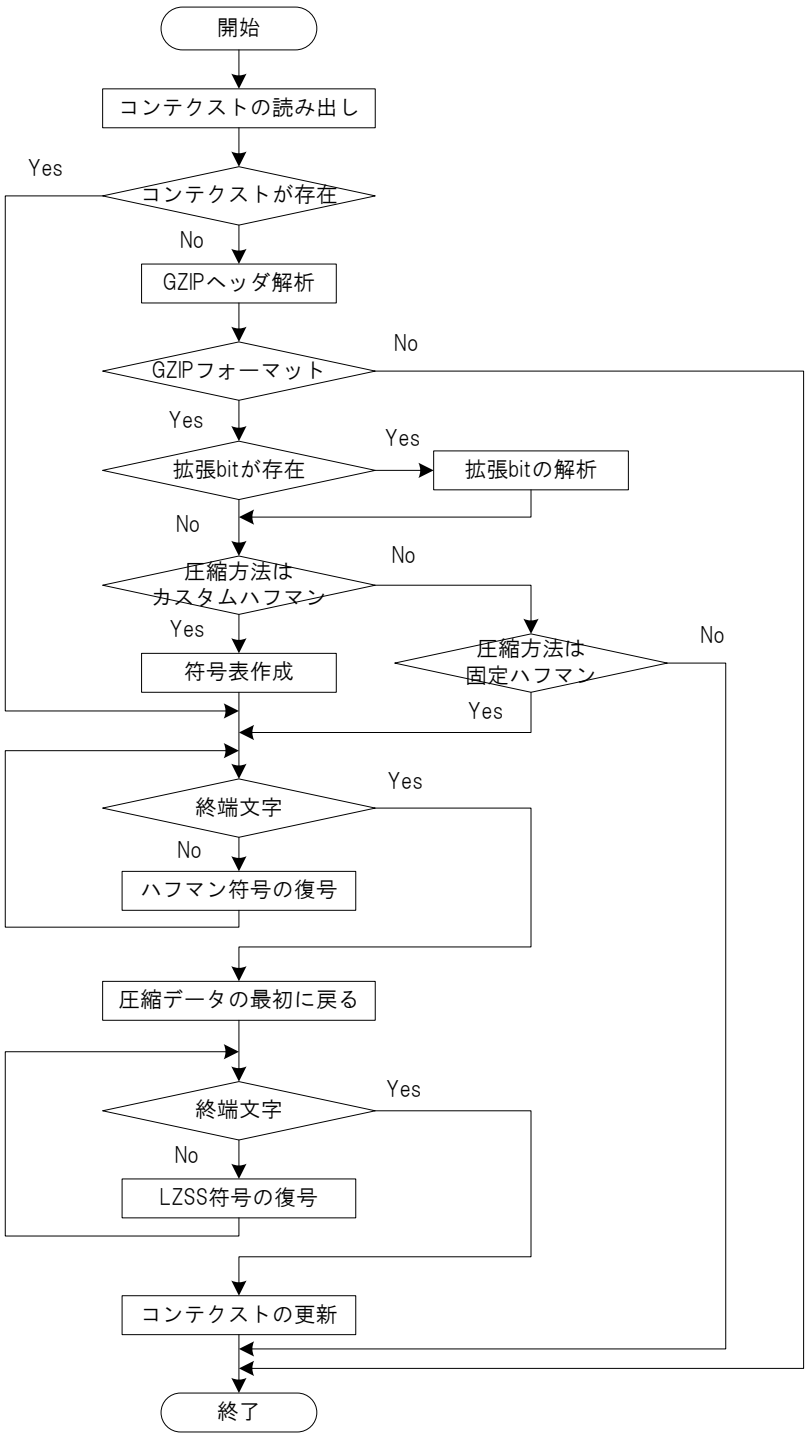


図 4.1: アプリケーションルータにおける GZIP 展開フローチャート

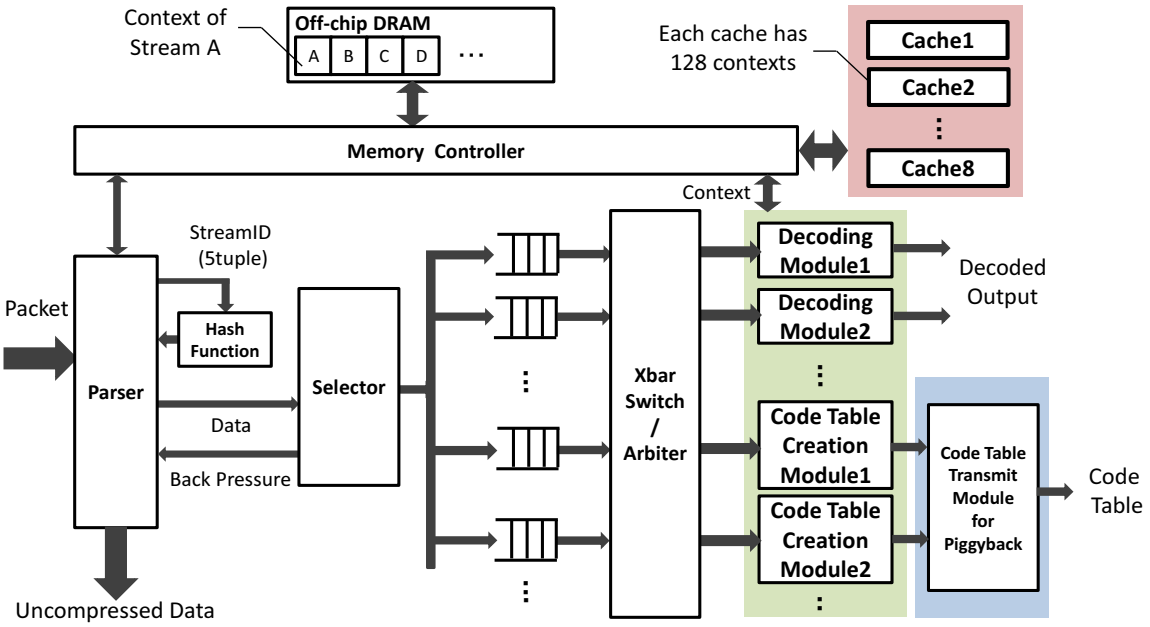


図 4.2: 本論文で提案する高速な GZIP 逐次展開アーキテクチャの概要

1. コンテクスト読み出しにかかる遅延
2. 符号表作成にかかる遅延
3. 復号にかかる遅延

そこで，これら三つのボトルネックに焦点を当て，ボトルネック 1 の改善としてコンテクストキャッシュ機構を，ボトルネック 1 および 2 の改善としてパラレルデコーディングの改良手法を，ボトルネック 2 および 3 の改善として並列アーキテクチャを GZIP 展開ハードウェアに加える．更には，複数間アプリケーションルータの協調によりボトルネック 2 を改善する手法としてピギーバック手法を提案する．コンテクストキャッシュ機構は 4.2.7 項で，パラレルデコーティングの改良手法は 4.2.5 項で，ハードウェアの並列化は 4.2.4 項および 4.2.5 項で，ピギーバック手法は 4.2.8 項で詳しく述べる．本論文で提案する高速な GZIP 逐次展開アーキテクチャの全体図を図 4.2 に示す．以降では，各モジュールおよび改善手法について処理順に述べる．

4.2.1 構文解析，ハッシュモジュール

図 4.2 に示された Parser（構文解析）モジュールでは，主にストリーム ID となる 5 タプルの抽出とコンテクスト情報の検索，GZIP プロトコルの解析がなされる．構文解析モジュールおよびハッシュモジュールにおける処理の流れを図 4.3 に示す．以下では，構文解析モジュールの処理を順を追って説明する．

1. **HTTP ヘッダの抽出:**
   構文解析モジュールでは，まず HTTP ヘッダまでの構文解析がなされる．ここでの主な処理

は，5 タプル情報の抽出と HTTP ヘッダ末尾の探索である．5 タプル情報はレイヤ 3 およびレイヤ 4 ヘッダを固定長解析することで得られる．得られた 5 タプルは，ハッシュモジュールに転送され，ハッシュ値となる．HTTP ヘッダの末尾はパケット内から”\r\n”のみの空行を探索することで得られる．

2. **5 タプルのハッシュ化:**
上記と同時に，5 タプル値はハッシュモジュールへと転送され，104bit から 15bit にハッシュ化される．このハッシュ値は，メモリやキャッシュのインデックスに用いられる．ハッシュモジュールの実装において，本研究では MurmurHash を用いた．MurmurHash[124] はハードウェアベースのハッシュ関数の中でも Avalanche 特性のよい関数として知られる [125][126]. 処理遅延に関しては，CRC[71] や Jenkins ハッシュ [127] に比べ MurmurHash が最も大きいが，GZIP 復号処理においてはその遅延の影響は無視できるほどに小さい．

3. **コンテクスト情報の検索:**
次に，パケットが属するストリームの GZIP コンテクスト情報の有無が検索される．検索は，メモリコントローラを介してコンテクストメモリに対して行う．ここで，コンテクスト情報が存在した場合にはパケットがストリームの途中パケットであることがわかり，コンテクスト情報が存在しなかった場合にはパケットが非 GZIP 圧縮データまたはストリームの最初のパケットであることがわかる．前者の場合には，先頭に 00 と 5 タプルのハッシュ値を付与し，後に続くデータが GZIP 圧縮データであることからそのままセレクタへと転送する．一方で，後者の場合には，GZIP フォーマットを判断するために GZIP ヘッダ解析がなされる．

4. **GZIP フォーマットの判別:**
もし，パケットが HTTP 圧縮データである場合には，HTTP ヘッダの終わりを示す”\r\n”の後に GZIP ヘッダが続くはずである．GZIP フォーマットはこの GZIP ヘッダの先頭 2 バイトにより判断できる．先頭 2 バイトが 0x1f，0x8b であった場合，そのデータは GZIP 圧縮データである．これによって，パケットが GZIP フォーマットであった場合にはヘッダの解析が続けられ，GZIP フォーマットでなかった場合には非圧縮データであるとして，入力がそのまま出力される．

5. **GZIP ヘッダの抽出:**
続いて，パケットが GZIP フォーマットであった場合の GZIP ヘッダ解析について述べる．GZIP ヘッダのデータサイズは複数の拡張フィールドによって変動する．3.2.2 項では，FLG の値により 4 つの拡張フィールドの有無が判断されることを述べた．そこで，構文解析モジュールでは，FLG 値によって以下に続く拡張フィールドを検知する．ここでは FLG の各ビットを FLG[i] と表す．表 4.1 に各 FLG 値を検知した場合の処理を処理順に示す．この処理によって，GZIP ヘッダの外された DEFLATE ブロックが得られる．

6. **DEFLATE ヘッダの抽出:**
更に，DEFLATE ブロックの先頭 1bit は BFINAL として破棄され，続く 2bit は BTYPE として抽出される．BTYPE の値によって，ハフマン符号の圧縮方式が判別する．BTYPE が 00

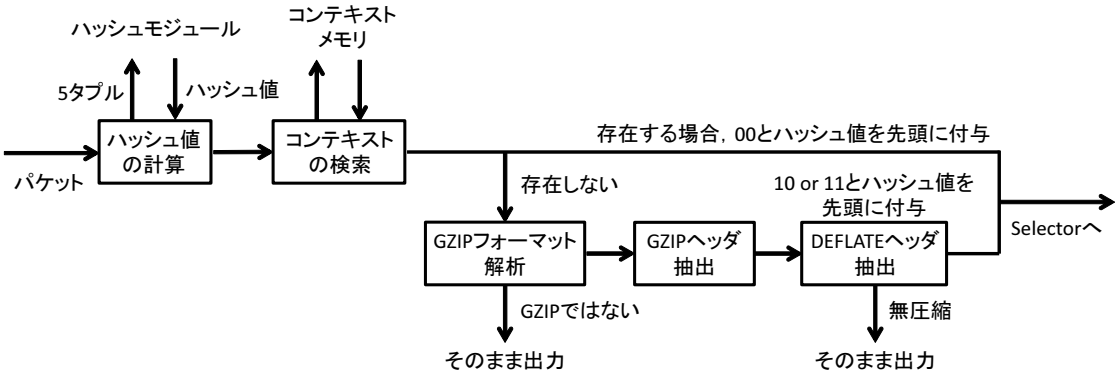


図 4.3: 構文解析モジュールおよびハッシュモジュールにおける処理フロー

表 4.1: FLG 値による GZIP ヘッダ解析モジュールの処理

FLG 値	処理
FLG[2]	2Byte の XLEN を把握．更に XLEN Byte の拡張フィールドを処理
FLG[3]	ファイル名が存在．0 終端まで破棄
FLG[4]	コメントが存在．0 終端まで破棄
FLG[1]	CRC の 2Byte を破棄

であった場合には後に続くデータが非圧縮であることから，入力データはそのまま出力される．BTYPE が 10 および 01 であった場合には符号表を作成する必要があることから，カスタムハフマンなら 11，固定ハフマンなら 10 の値と 5 タプルのハッシュ値をデータ先頭に付与し，入力データをセレクタへ転送する．

また，構文解析モジュールはセレクタから発された Back Pressutre を受信した際には，セレクタへのデータ出力を中断する．この Back Pressure は，セレクタへのデータの入力に対し，セレクタ以降の GZIP 展開処理が追いつかない場合にセレクタより発せられる．セレクタ以降の処理待ちキューがある程度処理されたところで Back Pressure 信号は下げられ，中断していた構文解析モジュールのデータ出力が再開される．

4.2.2 セレクタ

構文解析モジュール，ハッシュモジュールより転送されたデータはセレクタによって各キューに蓄えられる．割り当てるキューは 5 タプルのハッシュ値をもとに決定される．このハッシュ値の下位 7bit は後述するコンテクストキャッシュのインデックスに用いられるが，続く数 bit によって割り当てキューを決定する．例えば，キュー数が 8 ならば，ハッシュ値の 8bit から 10bit の値によって割り当てキューが決定される．5 タプルのハッシュ値によりキューを割り当てることで，ストリームのパケット順を乱すことなく処理が行える．また，セレクタは割り当てキューが一杯であった場合に 1bit の Back Pressure を構文解析モジュールに送信する．

4.2.3 キュー，クロスバースイッチ，アービタ

セレクタから転送されたデータはリングバッファで実装されたキューに蓄えられる．後述する符号表作成モジュールと復号モジュールの並列化の恩恵を受けるため，これら処理モジュールと同数のキューが用意される．キューと両処理モジュールはクロスバースイッチにより接続されており，全てのキューと全ての処理モジュール間でデータ転送が可能となっている．ここで，データ転送を行うキューとデータを受信する処理モジュールの組み合わせはアービタにより決定される．

アービタは，キューの先頭データが発する要求信号と，処理モジュールが発する受付信号を調停する役割を担う．キューの先頭データは，先頭 1bit の値によりコンテクスト情報の有無が判断され，先頭 bit が 1 ならストリームの最初であるとして符号表作成要求信号を，先頭 bit が 0 ならストリームの途中であるとして復号要求信号をキューから発する．また，符号表作成モジュールおよび復号モジュールは処理を行っていない時に，それぞれ符号表作成受付信号と復号受付信号を発する．アービタはこれらの信号をラウンドロビン方式により調停し，転送キューと受信モジュールを決定する．アービタからの Enable 信号が出たキューは，アービタに指示された処理モジュールへとデータを転送する．

ここで，キューが出す信号が復号要求信号であった場合には，データ転送後に要求信号を下げ，キューの先頭データを削除する．一方で，符号表作成要求信号であった場合は，先頭データを削除せず，要求信号を下げる．そして，符号表作成モジュールの処理完了後に送られてくる Fin 信号を検知したら，続いて復号要求信号を発する．この時，Fin 信号と同時に DEFLATE ブロックの符号表データサイズが符号表作成モジュールより送られてくるため，この値をキューの先頭データ末尾に格納する．この値は，復号モジュールにおいて初めの符号表を破棄するために使われる．

4.2.4 符号表作成モジュール

符号表作成モジュールに送られるデータは，構文解析モジュールによって，コンテクストの有無を示す 1bit，ハフマン圧縮の方式を示す 1bit，5 タプルのハッシュ値 15bit，DEFLATE ブロックフォーマットの符号表，圧縮データとなっている．まず，先頭の 1bit が 1 であるか判断される．先頭 1bit が 0 であった場合には，データは破棄される．続く 1bit によりハフマン圧縮の方式が判別される．この値が 0 であった場合には，固定ハフマン方式であることから，デフォルトの符号表が出力となる．一方で，この値が 1 であった場合は，カスタムハフマン方式であることから，以下の処理に従って符号表が復元される．

3.2.2 項で述べたように，カスタムハフマンの符号表は符号長の符号表，文字/一致長コードの符号表，戻り距離の符号表という三つの符号表を，更に符号長だけで表したものであり，符号表作成モジュールにおいて三つの符号表を復元する必要がある．本項では，処理の概要について説明する．

まず符号表データの先頭 14bit が抽出され，HLIT，HDIST，HLEN として値が保存される．これらの意味については 3.2.2 項で述べた．図 3.25 を共に用いて説明する．まず，符号長の符号表を復元する．符号長の符号表は，符号長の個数 HLEN に繰り返し記号 4 つを合わせた HLEN+4 個のコードを，更にハフマン符号化した際の符号長が 3bit で順に添付されている．従って，先頭か

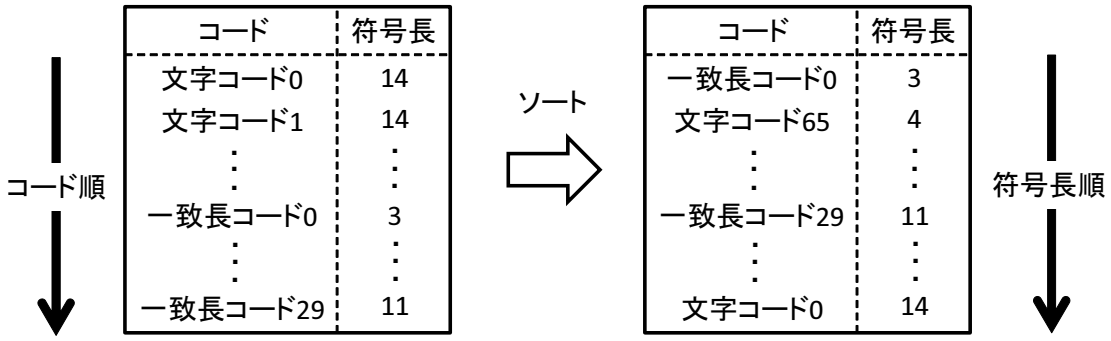


図 4.4: 符号表データのソート

ら 3bit ずつ読み取り，3bit×(HLEN+4) まで読み進めることで，符号長の符号表が復元できる．ここで，符号長 1 から 15 に対応する可変長のハフマン符号が得られる．

この後に，文字/一致長コードの符号表が続く．ビット列を先頭から読み進め，最初に完全一致したハフマン符号を，先ほど得られた符号長に変換していく．ここで，ハフマン符号は文字コード，終端文字コード，一致長コードの順に並べられている．即ち，ビット列の先頭には，文字コード 0 のヌル文字のハフマン符号長が記されており，次に文字コード 1 のヘッダ開始文字のハフマン符号長が続く．文字コード 256 種，終端文字コード 1 種，一致長コード 29 種の全 286 種のハフマン符号を復号することで，各コードのハフマン符号長が得られる．

ここからハフマン符号表を復元するためには，図 4.4 に示すように，まずコード順となっているハフマン符号長データを符号長の昇り順にソートしなければならない．この際，ハードウェアで高速にソートする手法として奇偶転置ソート [128] を用いる．

奇偶転置ソートは，ハードウェア向けに高速化されたバブルソートである．バブルソートは，最初のフェイズで隣同士の値を比較し，交換する．この操作をデータの最後まで繰り返すことで，一番大きな値が列の最後に出現する．これを一つのサイクルとして更に繰り返すことで，ソートが完了する．しかしながら，バブルソートはシーケンシャルにソートの各ステップを踏む必要があった．そこで，奇偶転置ソートはパイプラインを用いてバブルソートを最大限に並列化する．奇数 N 番目のデータと偶数 N+1 番目のデータの比較および交換を，1 サイクルで全て同時に行う．次のサイクルでは偶数 M 番目のデータと奇数 M+1 番目のデータの比較および交換を，同時に行う．この操作を繰り返すことで，ハードウェアの並列性を用いて効率的にソートが完了する．文字/一致長コードの符号表に奇偶転置ソートを用いることで，最大 285cycle によりソートが完了する．図 4.5 に，奇偶転置ソートの例を示す．

ソートにより得られた，符号長順の符号表データを用いて，符号表を復元する．符号表の復元は以下の操作に従う．また，このアルゴリズムに基いて，ハフマン符号を復元した例を図 4.6 に示す．最終的に文字/一致長コードの符号表の復元が完了する．

1. 先頭にある最も符号長の短いコードに，符号長の長さの 0 を割り当てる．

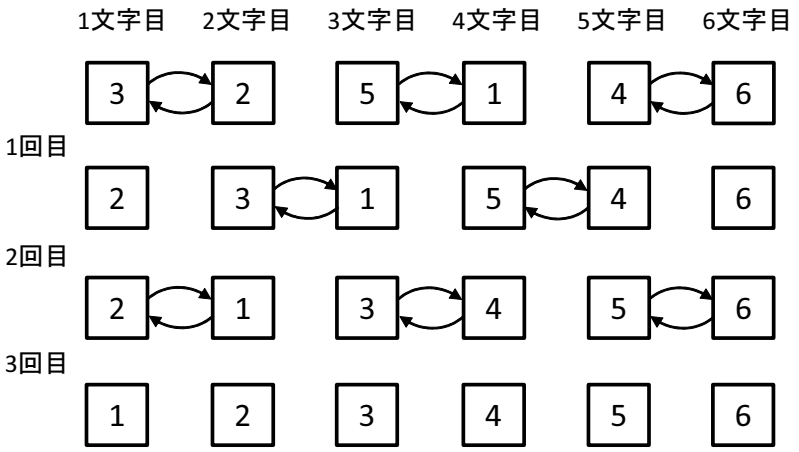


図 4.5: 奇偶転置ソートの例

複合される文字	ハフマンコード長	ハフマンコード
9	2	00
6	3	010
7	3	011
8	3	100
0	4	1010
5	4	1011
10	4	1100
11	4	1101
4	5	11100
12	5	11101
16	5	11110
17	6	111110
3	7	1111110
18	7	1111111

+1 & 右端に0足す
+1
+1
+1 & 右端に0足す
以下同じ

図 4.6: ハフマン符号の作成例

2. 次のコードのハフマン符号は，一つ上の符号に 1 を足した値となる．ここで桁数が増える場合は，1 を足した後に，桁数となるように左シフトする．

3. 以下同様にしてハフマンコードが生成される．

この後に，戻り距離の符号表が続く．符号表の作成方法は文字/一致長コードと同様である．一つ目の符号表で得られた符号長のハフマン符号を用いて，先頭からビット列を復号する．そして，得られた戻り距離コード順の符号長を，奇偶転置ソートを用いて符号長順にソートする．符号長の短い戻り距離コードから順にビットを割り当てることで，ハフマン符号を復元する．

全ての処理を終えることによって，文字/一致長コードとハフマン符号の対応表と，戻り距離コー

ドとハフマン符号の対応表という二つの対応表が得られる．これらの対応表は，それぞれハフマン符号の値が小さい順にソートされたままである．文字/一致長コードの対応表サイズは，文字/一致長コード 286 種類を表すために 9bit，それと対応するハフマン符号を表すために最大で 15bit，すなわち (9bit+15bit) × 286 で 858Byte となる．また，戻り距離コードの対応表サイズも同様に計算すると，(5bit+15bit) × 30 で 75Byte となる．本研究では，コンテクストスイッチを行う都合上，符号表のサイズはできる限り小さいことが望ましい．そこで，後述するパラレルデコーディングの改良手法を用い，符号表サイズを更に削減している．本手法の詳細については 4.2.5 項で述べるが，本手法により符号表は 423Byte となる．

完成された符号表は，5 タプルのハッシュ値と共にメモリコントローラへと転送され，キャッシュおよびコンテクストメモリの該当するエントリに書き込まれる．更に，データを転送してきたキューに対し Fin 信号および符号表作成に要した DEFLATE ブロックの bit 数を送ることで，続く復号処理への移行がなされる．最後に自身の符号表作成受付信号を上げ，符号表作成モジュールはデータの受信待ち状態となる．

なお，符号表作成処理は GZIP 展開の性能ボトルネックの一つであることから，本研究ではモジュールを複数並列化し，遅延を隠蔽する．ネットワーク経路上における GZIP 展開では，スループットの観点から複数のストリームを同時に処理することが求められる．ここで，ストリーム毎の処理は完全に独立しているため，ハードウェア並列化の恩恵を大きく受けることができる．

4.2.5 復号モジュール

GZIP 展開において，二つの符号の復号処理は最も遅延の大きい処理である．そこで，本研究では，符号表作成モジュールと同様に，復号モジュールを複数配置し並列化させることで，この処理遅延を隠蔽している．符号表作成と同様に，復号処理はストリーム毎に独立しており，ハードウェア並列化の恩恵が受けられる．

復号モジュールでは，まず，5 タプルのハッシュ値をもとにコンテクスト情報の読み出しが行われる．それと並行して，データの整形が行われる．具体的には，以下の処理に従う．

まず，データの先頭 2bit を抽出し，そのデータのストリーム内での位置とハフマン圧縮の方式を判別する．同時に 5 タプルのハッシュ値 15bit も抽出される．ここで，先頭 2bit の解析の結果，データがストリームの途中であったり，ストリームの最初だが固定ハフマン方式により圧縮されていると判別した場合には，後に続くデータが圧縮データ本体となる．一方で，データがストリームの最初であり，カスタムハフマン方式により圧縮されていると判別した場合には，DEFLATE フォーマットの符号表データが続く．そこで，後者の場合には，データ末尾に付与された値だけ，bit シフトさせる．この値は符号表作成モジュールによって与えられている．上記の操作によって，データの先頭が圧縮データ本体となり，データが整えられる．

更に，コンテクスト情報から得られる，前パケットで復号できなかったデータを先頭に付け加える．GZIP 展開における最大長の符号は，LZSS 符号 < 1，一致長コード，戻り距離コード > である．この符号がパケット末尾で分割されている場合には，LZSS 符号全体を，ハフマン圧縮された元の状態のままコンテクスト情報として保存する．ここで，上記の LZSS 符号はもともと，< 文字/一致長コードのハフマン符号，戻り距離コードのハフマン符号 > と符号化されていることか

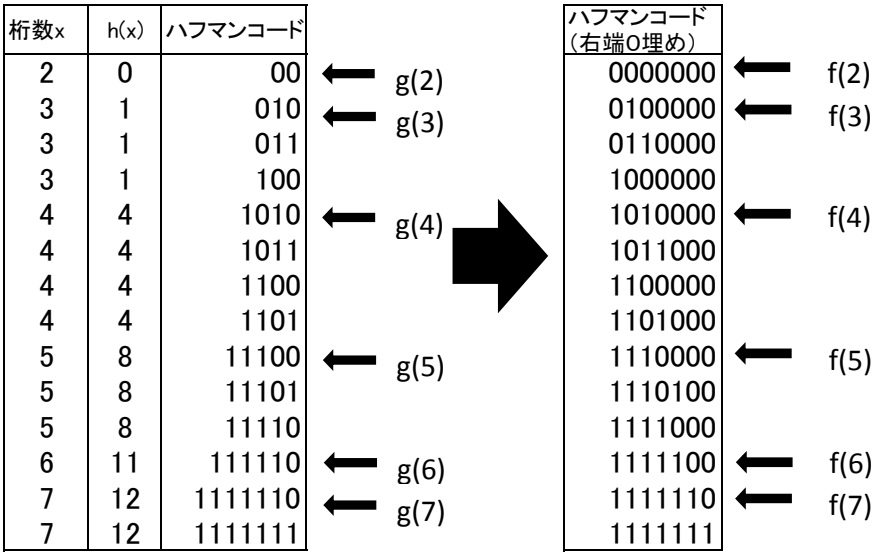


図 4.7: パラレルデコーディングの改良アルゴリズム

ら，最大 30bit が保存される．入力データに前パケットの末尾データが加えられることで，復号モジュールはただ文字/一致長コードのハフマン符号から復号すればよい．以降では，このデータに対してハフマン復号，LZSS 復号を行っていく．

ハフマン符号の復号:

復号処理は，まずハフマン符号の復号から行われる．ハフマン符号の最大長は 15bit であることから，15bit に切り出された入力データに対して符号表との一致を検索し，対応するコードに変換することで復号がなされる．3.2.2 項で紹介したパラレルデコーディングは，ハフマン符号の復号を高速化した研究である．パラレルデコーディングにより，2cycle で 1 ハフマン符号を復号可能であるが，この手法は更に改良が可能である．そこで，パラレルデコーディングの改良による，高速かつ省メモリコストなハフマン符号の復号手法を提案する．

まず，復号しようとするビット列（15 ビット）を b_{max}，符号長 x における最小のハフマン符号を $g(x)$，$g(x)$ の右端を 0 埋めしたものを $f(x)$ とする．$f(x)$ の具体例を図 4.7 に示した．また，$g(x)$ の符号表の先頭からの距離を $h(x)$ とする．

このとき，i を整数として，

$$f(i) \leq b_{max} < f(i+1) \tag{4.1}$$

が成り立つならば，b_{max} の先頭 i ビットがハフマン符号として存在する．例えば，先ほどの図 4.7 において $b_{max} = 1011101$ とすると，$f(4) \leq b_{max} < f(5)$ が成り立つ．ここで，b_{max} の先頭 i ビットを $b(i)$ と表すと，$b(4) = 1011$ はハフマン符号として存在していることが証明できる．このとき，

$$y = b(i) - g(i) + h(i) \tag{4.2}$$

と新たにマッピングすると，y の値は以下に従いただ一つの値が決定される．

- 符号表を用いて文字/一致長コードを復号する場合は 0〜285 の値
- 符号表を用いて戻り距離コードを復号する場合は 0〜29 の値

従って，y の値をインデックスとして文字/一致長，戻り距離コードを格納したメモリを探索すれば，一意に所望のコードを得ることができる．

ここで，符号表として必要になるメモリ量は，まず文字/一致長コードの符号表として，各符号長の最短ハフマン符号とその符号の符号表内での順番の組 (15bit+9bit) × 15 = 45Byte と，文字/一致長コードをハフマン符号の小さい順に並べた 9bit×286 = 322Byte の合わせて 367Byte となる．更に，戻り距離コードの符号表も同様に計算すると，(15bit+5bit) × 15 + 5bit×30 = 56Byte となる．これは，4.2.4 項の符号表作成モジュールで述べた，文字/一致長コードの符号表 858Byte，戻り距離コードの符号表 75Byte に比べ，半分以下の値となる．従来の一般的なハフマン復号とパラレルデコーディング，本研究で提案した改良手法の概要を図 4.8 で比較した．改良したパラレルデコーディングでは，1cycle によりハフマン符号が復号可能となる．

LZSS 符号の復号:

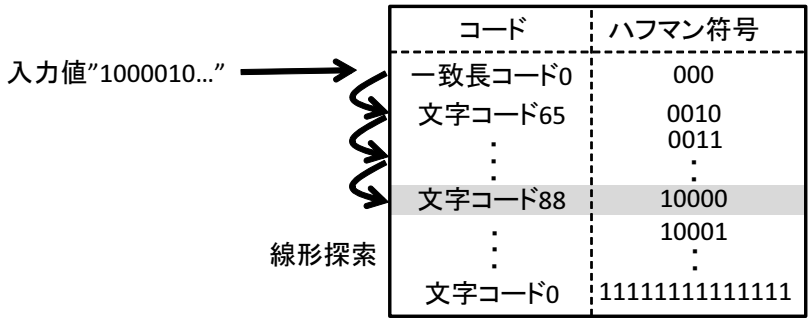
ハフマン符号の復号後，次に LZSS 符号が復号される．LZSS 符号の復号アルゴリズムは比較的容易である．まず，先頭の 9bit が読み出される．この符号が 0 から 255 の間であった場合は，その値を文字コードとして文字が復元される．また，符号が 256 から 286 であった場合は，符号が一致長コードであるとして，表 3.13 をもとに，後続 bit が読み取られる．この後続 bit の値により，一致長が決定される．更に，符号が一致長コードであった場合は，戻り距離コードが続くため，5bit が読み取られる．戻り距離コードと表 3.14 をもとに，後続 bit が読み取られる．上記の操作によって，一致長コードと戻り距離コードが得られる．

得られた 2 つのコードから，辞書内データの探索を行う．辞書はコンテクスト情報から取り出され，32KB の FIFO 形式のリングバッファに格納される．このバッファに対して，戻り距離からアドレスを計算し，一致長 Byte のデータを読み出すことで，LZSS 符号が復元される．そして，復号された文字は出力されると同時に，バッファ先頭にも格納され，随時辞書が更新される．

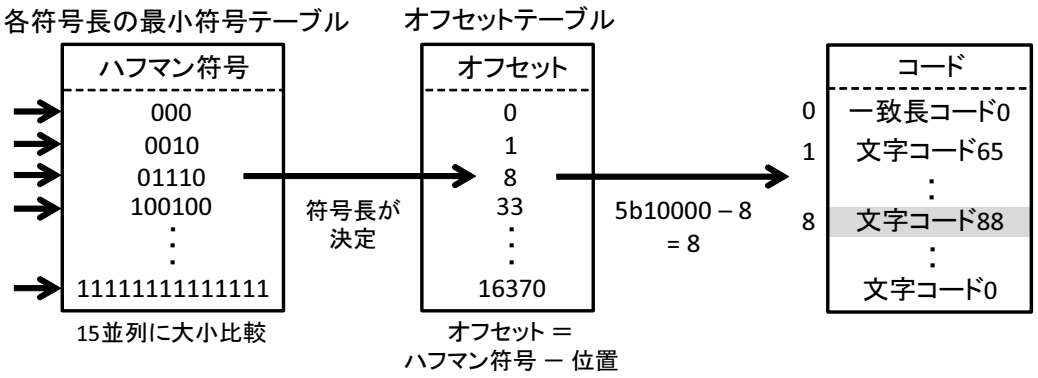
本処理をデータ末尾まで繰り返すことで，LZSS 符号の復号が完了される．なお，データ末尾のコードがフラグメンテーションの影響で復号できない場合，先に述べたように，復号できない LZSS 符号全体のハフマン符号がコンテクスト情報として保存される．このために，データ末尾の 30bit は最初に蓄えておく．全ての処理を終えた後，更新された辞書とデータ末尾のハフマン符号は，メモリコントローラへと転送され，コンテクスト情報が更新される．

4.2.6 コンテクストメモリ

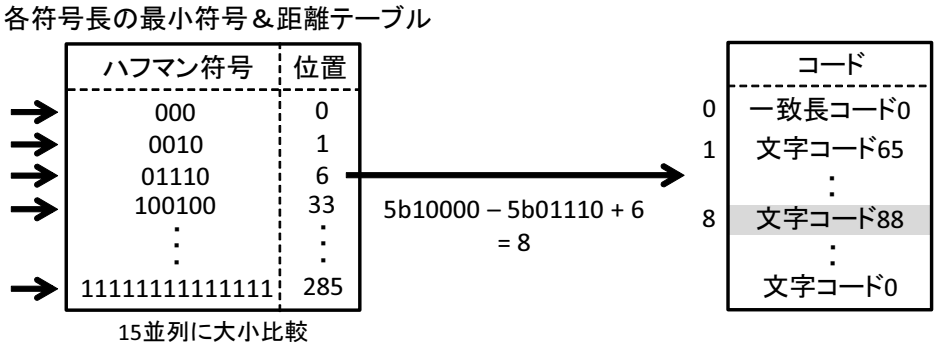
コンテクストメモリにはストリーム毎のコンテクスト情報が格納される．1 パケットの GZIP 展開において，コンテクストメモリへのアクセスは最低 3 回行われる．まず，構文解析モジュールによるコンテクスト情報の検索がある．この時は，タグにヒットしたかどうかのみが判断され，実



（a） 従来方式



（b） パラレルデコーディング方式



（c） 本研究の改良方式

図 4.8: ハフマン符号の復号方法の比較

際のコンテクスト情報読み出しは行われない．加えて，この時コンテクスト情報が存在しなかった場合には，空のコンテクスト情報によるエントリが挿入される．次に，復号モジュールによるコンテクスト情報の読み出しがある．ここでは，実際にコンテクスト情報が読み出され，復号モジュールへと転送される．そして，復号モジュールから転送される，更新されたコンテクスト情報の書き込みがある．更に，パケットがストリームの最初である場合には，符号表作成モジュールから転送される，符号表の書き込みがある．

従来，zlib などを用いた GZIP 展開では，1 圧縮データの展開に 64KB のメモリ資源を必要とした．このことから，石田らも論文 [2] で，GZIP 展開機構のコンテクスト情報に 1 ストリームあたり 64KB 程度のメモリを確保していた．しかしながら，本研究では，後述するパラレルデコーディングの改良手法により符号表として 423Byte 程度，辞書として 32KB，展開できなかったパケット末尾のデータとして 30bit のみでコンテクスト情報が管理できる．ここで，論文 [2] を参考にすると，10Gbps トラフィックを束ねた仮想 40Gbps トラフィックにおいて，ある瞬間の同時接続ストリーム数は，60 秒のタイムアウト値を設定した場合に平均 11 万ストリーム程度である．上記をもとに，100Gbps 環境における GZIP 圧縮ストリームの同時接続数を，トラフィックに対する GZIP データの割合 5.5%を用いて予測すると，1.5 万ストリーム程度となる．100Gbps 環境において必要となるコンテクストメモリのサイズは (423Byte+32KByte+30bit) × 15,000 となり，470MB 程度であると計算でき，オフチップの DRAM に十分格納できるデータサイズであることがわかる．

一方で，コンテクスト情報はデータサイズが大きく，また DRAM のアクセス遅延の影響を受け，読み出しに膨大な時間がかかることで GZIP 展開処理のボトルネックとなりうる．そこで，本論文では，コンテクストメモリではなく，まずキャッシュからコンテクスト情報を読み出すことで読み出しを高速化するコンテクストキャッシュ機構を提案している．

4.2.7 コンテクストキャッシュ

コンテクストキャッシュは，小容量だが高速なキャッシュメモリに，頻繁に参照されるコンテクスト情報をコピーしておくことで，キャッシュ参照によって高速にコンテクスト情報を読み出す機構である．ネットワークには，同一のストリームに属するパケットが短時間に連続して訪れるという時間的局所性が存在することが知られている [77]．これは即ち，コンテクスト情報の参照に局所性があることを示しており，キャッシュによる高速化が期待できる．そこで，本論文では，128 個のコンテクスト情報をダイレクトマップ方式により格納したキャッシュを，8 個並列に配置する．128 個のコンテクスト情報によるメモリ使用量は 4MB 程度であり，近年のプロセッサの L3 キャッシュと同程度の小規模メモリにより実装可能である．キャッシュを並列に配置することで，コンテクスト情報の読み出しを高速化するだけでなく，復号モジュールの並列化によるコンテクストメモリへのアクセス競合を解消することが可能になると考えられる．

キャッシュへのアクセスは，ハッシュモジュールで計算されたハッシュ値をもとに行う．ストリーム ID となる 5 タプルは全体で 104bit あり，これをそのままメモリアドレスに用いることは難しい．そこで，5 タプルのハッシュ値を用いて適切にアドレシングを行う．このハッシュ値は，前述したハッシュモジュールにおいて，104bit の 5 タプルを入力とすることで，15bit のハッシュ値として得られる．ハッシュ値の，下位 7bit をキャッシュインデックスとして用い，続く 3bit を

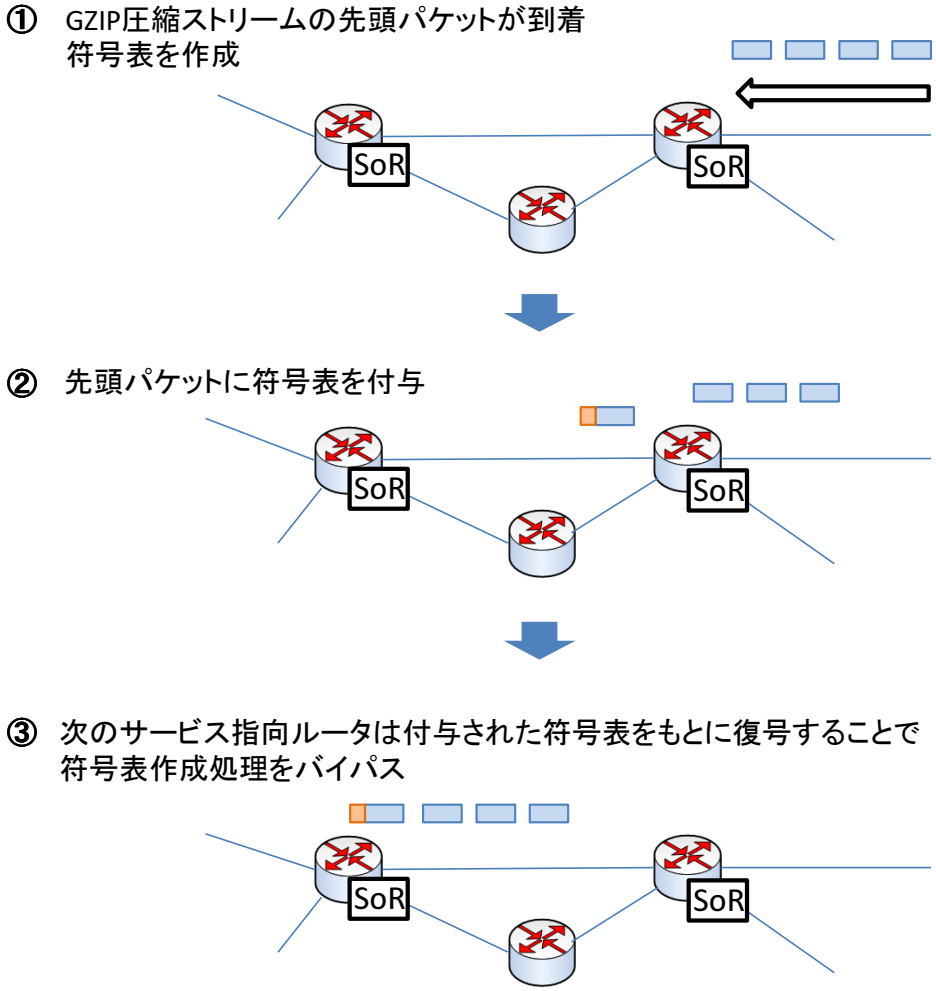


図 4.9: ピギーバックパケットを用いた符号表転送の概要

8 個のキャッシュへの割り当てに用いる．また，ハッシュ値の上位 8bit は，キャッシュタグとして用いる．

4.2.8 ピギーバックパケットを用いたルータ間での GZIP 展開高速化

アプリケーションルータが複数台ネットワークに存在する場合，一つの GZIP 圧縮パケットが異なるアプリケーションルータで何度も展開されると予想される．この時，各アプリケーションルータは同じ GZIP 展開処理を繰り返すため，ルータ毎に同じ符号表を作成しなければならない．そこで，アプリケーションルータ間において GZIP 復号処理を高速化する手法を提案する．

ルータ間で同じ符号表が利用されることから，前のアプリケーションルータにより作成された符号表を次のアプリケーションルータが使用することで，符号表の作成処理を省略することが可能になると考えられる．この際，ルータ間での符号表の転送にピギーバックパケットを用いることを想定する．本手法の概要を図 4.9 に示す．

本手法ではまず，アプリケーションルータ同士が互いの位置を事前に知る必要がある．これはOSPF[129] といったルータ間情報交換アルゴリズムを拡張し，経路上に他のアプリケーションルータが存在する場合にルーティングテーブルに 1bit のピギーバックフラグを設けることで可能になる．ピギーバックパケット送信側のルータ内部では，このフラグ情報を用いて，次のルータに向けて符号表をピギーバックするかどうかを判別する．符号表をピギーバックする場合，符号表作成モジュールで作成された符号表は，図 4.2 に示した Code Table Transmit モジュールより転送され，パケットにピギーバックされる．受け取り側のアプリケーションルータ内部では，まず構文解析モジュールにおいて，パケットに符号表がピギーバックされているか識別を行う．ここで，ピギーバックが確認された場合には，符号表作成モジュールをバイパスし，持っている符号表を用いて復号処理を開始する．結果として，GZIP 展開処理を高速化できる．後続の転送経路にアプリケーションルータが存在しない場合は，そのルータ以降ピギーバックされた符号表の送信を停止する．本手法は，完成された符号表をパケットに添付することでパケットサイズが増加するため，ジャンボフレームとしてパケットを処理するが，パケットサイズに制限がある場合は専用のパケットを別途生成することで対応する．

4.3 GZIP 展開シミュレータによる評価

本節では，前節で説明した高速な GZIP 逐次展開ハードウェアのシミュレータを作成し，スループットや回路規模に関して評価した結果を示す．本シミュレーションはネットワーク経路上でのトラフィックの GZIP 展開を想定し，実際のネットワークトレースを読み込むことで GZIP 展開シミュレーションを行った．

4.3.1 シミュレーション環境

シミュレーションでは，図 4.2 に示したアーキテクチャをハードウェア記述言語 Verilog HDL を用いて実装し，実際にネットワークトラフィックを流すことで，パケットの GZIP 展開をシミュレートした．ここで，実装に用いたツールを表 4.2 に表す. 出力の確認を行うために Cadence NC-Verilog LDV5.7 を，Verilog ファイルの論理合成に Synopsys Design Compiler 2005.09 を, ロジックのライブラリとして Free PDK 45nm の ASIC ライブラリ [130] を用いている．

また，ネットワークトラフィックとして，表 4.3 に詳細を示した PCAP 形式のトレースファイルが外部プログラムから読み込まれる．本トラフィックトレースは，慶應義塾大学理工学部西研究室内の 10Gbps ネットワーク内において採取されたトラフィックである．読み込まれたトレースから，Verilog Procedural Interface を用いて，パケットがハードウェアシミュレータの入力へと転送されている．ネットワークトラフィックにおける GZIP 圧縮データの大部分は，HTTP プロトコルの web ページアクセスにより生成されたストリームである．そのため，GZIP 圧縮ストリームを構成する平均パケット数は，5 パケット程度という小規模なストリームになったと考えられる．また，表 4.3 に示されている平均 GZIP 圧縮率 32%は，論文 [131] で GZIP 展開性能の評価に使われているワークロードの平均圧縮率が 31.55%であることを参考にすると，評価として妥当な値であると言える．

表 4.2: シミュレーションの実装環境

項目	ツール名
記述言語	Verilog-HDL
論理シミュレーション	Cadence NC-Verilog LDV5.7
ASIC 合成ツール	Synopsys Design Compiler X-2005.09
ASIC 合成用ライブラリ	FreePDK OSU Library (45nm)[130]
波形表示ツール	Cadence Simvision
PCAP 読み出しプログラム	C (Libpcap)
モジュールへのデータ転送	Verilog Procedural Interface

表 4.3: 評価用トラフィックの詳細

統計項目	平均値
パケット長	1,221.54byte
GZIP 展開後パケット長	3,778.43byte
GZIP 圧縮率	32.33 %
データ全体における GZIP 圧縮データの割合	5.50 %
1GZIP ストリームを構成するパケット数	5packet

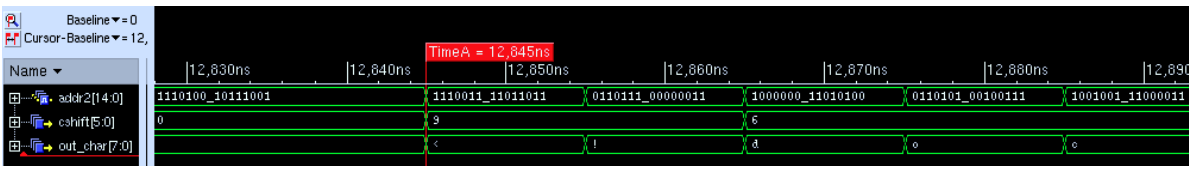


図 4.10: 1 文字ずつ復号しているときの復号モジュールの動作状況

シミュレーションで実際に 1 文字ずつ展開している様子を図 4.10 に示している．図では，7bit の out_char 線から，<!doctype> という文字列が出力される途中であることがわかる．

4.3.2 最も単純なハードウェア構成における評価

まず，最も単純な構成における GZIP 展開ハードウェアの回路規模およびスループットを算出した．ここで，最も単純な構成とは，キャッシュ，符号表作成モジュールおよび復号モジュールの並列化，符号表のピギーバックを全て適用せず実装した場合の GZIP 逐次展開ハードウェアのことである．表 4.4 に算出結果を示している．回路規模の算出は表 4.2 に示したツールを用いた論理合成によって行っている．また，スループットの計測はシミュレーション結果から，全データサイズを GZIP 展開にかかる時間で割って得た．

表 4.4: 最も単純なハードウェア構成における回路規模およびスループット

項目	回路規模またはスループット
復号表作成モジュールの回路規模	$0.057mm^2$
復号モジュールの回路規模	$0.016mm^2$
その他モジュールの回路規模	$0.011mm^2$
GZIP 展開スループット	2.82Gbps

最も単純なハードウェア構成では，本研究が目標としている 5.5Gbps のスループットが達成できないことがわかる．従って，提案手法を併用することで，高スループット化を行う必要がある．

4.3.3 モジュールの並列度に関する評価

本論文では，GZIP 展開における性能ボトルネックである符号表作成処理および復号処理に焦点を当て，それぞれのモジュールを並列化することで処理遅延を隠蔽するアーキテクチャを提案した．そこで，符号表作成モジュール数と復号モジュール数をそれぞれ変化させた場合のスループットおよび回路規模をシミュレーションにより算出した．これによって最適な符号表作成モジュール数と復号モジュール数の検討を行う．なお，本シミュレーションではコンテクストキャッシュおよびピギーバックパケットを適用していない．

シミュレーションでは，符号表作成モジュール数を n，復号モジュール数を m として，n と m を変化させた場合の GZIP 展開ハードウェアをそれぞれ実装し，その動作を比較した．まず，n を 1 として固定し，m を変化させた場合の動作を比較した結果を図 4.11 に示す．ここで，図の横軸はシミュレーションの経過時間を示し，図の各行は復号モジュールまたは符号表作成モジュールの出力信号を示す．また，各行の信号の色によって m の値が異なる．例えば，図の最上部の緑で示された 2 行は，$n = 1$，$m = 1$ の場合の動作を示しており，上段が復号モジュールの信号を，下段が符号表作成モジュールの信号を表す．更に，その下の黄色で示された 3 行は，$n = 1$，$m = 2$ の場合の動作を示しており，上段 2 行が復号モジュールの信号を，下段 1 行が符号表作成モジュールの信号を表す．図 4.11 では，m を 1 から 11 まで変化させ昇順に並べた．同様に，n を 2 として固定した場合の，動作の比較結果を図 4.12 に示す．

符号表作成モジュール数が 1，復号モジュール数が 7，8，9 の場合の動作状況において，符号表作成モジュールの動作に注目すると，復号モジュール数が 7 の場合は他の場合に比べ，符号表作成モジュールが動作していない期間が存在する．これは，復号処理がボトルネックとなって，符号表作成モジュールにアイドル時間が発生していることを示しており，最適な設計であるとはいえない．一方で，復号モジュールの動作に着目すると，復号モジュール数が 9 の場合は他の場合に比べ，復号モジュールが動作していないタイミングが多く存在する．したがってこれも最適な設計であるとはいえない．この中では，復号モジュール数が 8 の場合において，全てのモジュールが休むことなく動作しており，符号表作成処理速度と復号処理速度のマッチした最適な設計であるといえる．

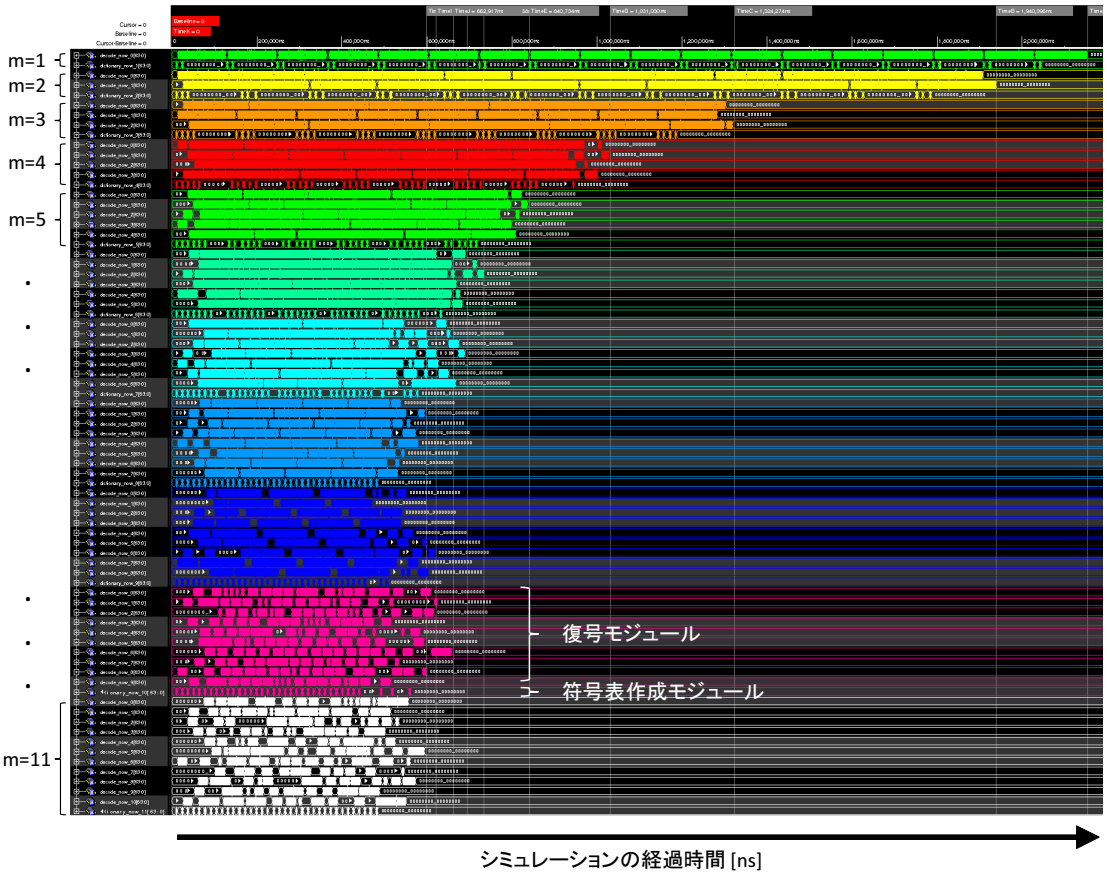


図 4.11: 復号モジュール数 m の変化における各モジュールの出力信号の比較（符号表作成モジュール数 n=1 の場合）

次に，以上の設計に対して全体の回路規模とスループットの測定を行った．符号表作成モジュール数が 1 の場合の結果を図 4.13 に，符号表作成モジュール数が 2 の場合の結果を図 4.14 に示す．

図 4.13 および図 4.14 では，符号表作成モジュールと復号モジュールの並列度を増やすことで，スループットが向上していくことがわかる．しかしながら，符号表作成モジュール数が 1 の場合には，復号モジュール数が 9 以降，すなわち 11Gbps 程度でスループットの向上が限界となっている．これは，上記の構成以降になると，復号処理よりも符号表作成処理のほうがボトルネックとなるためである．そこで，符号表作成モジュール数を 2 とした場合には，復号モジュール数を増やすことで，およそ 18Gbps までスループットが向上していくことがわかった．また，符号表作成モジュール数が 3 の場合には，25Gbps 程度が限界となる．

ここで，処理モジュール数の増加に伴って回路規模も増大していくため，実装コストが問題となる．そこで，図 4.15 には，スループットを回路規模で割って得られる値をパフォーマンス値として計測した結果を示している．

図 4.15 を見ると符号表作成モジュール数 1，復号モジュール数 8 の時がもっともパフォーマンスがよく，$58Gbps/mm^2$ である．このときの回路規模は $0.18mm^2$ である．この面積は，前述した 8Gbps での GZIP 展開を可能とする PowerEn[10] の回路面積に対して，およそ 1/12 の値である．

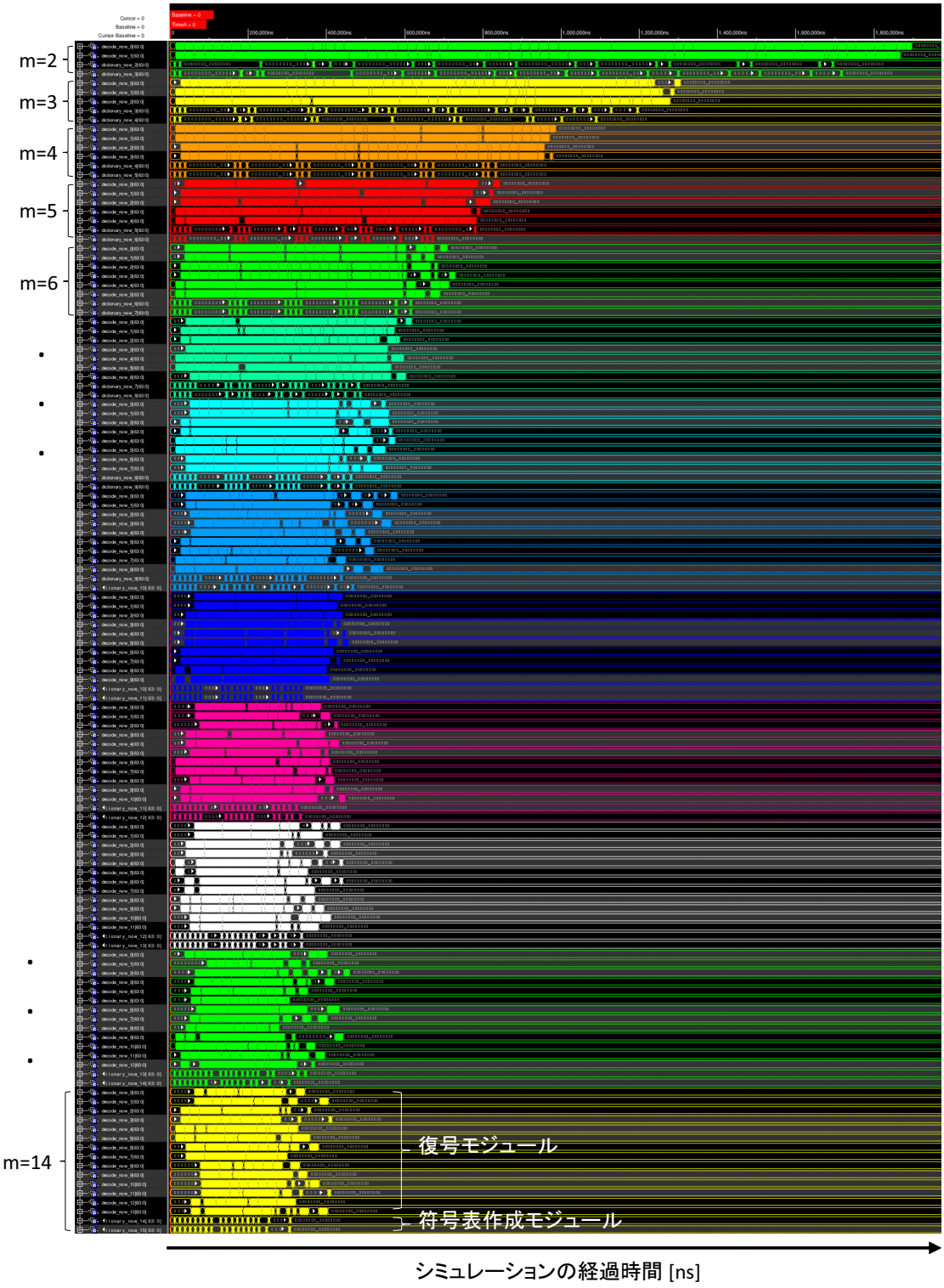


図 4.12: 復号モジュール数 m の変化における各モジュールの出力信号の比較（符号表作成モジュール数 n=2 の場合）

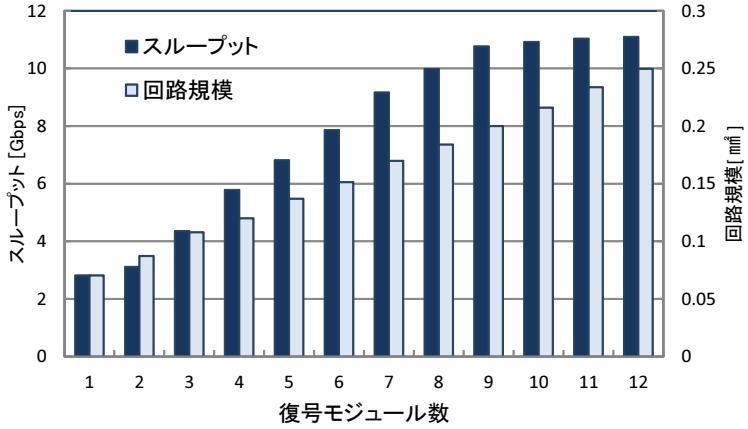


図 4.13: 符号表作成モジュール数 1 における復号モジュール数とスループット，回路規模の関係

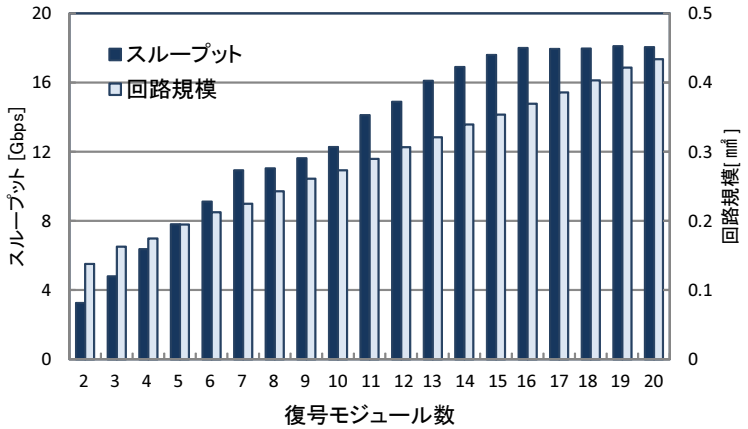


図 4.14: 符号表作成モジュール数 2 における復号モジュール数とスループット，回路規模の関係

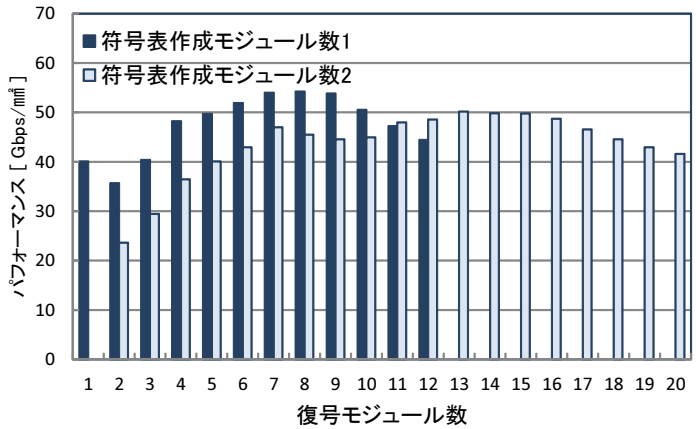


図 4.15: 各モジュール並列度によるパフォーマンスの測定結果

使用したライブラリの NAND2 ゲート面積からゲート数を算出すると，この面積は NAND 素子 22,556 個程度の比較的小規模な回路であると言える．また，このときの ASIC による論理合成結果では，最大動作遅延が 1.92ns であり，521MHz の周波数で動作させることが可能となっている．

トラフィックの違いによるスループットの差

本 GZIP 展開機構はルータ上での動作を想定している．そのため，トラフィックの傾向によって，処理性能に差がでることが考えられる．そこで，本研究ではストリームの到着順序に関してベストケースとワーストケースを想定し，それぞれのケースによる評価を行った．それぞれのケースにおけるパケットの到着順序を図 4.16 および以下に示している．なお，図中のアルファベットはストリーム ID を，番号はストリーム内のパケット番号を表している．

ベストケース

図 4.16(a) に示すように，パケット毎に異なるストリームのパケットが到着する．

ワーストケース

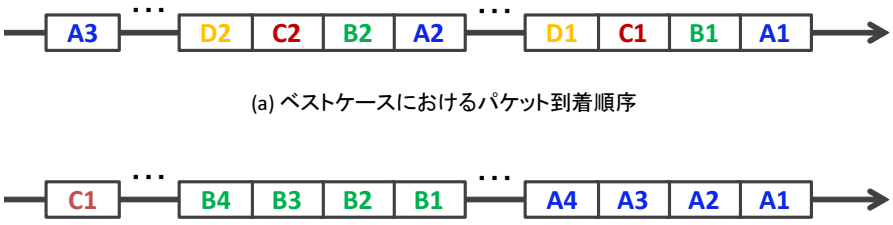
図 4.16(b) に示すように，同一ストリームのパケットがシーケンシャルに到着する．

両ケースに基いてパケット順序を入れ替えた，36 の GZIP 圧縮ストリームを用いて評価した結果を図 4.17 に示した．このワークロードは，前述した表 4.3 に示すトラフィックトレースから GZIP パケットを抽出し，GZIP パケットを任意の順序に入れ替えることで作成している．図 4.17 上部にベストケース時の動作状況を，下部にワーストケース時の動作状況を載せている．なお，本シミュレーションは符号表作成モジュール数 1，復号モジュール数 7 のハードウェア構成で行っている．このとき，ベストケースでは 60,384clock，ワーストケースでは 163,501clock が処理にかかっており，トラフィック傾向の差によって 2.7 倍程度の影響がスループットに現れることが示された．これは，ワーストケースのように，同ストリームのパケットが連続して到着する場合，前パケットの展開処理が終わるまで後続パケットがブロッキングされるためである．一方で，別ストリームのパケット同士はそれぞれ独立して処理が可能であるため，ベストケースの場合には本提案ハードウェアの並列性を活かし，高いスループットが得られた．

実際のネットワークでは，エッジネットワークであるほど同時接続ストリーム数が少ないためワーストケースに，コアネットワークであるほど同時接続ストリーム数が多いためベストケースに近い到着順序になることが予想される．従って、本提案ハードウェアは，高い処理スループットの求められるコアネットワークであるほど，処理の並列性を活かし，効果的に GZIP 展開処理を行うことができると考えられる．

4.3.4 コンテクストキャッシュに関する評価

GZIP 逐次展開ハードウェアのキャッシュ機構に関して，まずキャッシュヒット率の測定を行った．シミュレーションにおけるキャッシュ構成や評価環境，ヒット率の測定結果を表 4.5 に示した．シミュレーションでは，86%の高いキャッシュヒット率が得られている．このことから，ストリームのコンテクスト情報読み出しにおいてキャッシュは十分に有効であることが示された．



(a) ベストケースにおけるパケット到着順序

(b) ワーストケースにおけるパケット到着順序

図 4.16: 各ケースでのパケット到着順序の違い

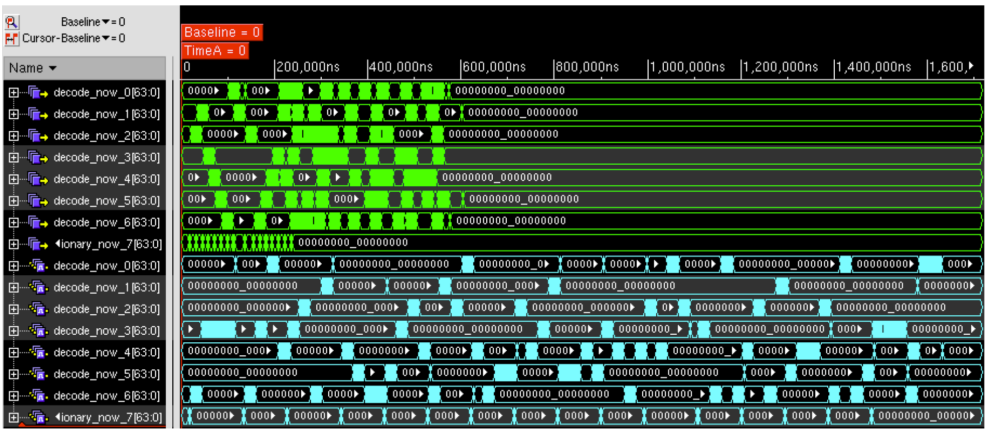


図 4.17: ベストケース，ワーストケースにおける展開状況の比較

次に，キャッシュを用いた場合と用いない場合の GZIP 展開スループットの差について比較した．本提案ハードウェアは，符号表作成モジュールおよび復号モジュールの並列数を増すほど，コンテクストメモリにおいて競合が発生すると予想される．従って，処理モジュールの並列数を増すほどキャッシュの効能は向上すると考えられる．そこで，処理モジュール数を変化させつつ，キャッシュを用いた場合と用いない場合でのシミュレーションを行った．スループットの測定結果を図 4.18 に，キャッシュによるスループットの向上割合を図 4.19 に示した．

図 4.18 よりわかるように，処理モジュール数が少ない場合にはコンテクストキャッシュの効果は小さい．これは，処理モジュール数が少ないため，コンテクスト読み出しの競合が発生しづらく，また，復号や符号表作成にかかる遅延に対してコンテクスト情報読み出しの遅延は小さいためであると考えられる．一方で，処理モジュール数が増加するに従って，コンテクストキャッシュによるスループットの向上が見られる．これは，処理モジュール数が増えることでコンテクスト情報読み出しの競合が頻繁に起こるようになったためであると考えられる．

本シミュレーションでは，図 4.19 に示したように，コンテクストキャッシュを用いることで最大 19.4%のスループット向上が見られた．本研究で目的とした，100Gbps ネットワークにおけるワイヤレートでの GZIP 展開に要する 5.5Gbps は符号表作成モジュール数 1，復号モジュール数 4 のハードウェア構成で達成できる．この場合のスループットは 6.37Gbps である．また，GZIP 圧

表 4.5: シミュレーションの環境およびキャッシュヒット率の測定結果

キャッシュのエントリ数	128
キャッシュのモジュール数	8
エントリマップ方式	ダイレクトマップ
GZIP 圧縮パケット数	38,488
キャッシュヒット率	86%

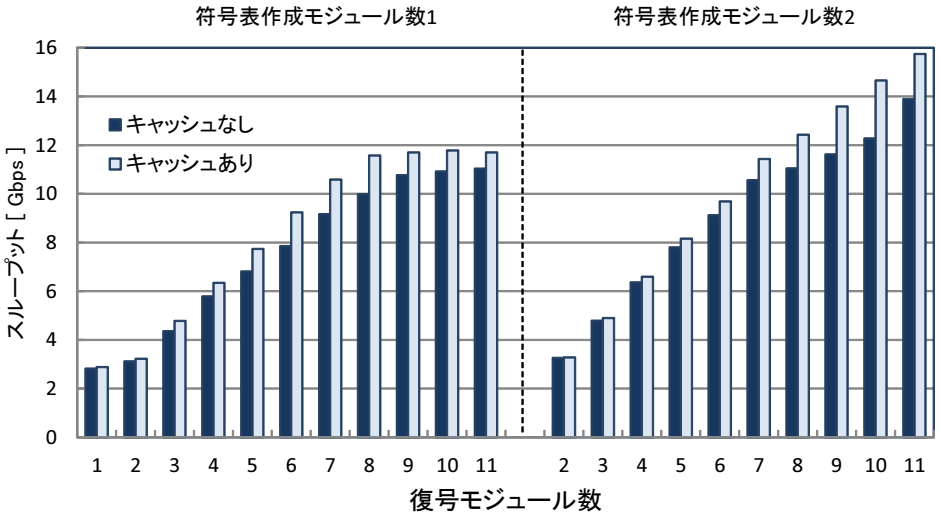


図 4.18: コンテクストキャッシュの有無によるスループットの比較

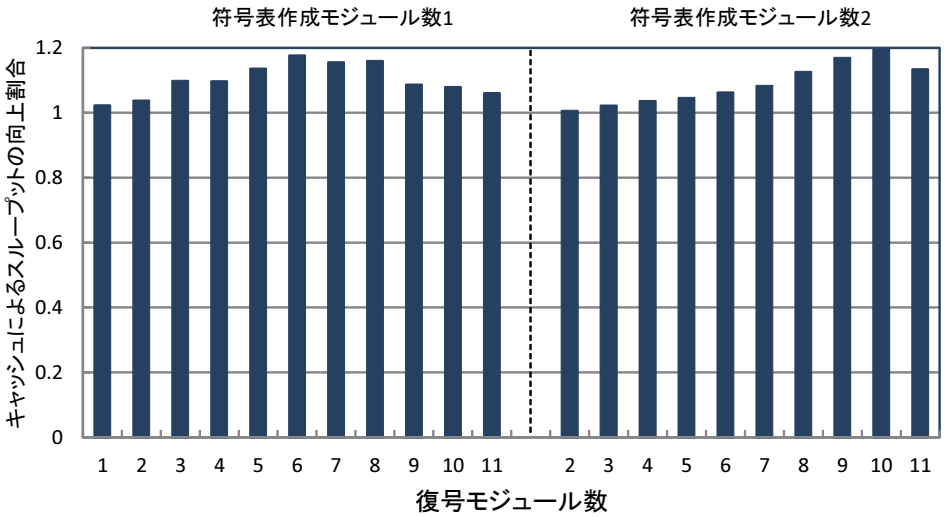


図 4.19: コンテクストキャッシュによるスループットの向上割合

縮トラフィックの普及を考慮して，トラフィックに占めるGZIP圧縮データの割合が現在の2倍の11%になることを想定しても，符号表作成モジュール数2，復号モジュール数8程度のハードウェア構成により11.0Gbpsが達成できることから，十分に処理可能であることが示された．

4.3.5 ピギーバックパケットに関する評価

アプリケーションルータが2台（アプリケーションルータ1，アプリケーションルータ2）存在し，アプリケーションルータ1で作成された符号表をパケットにピギーバックし，アプリケーションルータ2がそれを受け取る状況を想定する．このとき，アプリケーションルータ1において作成された符号表は，ストリームの先頭パケットのヘッダに付随させる．アプリケーションルータ2では添付された符号表を用いて，符号表作成プロセスを省き，展開を行う．

前項でのシミュレーション環境と同じ条件でシミュレーションを行った．ピギーバックパケットを用いた場合の動作状況を図4.20に，ピギーバックを用いない場合の動作状況を図4.21に示した．

ピギーバックパケットを用いない場合，GZIP展開処理全体で60,384clockかかったのに対し，ピギーバックパケットを用いた場合は45,814clockで処理を完了している．従って，およそ3/4のクロック数で処理を終えており，1.3倍程度のスループット向上が望める．本手法はスループット向上の観点からも有効な手法であるが，アプリケーションルータが複数台設置されたネットワークにおける解析遅延の削減に役立つ．

4.4 本章のまとめ

本章では，アプリケーションルータにおけるGZIP展開処理に焦点を当て，100GbpsネットワークでのGZIP展開処理を実現するGZIP展開機構を提案した．近年のトラフィックのGZIP圧縮データ率が5.5%程であることから，100Gbpsネットワーク下のパケット処理においては，5.5Gbps以上のGZIP展開処理スループットが要求される．

これに対し，まず，GZIP展開処理を高速化するハードウェアアーキテクチャを提案した．アプリケーションルータでの処理を想定し，本ハードウェアアーキテクチャにコンテクストスイッチ手法を実装することで，パケット毎の逐次なGZIP展開が可能となった．また，最も単純な構成にした場合の本ハードウェアアーキテクチャによって，2.82GbpsのGZIP展開処理スループットが得られることをシミュレーションにより示した．

次に，GZIP展開処理におけるストリーム毎の独立性を活かし，性能ボトルネックである復号処理および符号表作成処理を並列化するアーキテクチャを提案した．本アーキテクチャではモジュール並列度を増すことでスループットも向上し，符号表作成モジュール数を3，復号モジュール数を20とした場合には25Gbps程度のGZIP展開処理スループットが得られることを示した．

また，本アーキテクチャと併せて，コンテクスト情報の読み出し遅延を隠蔽するためのコンテクストキャッシュを提案した．コンテクスト情報は1ストリームあたり33KB程度とデータサイズが大きいため，読み出し遅延が高く，加えて処理並列数を増すほどメモリ読み出しに競合が発生する．そこで，128エントリ程度の小規模なコンテクストキャッシュを複数配置することで，コン

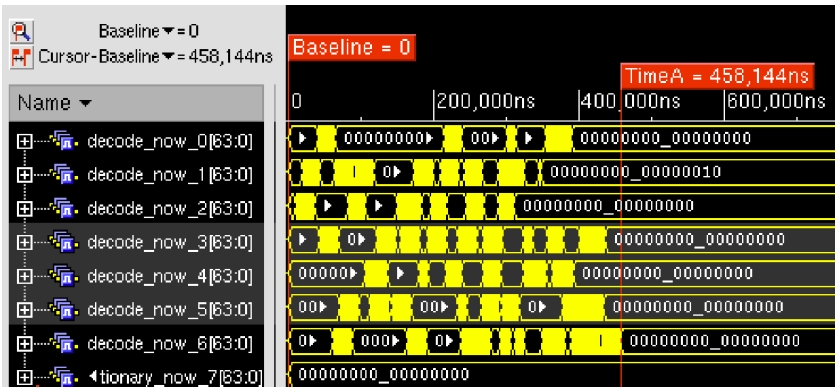


図 4.20: ピギーバックパケットを用いた場合の各モジュールの動作状況

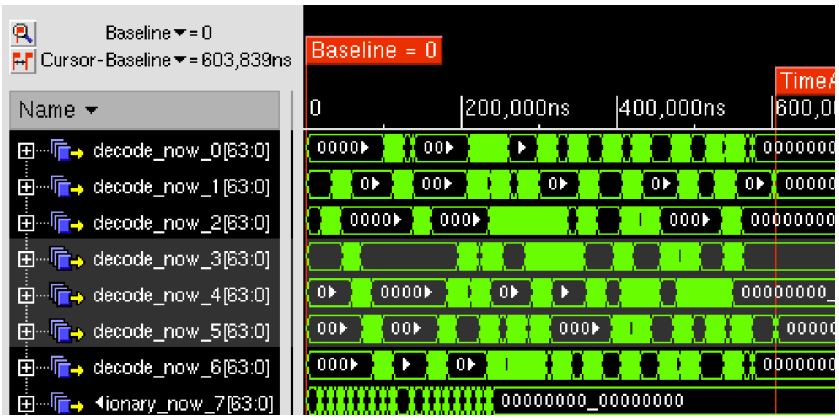


図 4.21: ピギーバックパケットを用いない場合の各モジュールの動作状況

テクスト情報の読み出し遅延を隠蔽した．シミュレーションでは，モジュール並列数を増すほどコンテクストキャッシュによるスループット向上効果も向上し，符号表作成モジュール数 1，復号モジュール数 4 のハードウェア構成の場合に 6.37Gbps が達成できることを示した．今後の GZIP 圧縮の普及を想定し，トラフィックに占める GZIP 圧縮データの割合が現在の 2 倍の 11%になったとしても，符号表作成モジュール数 2，復号モジュール数 8 のハードウェア構成により 11Gbps が達成できる．

更に，本論文では，複数台のアプリケーションルータがネットワーク上に存在する場合に，ピギーバックパケットを用いることで GZIP 展開処理を高速化する手法を提案した．このような状況下では，複数台のアプリケーションルータにおいて，GZIP 展開処理が重複することが考えられる．そこで，OSPF といったルータ間情報交換プロトコルを用い，GZIP 圧縮パケットの転送経路上にアプリケーションルータが存在する場合には，GZIP 展開処理で作成した符号表をパケットにピギーバックし転送する．符号表のピギーバックされたパケットは，添付された符号表を用いることで符号表作成処理をバイパスし，復号処理を行うことが可能となる．シミュレーションでは，2 台のアプリケーションルータがあるネットワークを想定し，ピギーバック手法を用いた場合にスループットが 1.3 倍程度向上できることを示した．

第5章　抽出ルールに基づいた文字列探索処理のオフロード

5.1　本研究の動機

アプリケーションルータにおいて，情報抽出機構の中核となるのが文字列探索処理である．従来，様々な文字列探索処理の高速化手法が研究されていることを第 3 章で述べたが，それらの手法により得られる文字列探索スループットは 7Gbps 程度であった．これに対し，本論文が目的とするのは，100Gbps コアネットワークにおける情報抽出の実現である．

アプリケーションルータは，実際には 100Gbps の文字列探索処理性能を有する必要はない．なぜならば，アプリケーションルータにおける情報抽出では，対象となるパケットが抽出ルールによって指定される場合が多く，トラフィック全体に対して情報抽出を行う必要がないためである．そこで，本章では，抽出ルールに基いて文字列探索処理負荷をオフロードするハードウェアアーキテクチャを提案し，文字列探索処理において 100Gbps の実効スループット獲得する．

5.2　アプリケーションルータに向けた文字列探索アーキテクチャ

文字列探索において，入力文字列に対する完全一致検索処理は，処理回数が膨大であることからスループットの向上が困難であった．アプリケーションルータにおける文字列探索処理ではパターンが動的に追加，削除されることから，完全一致検索処理を回路化し並列に処理させることも困難である．

第 3 章では，これに対して文字列探索の高速化や処理負荷削減を目的とした様々な既存研究があることを説明した．Boyer-Moore 法は，文字列探索処理における入力文字列のシフト量を増加させることで，パターン探索における完全一致検索回数を削減できるが，パターン数に対して線形に計算，実装コストが増加するため，アプリケーションルータへの実装には向いていない．Aho-Corasick 法は，複数パターンを一つのオートマトンにまとめることで，パターン数によらない一致探索処理が可能となるが，既存研究において達成されるスループットは 7Gbps 程度であり，100Gbps ネットワークの文字列探索には到底対応できない．そこで，Rabin-Karp 法による文字列探索処理負荷の削減を紹介した．

Rabin-Karp 法は文字列の完全一致検索の前に，簡易なハッシュ値の比較を行うことで，完全一致検索処理をオフロードすることができる．更に Rabin-Karp 法は，ハッシュ値をメモリのアドレスとして用いることで，全パターンを一度に検索できるため，パターン数によらないパターン探索が可能である．アプリケーションルータでは，例えば NIDS アプリケーションに対応するためには数千以上のパターンに対してマッチングを行う必要がある．このような目的から，アプリケー

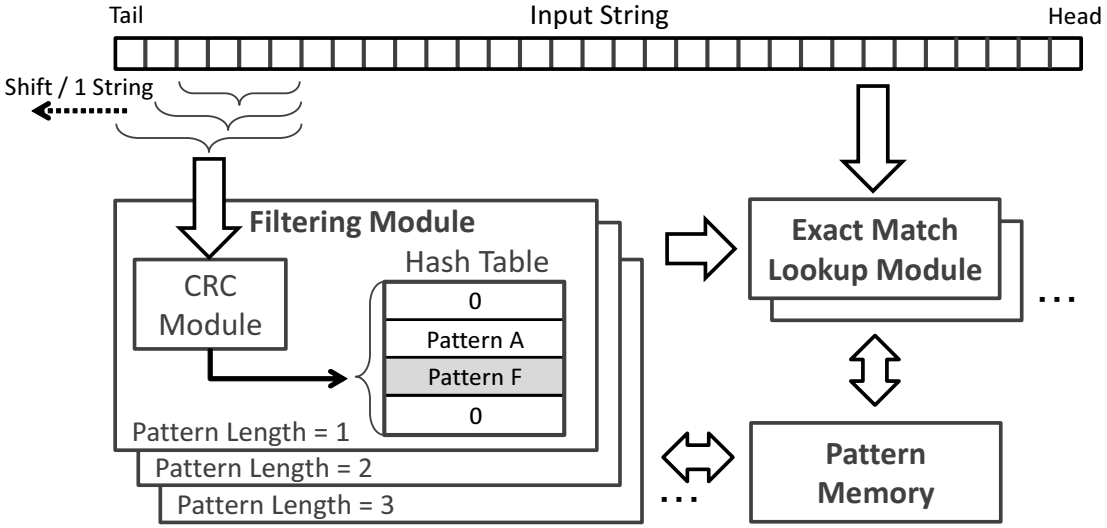


図 5.1: Rabin-Karp 法のアーキテクチャ

ションルータにおける文字列探索は，まず Rabin-Karp 法によって全抽出ルールの探索負荷を削減した後，オフロードされた入力に対して完全一致を検索することが望ましい．そこで，本論文では Rabin-Karp 法をベースとした，アプリケーションルータにおける文字列探索処理機構を検討する．

アプリケーションルータにおける Rabin-Karp 法の実装は，処理速度の観点から，処理モジュールをハードウェア化することが望ましい．そこで，Rabin-Karp 法のアーキテクチャを図 5.1 に示した．本アーキテクチャの構造はフィルタリングモジュールと完全一致検索モジュールの 2 つにわけることができる．フィルタリングモジュールでは入力文字列に対してハッシュ値が計算され，それをもとにハッシュテーブルにアクセスすることで，同じハッシュ値を持つパターンが検索される．ここで，ハッシュ値の一致がなかった場合には，当該入力文字列の完全一致検索処理を省略できる．一方で，同じハッシュ値を持つパターンが存在した場合は，完全一致検索モジュールにおいてそのパターンとの照合を行う．完全一致検索モジュールは，既存研究における完全一致検索高速化手法をそのまま適用できる．例えば，本モジュールを図 5.1 のように並列化することで，探索速度の向上が期待できる．

なお，図 5.1 に示したように，Rabin-Karp 法のハードウェア実装におけるハッシュ関数は，第 4 章で紹介したローリングハッシュよりも CRC を用いたほうが効率が良い．ローリングハッシュは 2 回目以降の計算量こそ少なくなるが，ハードウェア実装においてその恩恵は少なく，また，値の大きい乗算，除算を行うことからハードウェアでの実装には向いていない．更に，本アーキテクチャではハッシュ値をテーブルのインデックスとして用いるために bit 数を指定できること，ハードウェア実装において高速に動作することが求められることからも CRC が向いていると言える．

Rabin-Karp 法のハードウェア実装では，パターン長に応じた数のフィルタリングモジュールを実装する必要があり，ハードウェアコストが増大する．前述してきたように，アプリケーションルータでは数千を超えるパターンが想定されるため，多くのパターン長が混在すると考えられ，このような膨大なパターン長に対応するためには膨大なハードウェアコストを要する．そこで，ま

ず，Rabin-Karp 法の低実装コストなハードウェアアーキテクチャを提案する．

次に，Rabin-Karp 法によるスループットについて検討する．本手法の処理性能はハッシュ値がマッチする割合に左右される．第 4 章で述べたように，近年の文字列探索手法の研究では，完全一致検索処理において 7Gbps 程度のスループットが得られるとすると，100Gbps ネットワーク下でワイヤレートに文字列探索を行うには，Rabin-Karp 法によって 7%程度まで完全一致検索の処理負荷を削減する必要がある．そこで，本論文では，アプリケーションルータにおける抽出ルールの特性にもとづいたフィルタリング機構を追加することで，ハッシュマッチ後に更に完全一致検索処理負荷をオフロードする手法を提案する．ここで，アプリケーションルータにおける抽出ルールとしてはパターンが多様である NIDS を想定し，NIDS アプリケーション Snort におけるルールセットを参考にすることで，ネットワーク経路上での情報抽出に最適なアーキテクチャを検討した．

5.3 ハードウェアコストの削減手法

前節で述べたように，アプリケーションルータのパターン数は膨大となることが考えられる．これに対して，Rabin-Karp 法のハードウェア実装はパターン長の数だけハッシュモジュール，ハッシュテーブルを用意する必要があり，ハードウェアコストが膨大となる．そこで，ハードウェアコストを削減するための 2 つの手法を提案する．

5.3.1 case-insensitive アプローチ

文字列探索において大文字と小文字を区別しないことを case-insensitive，大文字と小文字を区別することを case-sensitive と呼ぶ．両パターンを Rabin-Karp 法で区別する場合，それぞれに対応したハッシュモジュール及びハッシュテーブルが必要となる．これは，例えば入力文字列が “String” であった時，case-insensitive なパターンに対しては “string”として，case-sensitive なパターンに対しては “String”としてハッシュ化する必要があるためである．このことから，フィルタリングモジュールのハードウェアコストが増大する．

しかしながら，Snort においてデフォルトで用いられるルールセットである snortrules-snapshot-2.8 を分析したところ，全体の 81%には case-insensitive が指定されていることがわかった．従って，case-sensitive パターンは抽出ルールにおいて支配的ではない．また，case-sensitive パターンは，ハッシュマッチングを case-insensitive で行った後，後段の完全一致検索において対応することも十分可能である．これを本論文では，case-insensitive アプローチと呼ぶ．図 5.2 では case-insensitive アプローチの処理を 3 種の例により示している．例えば，case-sensitive なパターン “Pattern”に対して入力文字列が “Puttern”である場合には，まず “pattern”と “puttern”のハッシュ値を比較する．ここで，ハッシュ値は一致しないため，入力文字列を完全一致検索モジュールへ送る必要はない．入力文字列が “PatTern”である場合は，case-insensitive な入力文字列 “pattern”と抽出パターン “pattern” のハッシュ値が一致するため，入力文字列 “PatTern”は完全一致検索処理に送られる．しかしながら，完全一致検索の結果，“PatTern”と “Pattern”は一致しないため，“PatTern”は “Pattern”ではないことが判別する．入力文字列が “Pattern”である場合は，case-insensitive なハッシュ値 “pattern”と

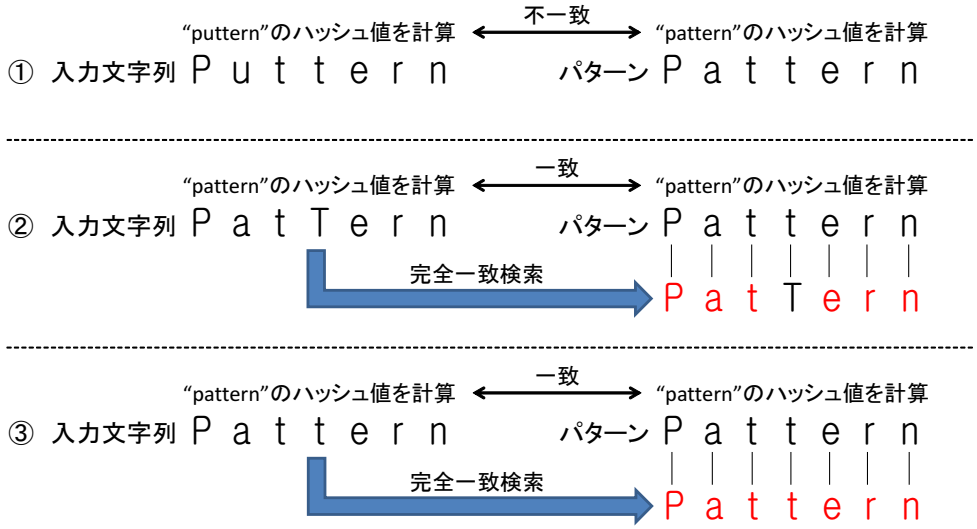


図 5.2: case-insensitive アプローチの例

“pattern”は一致し，なおかつ後段の完全一致検索でも一致するため，入力文字列 “Pattern”はパターンに一致することが判別する．

そこで，本論文では，ハッシュマッチングを全パターン case-insensitive で行う case-insensitive アプローチを提案する．これによって，Rabin-Karp 法の実装に必要となるハッシュ関連モジュールを 1/2 に削減できる．

5.3.2 文字長制限アプローチ

文字長制限アプローチでは，実装するフィルタリングモジュール数を文字長で制限する．従来手法ではフィルタリングモジュールはパターン長の数だけ実装しなければならなかったため，パターン長を制限して実装することでハードウェアコストが削減できる．

Snort で用いられる ASCII 文字列のパターンにおいて，長い文字列は 100 文字に達する．従来は，このような 100 文字のパターンであったら，100 文字全体のハッシュ値を計算し比較していた．このため，パターン長に応じたフィルタリングモジュールが必要であった．しかしながら，Rabin-Karp 法の特性を考慮すると，必ずしもパターン全体をハッシュ化する必要はない．Rabin-Karp 法では，入力文字列がパターンと一致する可能性を検知できればよいのであって，先頭数文字の比較であっても十分に効果は発揮されると考えられる．

図 5.3 は，テキサス大学の Lo らが Snort ルールセットに含まれるパターン長の分布を測定したグラフである [13]．ここで，Lo らは，Snort ルールセットとして 2006 年 12 月 15 日に Snort 公式サイトよりリリースされたルールセットを用いている．Lo らの測定によると，文字長の長いパターンが抽出ルールに占める割合は少ない．多くのパターンは 5 文字程度の短いパターンであることがわかる．そこで，文字長制限アプローチでは，パターンのハッシュ化を数文字に制限する．その文字数を超える文字長のパターンに対しては，先頭からその文字数目までのハッシュ値を用い

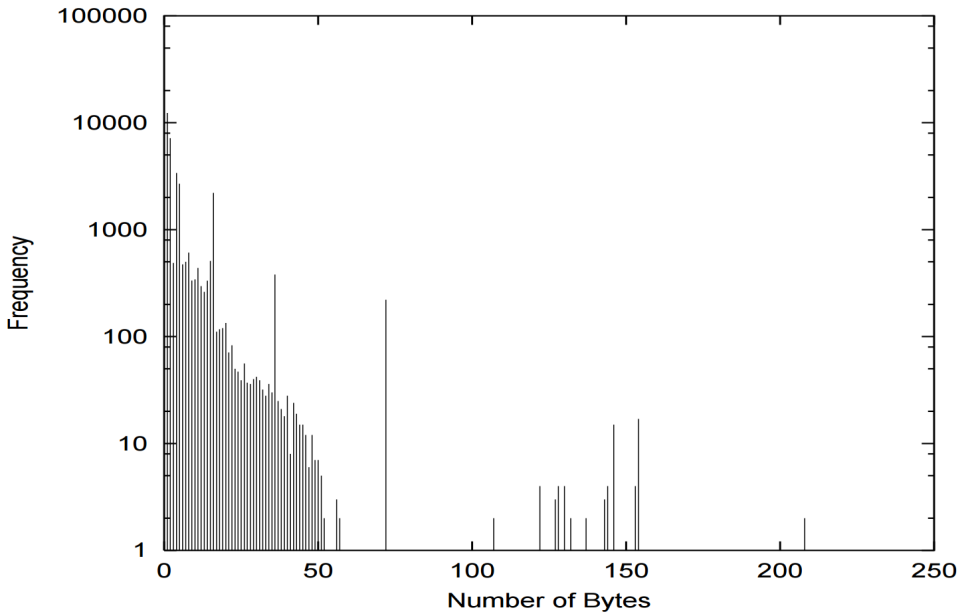


図 5.3: Snort ルールのパターン長分布 [13]

てフィルタリングを行う．本手法によって，実装すべきフィルタリングモジュールはたった数文字分まで削減される．

5.4 完全一致検索処理負荷の削減

本節では，Rabin-Karp 法をもとに，更に完全一致検索の処理負荷を削減する手法を検討する．そこで，抽出ルールで用いられる指定オプションに着目した．3.2.3 項の図 3.35 に示したように，抽出ルールには探索するパケットを指定するオプション（Snort ではルールヘッダ）が設けられることが多い．これは Snort に限らず，アプリケーションルータにおける多くのサービスの抽出ルールで適用されると考えられる．従来の様々なハードウェア文字列探索アーキテクチャでは，ルールによってオプションが異なることから，このようなオプションまで考慮することは困難であった．しかしながら，Rabin-Karp 法を改善し，ハッシュテーブルにこれらのオプション情報を付与することで，ハッシュマッチングした文字列に対し更にオフロードを行うことが可能となる．本論文では，このような手法として，宛先 IP アドレス指定および宛先ポート番号指定を用いたフィルタリング手法と，探索位置指定を用いたフィルタリング手法を提案した．提案手法のアーキテクチャを図 5.4 に示した．

5.4.1 宛先 IP アドレスおよび宛先ポート番号を用いたフィルタリング手法

抽出ルールのオプションとして最も用いられている項目が宛先 IP アドレスと宛先ポート番号である．Snort ver2.8 における ASCII 文字列ベースの重要ルール 4,881 余りを分析したところ，その内の 81%には宛先ポート番号指定のオプションが付いていることがわかった．また，全体の 51%に

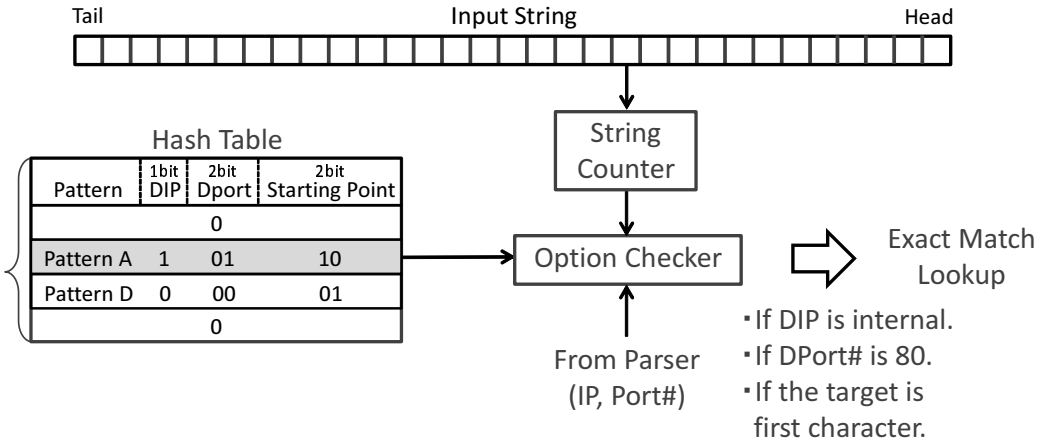


図 5.4: 抽出ルールオプションを考慮したフィルタリングアーキテクチャ

は宛先 IP アドレス指定オプションが付いていることもわかった．本手法では，これらの指定オプションをハッシュテーブルに記載することで処理オフロードに用いる．ここで，宛先 IP アドレスおよび宛先ポート番号をそのままハッシュテーブルに記載すると，1 エントリあたり 48bit を要し，ハッシュテーブルの巨大化が懸念される．そこで，本論分ではこれらの情報を圧縮する．

先程と同様に Snort ルールを分析したところ，表 5.1 および表 5.2 に示すように，宛先 IP アドレス指定のオプションは 2 種類に，宛先ポート番号指定のオプションは 3 種類に大別することができる．このことから，提案手法では，宛先 IP アドレス指定オプションを内部ネットワークか指定なしの 2 種類で表し，宛先ポート番号指定オプションを 80 番かそれ以外，または指定なしの 3 種類で表すことで，それぞれを 1bit，2bit まで圧縮する．本論文では，宛先 IP アドレスが指定なしの場合を 0，内部ネットワーク指定の場合を 1，宛先ポート番号が指定なしの場合を 00，80 番指定の場合を 01，それ以外の番号指定の場合を 10 と設定した．

本手法では，上述したオプション情報を図 5.4 に示したように，ハッシュテーブルに記載する．そしてハッシュマッチングがあった場合には，これらの情報が Option Checker モジュールへと転送され，デコードした後にパーサからの IP アドレス，ポート番号情報と比較される．図 5.4 の例では，パケットの IP アドレスが内部ネットワークであり，ポート番号が 80 であるという情報がパーサから得られた時にのみ，入力文字列が完全一致検索モジュールへと転送される．

5.4.2 探索位置指定を用いたフィルタリング手法

前項で述べた宛先 IP アドレス指定，宛先ポート番号指定と同様，Snort ルールにおいてよく使われるオプションとして探索位置指定がある．このオプションは，ペイロード中にパターンが現れる文字位置を示す．探索開始位置が設定されている場合には，一度の文字列探索によってパターン探索が完了されるため，多くの入力データをオフロードすることが可能となる．我々の分析によると，Snort ルールセットにおける探索位置指定オプションの設定割合は表 5.3 となっている．

そこで，前節で提案した手法と同様に，探索位置指定情報をハッシュテーブルに追加すること

表 5.1: Snort 全ルールにおける宛先 IP アドレス指定の分類

宛先 IP アドレス指定	割合
内部ネットワーク	51%
指定なし	49%

表 5.2: Snort 全ルールにおける宛先ポート番号指定の分類

宛先ポート番号指定	割合
80	57%
その他の番号	24%
指定なし	19%

表 5.3: Snort 全ルールにおける探索開始位置指定の分類

探索開始位置指定	割合
指定なし	50%
40 文字目	40%
1 文字目	8%
2 文字目から 11 文字目	1%

で，処理のオフロードを行う．本手法では，図 5.4 に示したように，探索開始位置情報を 2bit に圧縮し，ハッシュテーブルに記載する．ここで，2bit の値は表 5.3 に示した分類に対して上から順に数値を割り当てる．例えば，00 なら指定なし，01 なら 40 文字目指定となる．入力文字列がハッシュマッチングした場合，この 2bit の値が Option Checker モジュールへと転送され，デコード後に現在の探索対象位置を数える String Counter の値と比較される．探索開始位置が指定されている場合には，カウンタの値がここで指定された文字数になった時のみ完全一致検索モジュールへ転送することで，処理オフロードが実現される．

5.5 Snort を用いた文字列探索シミュレーションによる評価

本節では，ここまで提案したハードウェアコストの削減手法および完全一致検索処理負荷の削減手法に関して，シミュレーションにより評価を行う．本シミュレーションでは，アプリケーションルータによる NIDS サービスの提供を想定し，ネットワークトラフィックに対し Snort の適用を行った．まず，本シミュレーションの環境について述べ，その後に提案手法を評価する．

表 5.4: ネットワークトレースの詳細

項目	値
ストリーム数	47,514
宛先ポート 80 番の割合	90%
プライベート IP アドレスの割合	17%

表 5.5: シミュレーションの実装環境

項目	ツール名
記述言語	Verilog-HDL
論理シミュレーション	Cadence NC-Verilog LDV5.7
ASIC 合成ツール	Synopsys Design Compiler X-2005.09
ASIC 合成用ライブラリ	FreePDK OSU Library (45nm)[130]

5.5.1 シミュレーション環境

本シミュレーションでは，2 つの実装を行っている．まず、図 5.1 で提案したアーキテクチャをソフトウェアにより実装し，慶應義塾大学理工学部西研究室の外部ネットワークに通ずるゲートウェイの WAN 側の NIC において取得したネットワークトレースを評価データとして用いることで，実ネットワーク環境を想定した文字列探索処理負荷の評価を行った．使用したネットワークトレースの詳細は表 5.4 に示した．完全一致検索モジュールには NIDS ソフトウェアである Snort を実装し，Snort が呼び出される回数を完全一致検索回数として処理負荷の指標とした．ここで，抽出ルールとして用いた Snort のルールセットは snortrules-snapshot-2.8 であり，rules フォルダ内において初期状態でコメントアウトされていない 4881 のルールを対象とした．

次に，提案した各手法におけるフィルタリングモジュール全体をハードウェア記述言語 Verilog によって実装し，動作の確認および論理合成による回路規模の見積もりを行った．本シミュレーションの環境は表 5.5 に示した．

5.5.2 ハードウェアコストの削減効果に関する評価

5.3 節で提案した二つのハードウェアコスト削減手法に関して評価を行った．Verilog の論理合成による各手法の回路規模の測定結果は，本章の最後に表 5.6 として示した．表 5.6 には，フィルタリングモジュール内の CRC 関数の回路規模，インデックスメモリに要するメモリ容量，フィルタリングモジュール全体の回路規模をそれぞれ記載した．なお，手法の項目にある Conventional は従来の単純な手法を，insensitive は case-insensitive アプローチを，prefix(5) は 5 文字目までの文字長制限アプローチを，option は指定オプションを用いたフィルタリング手法をそれぞれ表している．

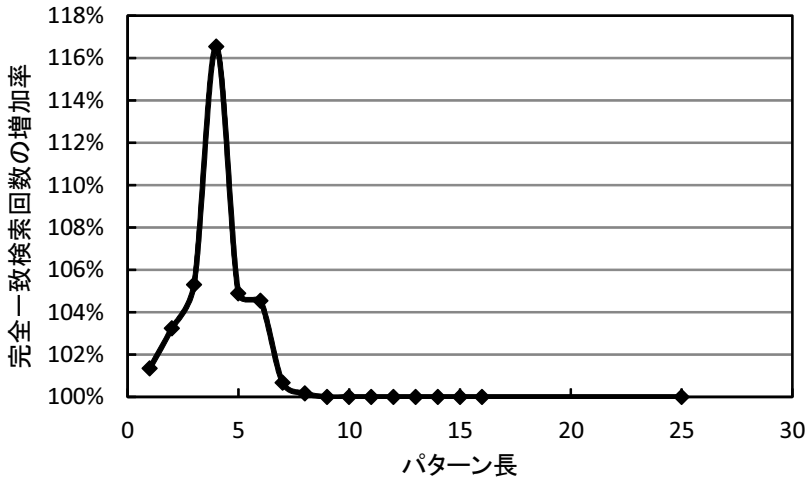


図 5.5: case-insensitive アプローチによる完全一致検索回数の比較結果

まず，case-insensitive アプローチに関して考察を行う．本手法は，case-sensitive パターンと case-insensitive パターンの両方に対応するために必要となるフィルタリングモジュールの実装を case-insensitive のみに制限することで，表 5.6 に示したように，ほぼ 1/2 のハードウェアコストでの実装を可能とする．しかしながら，大文字と小文字を区別すべきパターンであっても区別せずハッシュ化することで false positive が発生してしまい，完全一致検索負荷が増加してしまうことが考えられる．そこで，シミュレーションにおいて，case-insensitive のフィルタリングモジュールのみを実装した場合と，両方を実装した場合での完全一致検索回数の測定を行い，比較結果を図 5.5 に示した．

図 5.5 では，文字長毎のパターン探索において，case-sensitive，case-insensitive の両方を実装した場合に対する case-insensitive アプローチの完全一致検索回数の増加率を表した．図を見ると，文字長が 4 の場合のパターン探索では完全一致検索回数が 16.5%程増加していることがわかる．しかしながら，これは全体として見たときには 1.6%程度の完全一致検索回数の増加であるため，ハードウェアコストの削減効果を考慮すると，十分に有効な手法であると言える．

次に，文字長制限アプローチに関して考察を行う．本手法では，本来，Rabin-Karp 法がパターン全体をハッシュ化することに対し，文字長の長いパターンに対しては先頭の数文字のみをハッシュ化することで，フィルタリングモジュールの実装を数文字分に制限できる．しかしながら，先頭数文字分のみの比較によりフィルタリングを行うため，case-insensitive アプローチと同様に flase positive が増加する．そこで，先程と同様に，シミュレーションにおいて全パターンの文字長分のフィルタリングモジュールを実装した場合と，本手法により数文字分の実装に留めた場合での完全一致検索回数の比較を行い，結果を図 5.6 に示した．なお，シミュレーションでは，30 文字超までハッシュ化した場合の比較も行っているが，結果としての重要度は低いため図 5.6 には記載していない．

図を見ると，先頭 5 文字までのハッシュ値を用いたフィルタリングであっても，全文字をハッシュ化した場合と比べ，完全一致検索回数の増加は 1.4%程度である．これに対して本手法では，

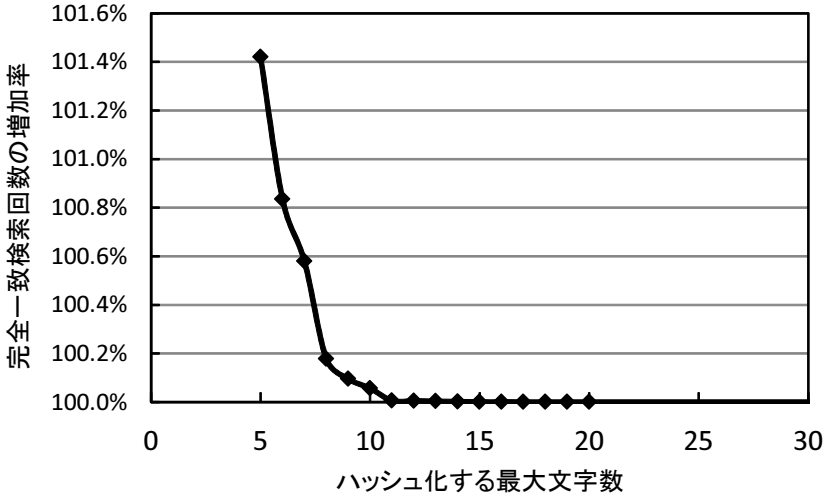


図 5.6: 文字長制限アプローチによる完全一致検索回数の比較結果

ハードウェアコストをフィルタリングモジュール全体で 25.9%程度に，ハッシュモジュールだけならば 1.31%に削減することができるため，有効な手法であることがわかった．

5.5.3 完全一致検索処理負荷の削減効果に関する評価

5.4 節で述べた二つの完全一致検索処理オフロード手法に関して評価する．宛先 IP アドレス指定，宛先ポート番号指定，探索位置指定のそれぞれのオプションを提案手法により適用した際の，完全一致検索回数の測定結果を図 5.7 に示した．ここで，normal は本手法を実装しなかった場合，dip および dport はそれぞれ宛先 IP アドレス指定，宛先ポート番号指定フィルタリングを実装した場合，range は探索位置指定フィルタリングを実装した場合の結果を示す．

この図からもわかるように，一つのフィルタリングオプションを用いた場合でも完全一致検索回数を 20%前後まで削減できている．また，3 つのフィルタリングオプションを全て組み合わせることで，4%まで処理回数を削減できる．このことから，本手法を適用した場合に，100Gbps ネットワーク下での文字列探索において完全一致検索モジュールに求められるスループットは 4Gbps 程度あればよいことがわかった．3.2.3 節で紹介した既存研究の完全一致検索手法において達成されるスループットは 7Gbps 程度であったことから，これらの既存研究を用いて 100Gbps ネットワークの処理が十分に可能であると考えられる．

最後に，本章で提案した各手法の性能を定量的に評価する．本論文では，性能評価指標 Performance を以下の式に計算した．

$$Performance = \frac{\text{従来手法の完全一致検索回数}}{\text{提案手法の完全一致検索回数}} \times \frac{\text{従来手法の全体回路規模}}{\text{提案手法の全体回路規模}} \tag{5.1}$$

Performance の計算結果は表 5.6 に Perf. として示している．本章で提案した 4 つの手法を全て組み合わせることで，従来手法に対して 56 倍の性能評価指標が得られることがわかった．

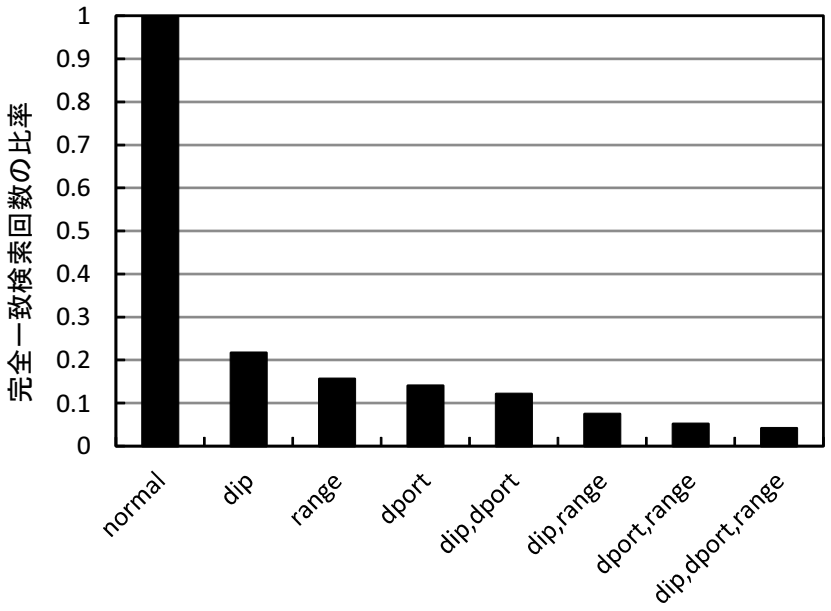


図 5.7: 指定オプションを用いたフィルタリングによる完全一致検索回数の比較結果

表 5.6: 提案手法のシミュレーション結果のまとめ

手法	完全一致検索回数の比較	CRC 関数 [μm^2]	メモリ容量 [kByte]	全体回路規模 [μm^2]	Perf.
Conventional	100%	86,373	184	161,216	1
+insensitive	102%	43,187	92	80,609	2.0
+insensitive +prefix(5)	103%	1,128	100	41,804	3.7
+insensitive +option	4%	43,187	129	95,659	42
+insensitive +prefix(5) + option	5%	1,128	140	58,074	56

5.6 本章のまとめ

本章では，アプリケーションルータにおける文字列探索処理に焦点を当て，100Gbps ネットワーク下におけるアプリケーションルータの文字列探索処理の実現を目指した．既存の文字列探索に関する高速化手法は，7Gbps 程度のスループットを達成することが限界であり，100Gbps ネットワークの処理には不十分であった．加えて，アプリケーションルータではパターン数が多くなることから，複数パターンを高速に探索できる手法が望ましい．そこで，Rabin-Karp 法を基に入力パケットとパターンのハッシュ値を比較することで完全一致検索処理をオフロードし，なおかつ複数パターンを一度に探索するハードウェアアーキテクチャを提案した．更に，アプリケーションルータの情報抽出ルールに基いてパケットのオフロードを行う手法を併せて提案した，

Rabin-Karp 法のハードウェア実装においては，パターン長の数と同数のフィルタリングモジュールを実装しなければならず，100 を超えるパターンが存在するアプリケーションルータにおいては実装コストが増大する．そこで，まず，case-insensitive アプローチと文字長制限アプローチを提案し，実装コストを削減した．case-insensitive アプローチでは，ハッシュ値の比較を case-insensitive のみで行い，case-sensitive なパターンは後段の完全一致検索モジュールで検索することで必要なハードウェアコストを 1/2 に削減した．本手法では，ハッシュ値を case-insensitive として比較することから，全体として 1.6%程度の false positive が発生した．しかしながら，本手法は 1.6%の負荷増加に対してハードウェアコストの削減効果が大きいため，十分に実用的である．次に，文字長制限アプローチでは，実装するフィルタリングモジュールを数文字分に制限することでハードウェアコストを削減した．この文字長を超えるパターンに対しては，先頭数文字分のみをハッシュ化し，入力文字列とのマッチングを行った．文字長制限アプローチでは文字長の長いパターンの場合，パターン全体ではなく先頭数文字のみを比較することから，false positive が生じる．シミュレーションでは，5 文字までのハッシュ化に制限した場合，全文字をハッシュ化した場合と比べ，1.4%程度の false positive が生じた．これに対して，本手法はハッシュモジュールを 1.31%に，フィルタリングモジュール全体を 25.9%に削減でき，有用なハードウェアコスト削減手法であると言える．

本論文では，更に完全一致検索の処理負荷を削減するため，宛先 IP アドレスおよび宛先ポート番号を用いたフィルタリング手法および探索位置指定を用いたフィルタリング手法を提案した．これらの手法では，インデックスメモリに宛先 IP アドレスとして 1bit，宛先ポート番号として 2bit，探索位置指定として 2bit の情報をエントリ毎に加え，ハッシュ値の一致した入力に対し，更にパケット毎のフィルタリングを可能とした．これらのフィルタリングオプションを一つ用いた場合でも，完全一致検索負荷を 20%まで削減できることをシミュレーションにより示した．また，フィルタリングオプションを全て組み合わせることで，完全一致検索負荷を 4%まで削減できることを示した．

最終的に，本論文で提案した手法を全て実装した場合には，従来の 36%のハードウェアコストで完全一致検索処理負荷を 5%まで削減できることを示した．これにより，アプリケーションルータにおいて 100Gbps ネットワーク下で文字列探索処理を行うには，5Gbps 程度のスループットが達成できればよく，既存研究によって実現可能であることが示された．

第6章　マルチコンテクストキャッシュを用いたテーブル検索の高速化，低消費電力化

6.1　本研究の動機

近年のテーブル検索処理は，TCAM を用いることで 1cycle で所望のエントリを見つけ出す．3.3 節では，TCAM の格段に大きい電力コストや実装コストから，TCAM の多用が問題となっていることを述べた．また，アプリケーションルータでは情報抽出の結果に応じて各種テーブルエントリが追加，変更されることから，テーブル検索処理の負荷が増大する．これによって，TCAM のメモリアクセス速度では処理が間に合わなくなる可能性がある．

そこで，本論文では 3.1.2.1 項で紹介したフローキャッシュを応用し，テーブル検索を高速かつ低消費電力に行う手法を提案する．従来用いられてきたフローキャッシュ方式は，帯域幅の向上によりキャッシュミス数が増加したこと，それに伴いキャッシュミス時のミスペナルティが無視できなくなったことが問題であると 3.1.5.5 項で述べた．しかしながら，フローキャッシュは，既存のテーブル検索アーキテクチャを変更することなく利用できる．すなわち，フローキャッシュを既存の TCAM アーキテクチャの上に加えることで，TCAM のアクセラレータとして利用できる．この場合のミスペナルティは TCAM によるテーブル検索遅延のみであり，既存ルータのテーブル検索性能には影響を与えない．このようなフローキャッシュと TCAM のハイブリッド方式によるテーブル検索の概要を図 6.1 に示した．

3.3 節で紹介したように，TCAM は同規模の SRAM と比較しても格段に消費電力や回路規模，製造コストが大きい．また，当然のことながら，膨大なエントリを格納した TCAM と小規模な SRAM キャッシュでは，SRAM キャッシュのレイテンシのほうが小さい．一般的に，TCAM のレイテンシが 5ns 程度であることは 3.1.5.4 項で述べたが，SRAM のレイテンシはプロセッサの L1 キャッシュのような小規模なメモリを用いた場合，1ns 程度となる．従って，フローキャッシュが

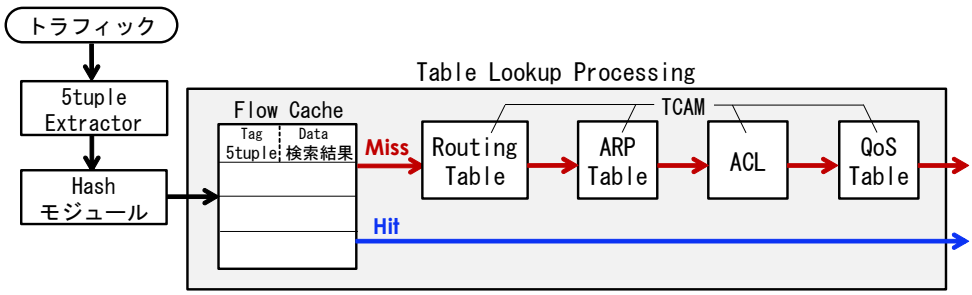


図 6.1: フローキャッシュと TCAM のハイブリッド方式によるテーブル検索

表 6.1: キャッシュ容量とキャッシュミス率，メモリアクセス速度の関係

	32KB SRAM	256KB SRAM	2MB SRAM
レイテンシ	1.08 ns	3.24 ns	11.88 ns
キャッシュエントリ数	1K	8K	64K
キャッシュミス率	25.8%	8.1%	7.6%

キャッシュヒットした場合には，TCAM よりも高速な処理が期待できる．更に本方式では，TCAM が使用されるのはキャッシュミス時のみであるため，TCAM 使用による消費電力の増大を抑制することができる．このように，本方式における処理性能や消費電力はフローキャッシュのミス率に左右される．従って，キャッシュミスの削減が重要な要素である．

一方で，フローキャッシュは，キャッシュタグとして用いるフロー ID（一般的にはストリーム ID と同じ 5 タプル）と，各種テーブル検索結果をまとめたキャッシュデータの bit 数が大きいことから，十分なキャッシュエントリ数を確保することが困難であり，ミスの削減が容易ではない．これに対し従来のフローキャッシュに関する研究では，キャッシュタグのサイズを圧縮する，最適なキャッシュ構成を検討する，より衝突の少ないハッシュ関数を検討するといった手法によって，キャッシュミスの削減を行ってきた．しかしながら，このような手法は，実装において問題点があり，実用的ではないことを 3.1.5 項で述べた．そこで，本章では，より効果的かつ実用的なキャッシュミス削減手法を併せて提案する．

6.2 フローキャッシュの予備評価

一般的にキャッシュはメモリサイズを大きくし，エントリ数を増やす程キャッシュミスを削減することが可能となるが，メモリ容量とアクセス速度の間にあるトレードオフの関係を考慮しなければならない．故に，メモリ容量の大きいキャッシュによりミスを削減することが処理の高速化に繋がるとは言えない．そこで，まず，フローキャッシュのシミュレーションを行い，キャッシュ容量によるミス率の変化を測定した．本シミュレーションは，6.4 節で後述するフローキャッシュシミュレータを用い，実ネットワークトレースである WIDE（Widely Integrated Distributed Environment）トラフィックをワークロードとして使用した．キャッシュ構成，トラフィックなどの詳細は 6.4 節で述べている．

表 6.1 に，プロセッサキャッシュの各レベルと同規模のフローキャッシュを実装した場合に達成されるキャッシュミス率を示した．プロセッサキャッシュの容量およびレイテンシは，2015 年に発売された Intel 第 6 世代プロセッサ Skylake の値を参考にした [132]．更に，シミュレーションにおけるキャッシュ1 エントリのデータサイズは，タグとして 5 タプル 13Byte，データとして 15Byte（ルーティングテーブル検索結果として MAC アドレス 12Byte およびインタフェース番号 1Byte，ACL テーブルの検索結果として 1Byte，QoS テーブルの検索結果として 1Byte）の合わせて 28Byte とし，テーブル検索キャッシュを想定した．

表 6.1 からわかるように，32KB キャッシュは高速ではあるが 25%程度のミスが生じており，256KB

キャッシュは32KBキャッシュに比べ3倍低速だが8%程度のミスに留まっている．この値より，各容量のキャッシュとTCAMを併せた際の平均テーブル検索遅延を計算すると，32KBキャッシュの場合は3.38ns，256KBキャッシュの場合は3.79nsとなる．ここで，TCAMの性能は，論文[77]で紹介されている各種TCAMを参考にし，2.5MBのメモリ容量で200MHz動作，すなわち5nsのアクセス遅延であると考えた．上記の結果と合わせて，256KBキャッシュはこれ以上のミスの改善が望めないことを考慮すると，ミス率が高くとも32KB程度の高速なメモリを用いたほうがスループットは向上できることがわかった．一方で，消費電力について考えると，32KBキャッシュを用いた場合はTCAMの負担が大きくなり，消費電力の削減効果が小さい．それに対して，256KBキャッシュを用いた場合はTCAMの負担が少なく，消費電力を大きく削減することができる．そこで，本研究では，フローキャッシュとしてL1キャッシュサイズのメモリを想定し，生じる25.8%のミスを更に削減する手法を提案することで，スループットおよび消費電力の改善を目的とした．

キャッシュミスの削減にあたっては，まず，ミスの要因を分析することが重要である．一般的にキャッシュでは，以下の三つがミスの要因として知られており，生じたミスはこのいずれかに必ず分類される．

1. **容量性ミス**
   容量性ミスは，キャッシュ容量が小さいことから必ず溢れてしまうエントリによって発生するミスである．キャッシュ容量を増やすことで本ミスは改善される．

2. **衝突性ミス**
   衝突性ミスは，エントリのインデックスが衝突した結果追い出されてしまうエントリによって発生するミスである．キャッシュのway数を増やすことで本ミスは改善される．

3. **初期参照ミス**
   初期参照ミスは，同一コンテクストの最初のデータによって発生するミスである．初期参照ミスはキャッシュの性質上，必ず発生するものであり改善が難しいが，アドレスの連続するデータも合わせてキャッシュするプリフェッチによって改善できることがある．

ここで，先ほどのフローキャッシュシミュレーションにおいて生じたミスの要因を分析し，3要因の比率を計測した結果を図6.2に示した．図ではフローキャッシュの容量を32KB，64KBの2種類に設定し，測定を行っている．図を見てわかるように，容量性ミスはキャッシュ容量を大きくすることで改善される．しかしながら，前述してきたようにフローキャッシュの容量を増やすことはアクセス遅延の増加に繋がる．そこで，本研究では衝突性ミスおよび初期参照ミスの削減に焦点を当て，ネットワークトラフィックの特性に基づいてエントリを最適に制御するマルチコンテクストキャッシュを提案する．

6.3 マルチコンテクストキャッシュの提案

マルチコンテクストキャッシュは，フローキャッシュの衝突性ミスおよび初期参照ミスを削減するために，キャッシュエントリの最適な制御を行う．ここで，エントリ制御方法として置換ポリ

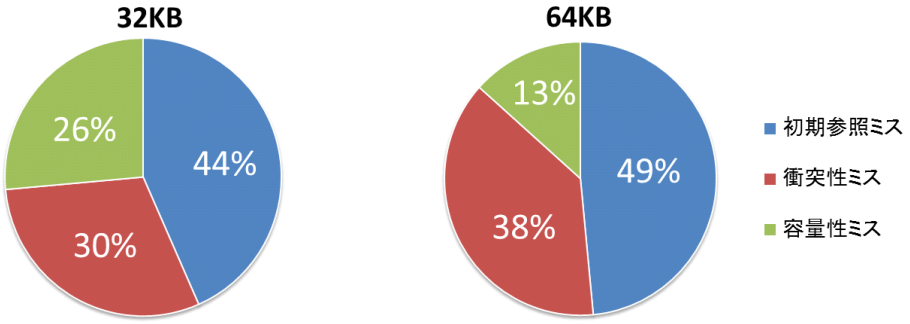


図 6.2: キャッシュミス要因の分析

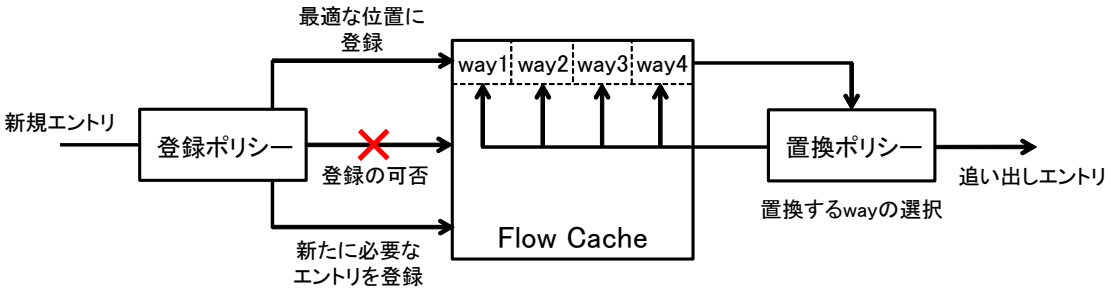


図 6.3: 登録ポリシーおよび置換ポリシーの概要

シーと登録ポリシーの 2 種類を提案する．それぞれの処理概要を図 6.3 に示した．

置換ポリシーは既に登録されているエントリの中から追い出しエントリを決定する．これは 3.1.2.1 項で述べた追い出しアルゴリズムに等しい．一般的な置換ポリシーでは，各 way が追い出し優先度を持ち，追い出し優先度の高いものから順次エントリを入れ替えるという方式が多い．追い出し優先度の決定にはタイムスタンプや参照回数等様々なパラメータが用いられる．3.1.2.1 項で述べたように，最もよく用いられる置換ポリシーは Least Recently Used （LRU）であり，参照された時間が一番古いものから置換を行う．しかしながら，LRU は多くのキャッシュシステムで高い性能を発揮することから，経験的に用いられているだけであり，フローキャッシュに対して最適にチューニングされた置換ポリシーを用いることで，LRU よりも更に高いパフォーマンスを得ることが可能になると考えられる．

一方で，登録ポリシーはキャッシュミスしたパケットのヘッダ情報を解析し，当該パケットのキャッシュ登録の可否や最適なエントリ登録位置を決定する．登録ポリシーによって，パケットが持つ情報を反映したキャッシュ制御が可能となる．例えば，登録ポリシーでキャッシュ不要なフローを判断し，キャッシュ登録を拒否することで，キャッシュエントリの汚染を防ぐことができる．

マルチコンテクストキャッシュでは，様々な要因により生じるミスに対応するため，複数の登録ポリシーおよび置換ポリシーを提案する．図 6.4 に提案手法全体の分類を示した．本論文では，置換ポリシーとして衝突性ミスの削減を目的とした Hit Priority Cache を，また登録ポリシーとして衝突性ミスの削減を目的とした Attack Aware Cache と Port Split Cache，初期参照ミスの削減を

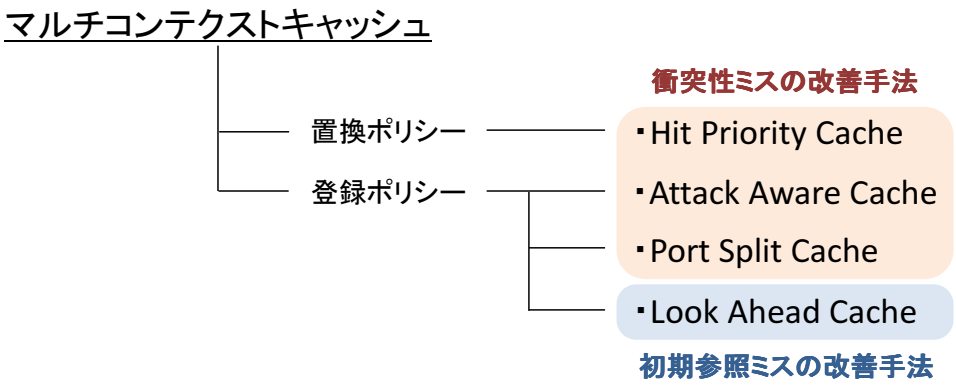


図 6.4: マルチコンテクストキャッシュの分類

目的とした Look Ahead Cache をそれぞれ提案した．以降では，各手法について述べる．

6.3.1 Hit Priority Cache の提案

置換ポリシーでは，既に登録されているエントリの中から，いかにしてキャッシュに有用なエントリを残し，不要なエントリを迅速に追い出すかが重要となる．Hit Priority Cache は，有用なエントリに高い優先度を与え，不要なエントリに低い優先度を与えることで，メモリ利用効率の向上を目指す．

まず，フローキャッシュにおける有用なエントリと不要なエントリについて考察することから始める．ネットワークにはデータサイズの大小様々なフローが流れている．例えば，動画などの大容量コンテンツによる通信データはデータサイズが MB，GB オーダとなるため，パケット数の格段に多いフローとなる．それに対して，ARP のような 1 パケットしか送られないフローも存在する．ネットワークでは，これら 2 種類のフローのパケットがトラフィックの大部分を占めているという，Elephant and Mice の法則が広く知られている [133, 134, 135]．これはすなわち，ネットワークを流れるパケットを見たとき，その大部分は極少数のパケットで構成されるフロー（Mice フロー）か極多量のパケットで構成されるフロー（Elephant フロー）だということである．実際に，前述した WIDE トラフィックと，学術情報ネットワーク SINET4[136, 137] のトラフィックトレース（SINET トラフィックと呼ぶ）を解析し，フローを構成するパケット数の分布を測定した結果を図 6.5 に示した．ここで，WIDE トラフィックは 150Mbps 回線のトラフィックトレースであり，2009 年 4 月 2 日の 0:00-0:15 の間に採取されたおよそ 2500 万パケットを解析に用いた．また，SINET トラフィックは 10Gbps 回線のトラフィックトレースであり，2010 年 6 月 17 日の 14:26-14:57 の間に採取されたおよそ 4 億パケットを解析に用いた．この図を見てわかるように，トラフィックの 90%程度は 10 パケット以下の Mice フローに占められている．一方で，パケット数が 10 万を超える Elephant フローの存在も確認できる．上記の測定結果からも，ネットワークは Mice フローと Elephant フローに占められていることがわかる．

フローキャッシュにおいて，Mice フローの重要度は低い．なぜならば，Mice フローはキャッシュ

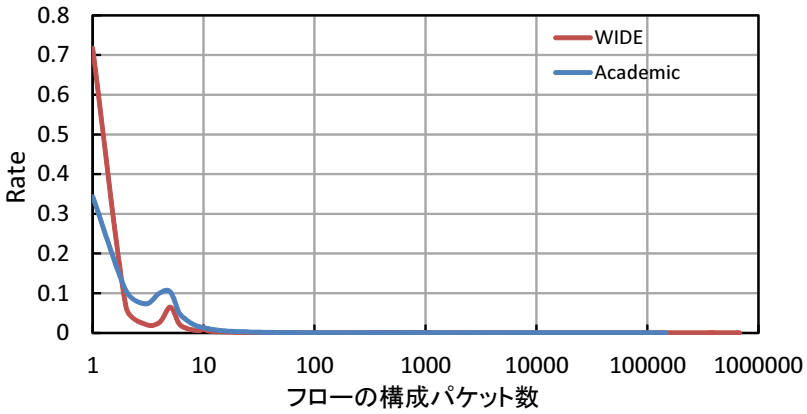


図 6.5: フローを構成するパケット数の分布

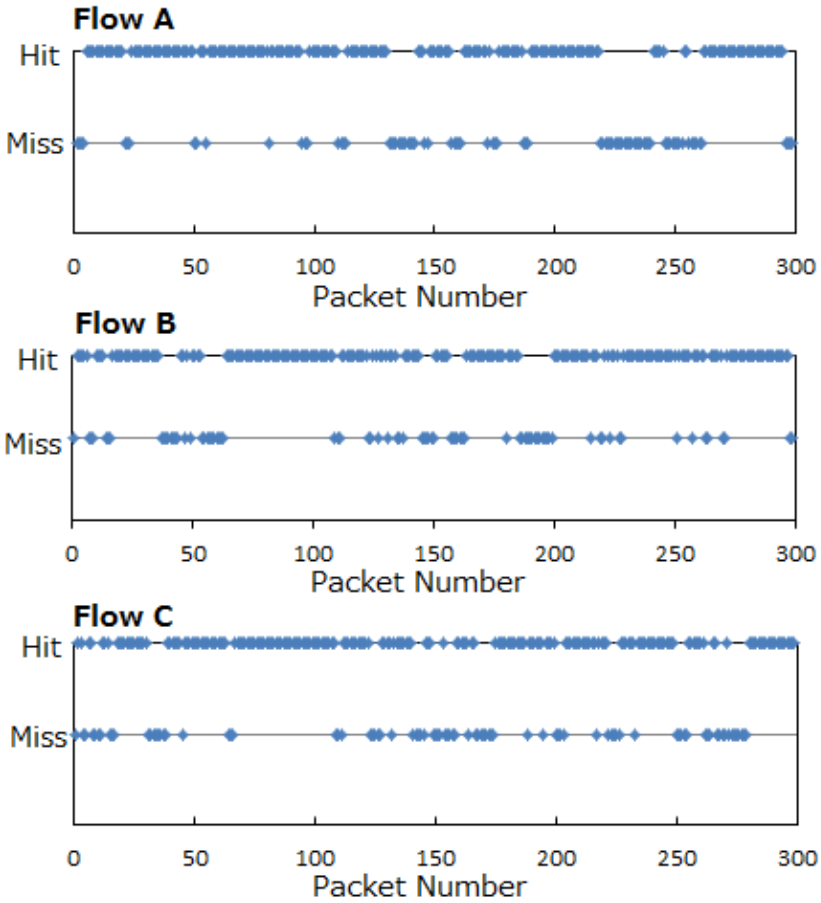


図 6.6: Elephant フローにおけるキャッシュヒット，ミスの発生状況

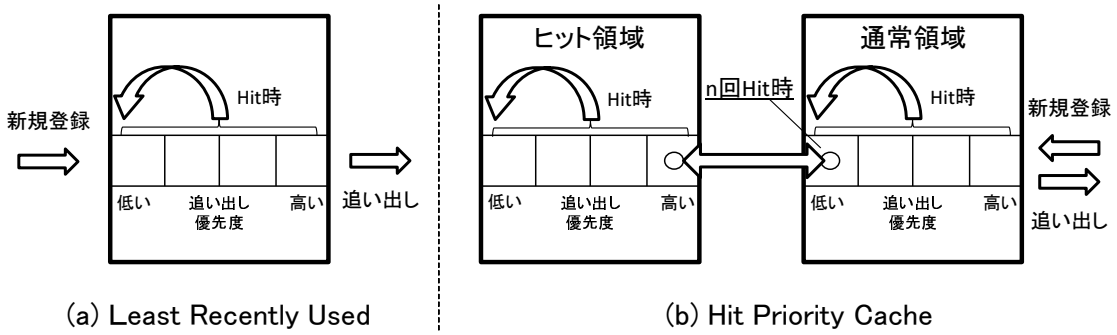


図 6.7: LRU および HPC の概要

ヒットする可能性が低いからである．特に，1 パケットで構成されるフローはキャッシュヒットすることがないためキャッシュ不要であると言える．このような Mice フローが大量にキャッシュアクセスすることで，キャッシュ内の有用なエントリが追い出されるキャッシュ汚染が生じることが考えられる．

一方で，Elephant フローは高いキャッシュヒットの可能性を持っているため重要度が高い．例えば，先ほどの測定結果にあった 10 万パケットを超える Elephant フローならば，最大で 10 万回以上のヒットが期待できる．ここで，図 6.6 に，フローキャッシュシミュレーションにおける，三つの Elephant フローの 1 パケット目から 300 パケット目までのヒット状況を示した．図 6.6 によると，Elephant フローであってもフロー全体の 1 割程度のパケットがミスとなっており，同じ内容のエントリが何度も出し入れされるという無駄が生じている．

そこで，フローキャッシュにおいては，いかに Mice フローを迅速に追い出し，Elephant フローに高い優先度を与えるかが重要となる．この知見をもとに，本論文では Hit Priority Cache（HPC）を提案した．図 6.7 に HPC の概要を示した．図 6.7 では，利点をわかりやすくするために HPC と LRU を併せて記載している．

HPC では，まずキャッシュ領域を 2 分割し，それぞれをヒット領域，通常領域とする．エントリの登録は通常領域に対して行われる．従来の LRU では，新規エントリは最も追い出されにくいポジションに登録される．しかしながら，このような配置はエントリ追い出しに少なくとも way 数回のエントリ移動を要すため，Mice フローが長く残ることになる．そこで，HPC では新規エントリを最も追い出されやすいポジションに登録することで，最短 1 回のエントリ移動により追い出すことが可能となる．これによって，Mice フローの迅速な追い出しが期待できる．また，HPC では，通常領域において複数回ヒットしたエントリは，ヒット領域の追い出し対象エントリと交換される．このようにすることで，Elephant フローの可能性があるエントリに対して高い優先度を与えることができる．なお，エントリがヒットした場合は，エントリが存在する領域において最も追い出されにくいポジションへと移動させる．HPC によって，Mice フローを最短 1 回のエントリ移動で追い出し，Elephant をフローをより追い出されにくくすることが可能となる．

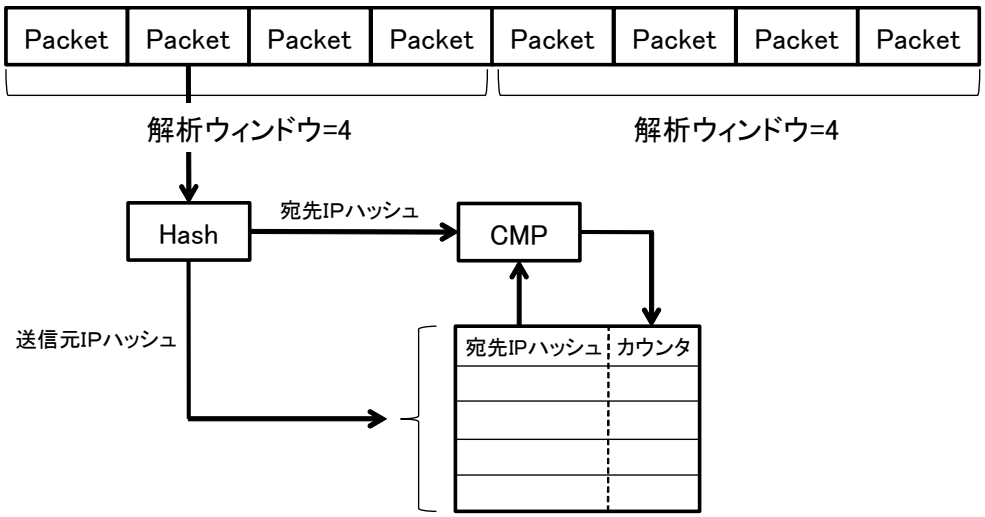


図 6.8: 動的 Attack Aware Cache の処理概要

6.3.2 Attack Aware Cache の提案

フローキャッシュは短時間に大量のアタックパケットが送られるようなネットワークアタックに弱いという特性がある．一般的にこのような攻撃は，悪意を持ったユーザが不特定多数の宛先 IP アドレスに対してパケットを多量に生成し転送するため，短時間でエントリが大量に置換されてしまう．そこで，アタックパケットによるキャッシュ汚染を防ぐために，アタックパケットを識別し，キャッシュ登録を拒否する Attack Aware Cache を提案する．本論文では，到着パケットを解析しながら統計情報を基にアタックパケットを判断する動的 Attack Aware Cache と，ブラックリストといったアタックパケットの特徴を基にアタックパケットを判断する静的 Attack Aware Cache の二つの手法を提案する．

6.3.2.1 動的 Attack Aware Cache

まず，動的 Attack Aware Cache では，ネットワークアタックの多くが短時間に同一送信元 IP から多数の宛先 IP アドレスに対してパケットを送信するという傾向 [138] を利用し，不正送信元を判断する．図 6.8 に動的 Attack Aware Cache の概要を示す．あるパケット数のウィンドウを定め，そのウィンドウ内でアタックの判断を行い，アタックと判断されたパケットのキャッシュ登録をウィンドウサイズに達するまでの間拒否する．次のウィンドウでは，前のウィンドウで判断した結果は無効となる．ウィンドウ内では，到着パケットを解析し，同じ送信元 IP アドレスを持ったパケットの宛先 IP アドレスの種類をカウントする．宛先 IP アドレスの種類がある閾値を超えた時に，該当する送信元 IP アドレスを攻撃元送信源と判断しキャッシュ登録を拒否する．ウィンドウサイズは 512 パケット程度が妥当であり，ウィンドウ内で同一送信元に対して異なる宛先 IP アドレスが 32 個以上存在した場合に，その送信元を攻撃元送信源と判断する．本手法を実装するにあたり，IP アドレスでは bit 幅が大きく必要メモリ量が増大してしまうため，図 6.8 のように IP アドレスを CRC ハッシュ化したハッシュ値を用いる．

6.3.2.2 静的 Attack Aware Cache

次に，静的 Attack Aware Cache では，ネットワークアタックで使用されるアタックパケットをポート情報を用いて判断し，不要であるパケットのエントリ登録を拒否する．静的手法は単純なポート番号を用いてキャッシュの可否を決めるため，単純な回路により実現できる．まず，以下ではネットワークアタックの種類と，それに対して起こるフローキャッシュの問題について考える．

ウィルスやワームによる感染: マルウェアの中には，他のシステムへと感染するウィルスやワームといったものがある．これらの感染方法は，特定のアプリケーションの脆弱性を利用したパケットにウィルスやワームを忍ばせ，不特定多数の宛先に送りつけるという方法を用いることが主である．特に，ウィルスやワームは対策が発表されたからといって，アタックパケットの送信が止まることはなく，ネットワーク中にはこのようなパケットが多々流れる．フローキャッシュでは，このようなパケットによって短時間に大量のエントリが生成されてしまい，有用なエントリが追い出されることが考えられる．

ポートスキャン攻撃: ポートスキャンとは，現在通信に利用可能な開いているポートを探すため，信号を送ってチェックをすることである．ポート番号には，アプリケーションによって一意に決定するものや，ソフトウェアが勝手気ままに使っているものが存在するが，サービスのプログラムに欠陥があり，異常な動作や安全上の問題を孕んでいると目されるポートも幾つか存在している．ポートスキャンを用いた攻撃では，これらを利用し，不特定多数の大量のホストに対して，欠陥のあるポートが開いているか調べるためにパケットを送信する．従って，フローキャッシュにこれらのポート番号を含んだエントリが大量に格納されることが懸念される．

このように，種々のネットワークアタックは特定のポート番号を用いる可能性が高い．例えばポートスキャン攻撃として宛先ポート番号 1433 及び 1434 が用いられることが多々ある [139]．ポート 1433，1434 は Microsoft SQL Server 2000 において利用されるポート番号であり，本アプリケーションの既知の脆弱性を使って伝播するワームの感染に利用されている．また，別の事例では，送信元ポート番号 6000 を用いた攻撃もある [140]．送信元ポート 6000 は，2005 年に流行した W32/Dasher ワームと呼ばれるワームが感染を広げる際に使用していたポート番号だが，現在でもこれに倣って感染を広めるワームが存在しているようである．このようなアタックにおけるフローは一回限りのパケット転送であることが多く，アタックパケットの大部分は 1 パケットフローであることが予測される．

以上の事実から，静的手法では，既知のソフトウェアにおける脆弱性としてアタックに利用されることの多いポート番号について，予め取得し，当該ポート番号を持つパケットのエントリ登録を拒否する．本論文ではこのようなポート番号として，先述した送信元ポート番号 6000 及び宛先ポート番号 1433，1434 と，これらと同様にアタックとして知られる宛先ポート番号 3306 を対象とする．

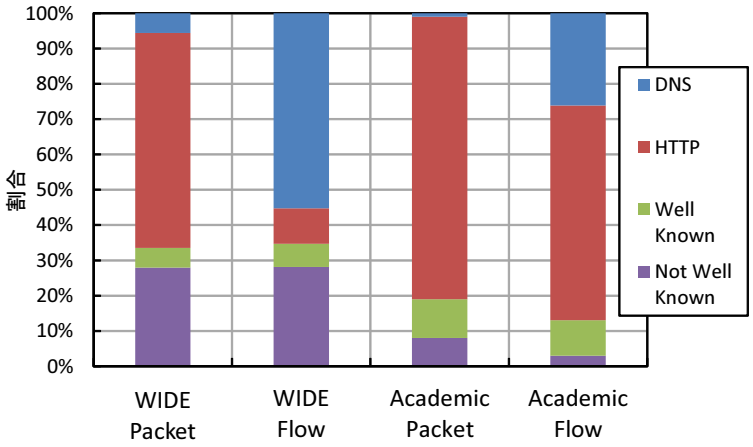


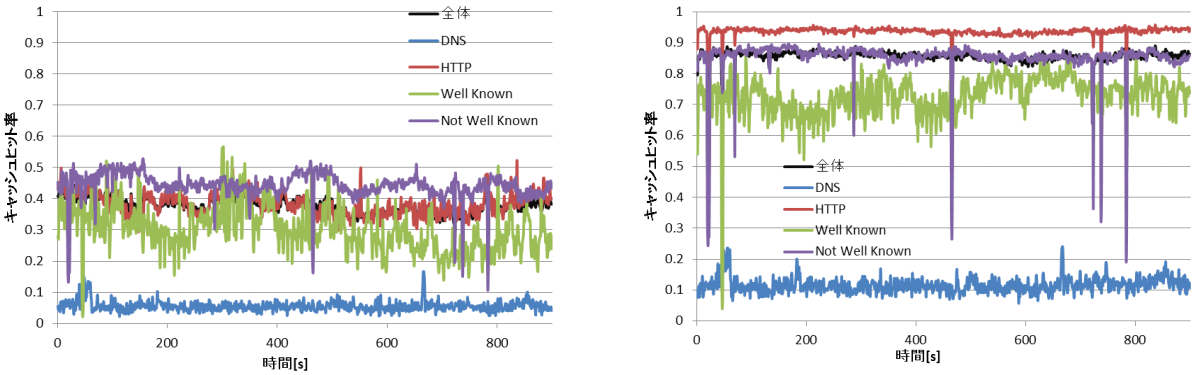
図 6.9: ネットワークにおける各ポートの割合

6.3.3 Port Split Cache の提案

前述したように，ネットワークには Elephant フローや Mice フロー，アタックフローといった様々な特徴を持ったフローが存在する．フローを区別する代表的な手段として，アプリケーションによる分類が考えられる．例えば，ARP や DNS（Domain Name System）といったアプリケーションでは，名前解決を行うために，その情報を持ったサーバとのやり取りが行われる．この通信は，基本的に要求 1 パケットに対して応答 1 パケットが返ってくることで完了される．従って，これらのアプリケーションのフローは Mice フローであることが予測できる．この他でも，例えば HTTP について考えると，HTTP 通信では web ページの HTML や画像，動画の埋め込みファイルが転送されることから，フローは Mice フローにならないと予測できる．このように，ある程度のフロー特性はアプリケーションから予測することが可能である．

容易なアプリケーションの識別方法はポート番号による識別である．例えば，HTTP では 80 番，DNS では 53 番，SSH では 22 番が用いられるように，代表的なアプリケーションは Well-known ポートと呼ばれる 1023 以下のポート番号が一意に割り振られている．一方で，アプリケーションが使用するポート番号を自由に選択することもある．この場合は，1024 番から 65535 番の中で自由に選ばれることが一般的である．

そこで，まずネットワークにおけるポートの分布を測定し，支配的なアプリケーションを分析した．測定結果を図 6.9 に示した．本測定は前述した WIDE トラフィックと SINET トラフィックを使用し，それぞれのトラフィックにおけるポート番号の分布をパケット数およびフロー数の観点から算出している．図 6.9 を見てわかるように，HTTP と DNS はネットワークにおいて支配的である．まず，パケットの 6 割以上が HTTP である．ここで，HTTP をフローの観点から見ると，パケット数に比べ割合が少ないことがわかる．これは，HTTP のフロー構成パケット数が他と比べて多いことを示す．同様に DNS に着目すると，DNS フローはネットワークにおいて支配的であるにも関わらず，DNS パケットは極端に少ない．従って，DNS フローは Mice フローであると考えられる．DNS と HTTP 以外の Well-known ポートに着目すると，ネットワークにおいてほぼ



（a）4entry×4way キャッシュにおける評価　　（b）1,024entry×4way キャッシュにおける評価

図 6.10: 各ポート分類におけるキャッシュヒット率のエントリ数による変化

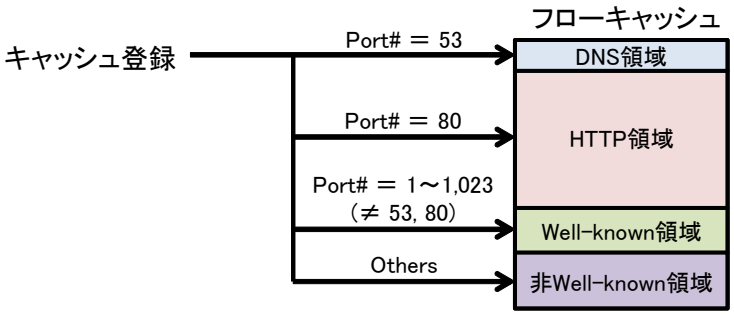


図 6.11: Port Split Cache の概要

10%前後のフローが Well-known ポートを持つことがわかる．また，上記以外の 1024 番以降のポート番号を持ったフローは，10%から 30%程度存在している．

次に，ここで分けた 4 つのポート分類それぞれのフローキャッシュにおけるヒット率を測定し，図 6.10 に示した．本測定では，キャッシュ構成が 4 エントリ ×4way の場合と，1,024 エントリ ×4way の場合の 2 種類で比較している．図 6.10 からは 3 つの重要な特徴が読み取れる．まず一つに，DNS は 10%程度のヒット率しか得られず，エントリを増やすことによるヒット率の向上も少ないことが伺える．これは DNS フローの大部分が Mice フローであることに起因する．しかしながら，数エントリでも与えることで 10%弱のヒットが期待できる．次に，HTTP は 4K エントリ程度で 95%のヒット率が得られており，エントリを増やすことにより大きなヒット率の向上が望めることがわかる．これは HTTP に Mice フローが少ないことが起因していると考えられる．そして 3 つ目に，アタックパケットによるミスの増大が，他のポート分類のヒット率まで低下させていることがわかる．これはアタックパケットによって他のポート分類の有用なエントリが追い出されることが原因であると考えられる．

これらの知見をもとに，Port Split Cache では，ポート分類毎にキャッシュする領域を分けることで最適なエントリ配置を実現する．本手法の概要を図 6.11 に示した．本手法では，キャッシュ領域をポート番号により，DNS 領域，HTTP 領域，Well-known 領域，非 Well-known 領域に分割し，

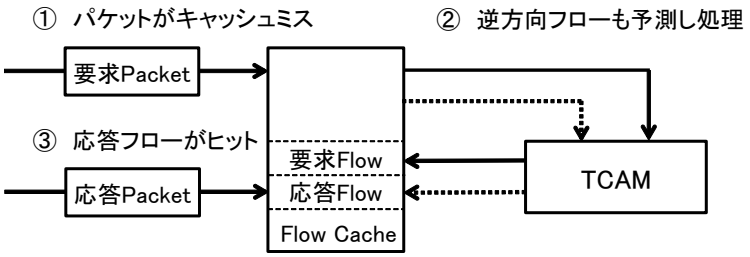


図 6.12: Look Ahead Cache の処理概要

パケットはそれぞれ該当する領域にキャッシュさせる．まず，DNS は Mice フローが多いことから，DNS 領域のエントリ数を極端に少なくすることでエントリの利用効率を向上させる．次に，HTTP 領域にはなるべく多くのエントリを与えることで，ヒット率の向上を図る．更に，Well-known ポートフローと非 Well-known ポートフローにそれぞれキャッシュ領域を与えることで，これらのポート番号を用いたアタックが他の領域に影響を及ぼさないようにする．

6.3.4 Look Ahead Cache の提案

フローキャッシュでは，フローの最初のパケットは必ず TCAM でテーブル検索を行う必要があり，この場合に初期参照ミスが生じる．そのため，キャッシュサイズをどれほど大きくしても達成できるキャッシュヒット率には上限がある．Look Ahead Cache は，このような初期参照ミスに焦点を当て，ネットワークアプリケーションの特性をもとに初期参照ミスを削減する．

多くの通信では 2 回の初期参照ミスが生じる．これは，多くのネットワークアプリケーションが双方向の通信モデルを用いており，クライアントの要求フローに対してサーバが応答フローを返すことで生じる．ここで，応答フローに着目すると，そのヘッダ情報は要求フローの送信元情報と宛先情報を入れ替えることで，予め予測することが可能である．そこで，Look Ahead Cache ではキャッシュミスしたパケットに対し，送信元情報と宛先情報を入れ替えて予測した逆方向フローの処理も行い，予め応答パケット用のフローエントリをキャッシュに登録しておくことで，到着した応答フローの 1 パケット目からキャッシュヒットを可能とする．図 6.12 に Look Ahead Cache の概要について示す．

Look Ahead Cache は，キャッシュミスしたパケットに関して余分にテーブル検索を行うことで，処理負荷が増大することが考えられる．そこで，応答フロー用のテーブル検索は TCAM の処理待ち行列が比較的空いている時を選んで行うことで，アプリケーションルータの処理に影響を与えることなく本手法が実現される．このために，予測された応答フローは通常とは異なる，低い優先度を持たせたキューへ蓄えておく．一般的に要求パケットがルータを通過してから応答パケットが返ってくるまでの時間は，サーバ・クライアント間の物理的な距離や仲介するネットワーク機器の数，アプリケーションの種類など様々な要因によるが，μs または ms オーダであると考えられる．これは ns オーダでテーブル検索を行うルータにとっては，応答フローエントリを作成するのに十分な時間であると言える．

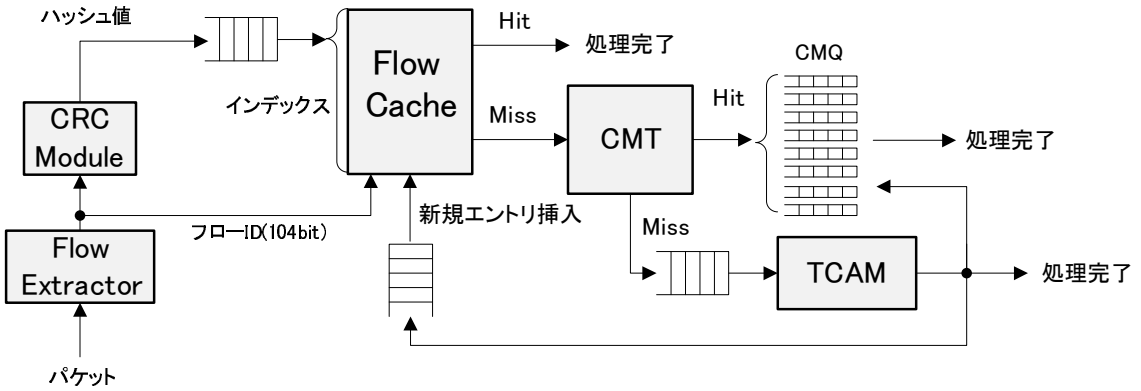


図 6.13: フローキャッシュシミュレータの全体ブロック図

6.4 評価

本節では，マルチコンテクストキャッシュの各手法についてシミュレーションを行い，その有効性について評価した．また，シミュレーションの結果をもとに，テーブル検索にかかるスループットや消費電力量について検討した．

6.4.1 評価環境

まず，本評価におけるシミュレーション環境について述べる．本評価は，主にソフトウェアシミュレータによるキャッシュミス率の測定と，ハードウェア記述言語によるハードウェアコストの測定を行った．以下ではそれぞれに分けて述べる．

ソフトウェアシミュレーション

ソフトウェアシミュレーションでは，3.1.2.1 項で紹介した P-Gear の C-Engine をシミュレートするフローキャッシュシミュレータを実装し，実ネットワークトレースを用いてミス率を測定した．なお，C-Engine における PLC をフローキャッシュとして扱っている．図 6.13 にフローキャッシュシミュレータの全体ブロック図を示す．本シミュレータは C++で実装しており，PCAP や ERF, TSH といった形式のネットワークトレースファイルを読み込んだ後，パケットのタイムスタンプ値に合わせてタイムベースな処理を行う．ここで，時間のレゾリューションは，PCAP や TSH なら μs 単位，ERF 形式なら ns 単位と，トレースファイルの形式により適切な単位が選択される．シミュレータの設定項目としては，この他に，キャッシュのエントリ数や way 数，キャッシュのアクセス遅延，CMH のエントリ数，CMH の way 数，CMH のアクセス遅延，TCAM のアクセス遅延，キューのサイズなどがある．これらの情報を，読み込むトレースファイルのパスと一緒に設定ファイルに記載し，まず，設定ファイルを読む事でシミュレータの初期化が行われる．以降では，シミュレータの処理を順番に述べる．

本シミュレータでは，フローキャッシュがハードウェア実装されることを考慮し，キューやフ

表 6.2: シミュレータ構成

対象	設定項目	構成
フローキャッシュ	マッピング方式	4way set associative
	追い出しアルゴリズム	LRU
	アクセス遅延	1ns
CMT	総エントリ数	1,024
	マッピング方式	4way set associative
	追い出しアルゴリズム	LRU
	アクセス遅延	1ns
TCAM	アクセス遅延	5ns

ラグ，ステート遷移によるハードウェアベースの処理がエミュレートされている．シミュレータの時間を単位時間ずつ進め，トレースファイルの先頭パケットのタイムスタンプと一致した時に，そのパケットを取り出す．まず，最初にパケットから 5 タプルの抽出が行われる．5 タプル情報はそのままキャッシュタグとして使用される他，ハッシュ値をキャッシュのインデックスとして用いるため，CRC によりハッシュ化される．取り出したパケットはキューへと蓄えられ，5 タプルのハッシュ値を用いてフローキャッシュの検索がなされる．検索には，設定ファイルで設定したキャッシュ遅延がタイムスタンプ値に課される．ここでキャッシュヒットした場合には，当該パケットの処理を完了とする．一方で，ミスした場合は CMH へ送られる．

CMH は，パケット処理中に同一フローの後続パケットが来た場合に，ブロッキングされることを防ぐ機構である．まず，CMT との一致が 5 タプルのハッシュ値を用いて検索される．ここでも設定ファイルで設定した CMT のアクセス遅延が処理に課される．CMT でヒットした場合は先に処理中のパケットが存在することを意味するため，CMQ にパケットが転送され蓄えられる．ミスした場合は当該パケットが初見のフローであることを意味するため，パケットはキューに蓄えられた後，TCAM 処理モジュールへと送られる．

TCAM 処理モジュールは，キューに蓄えられたパケットを，TCAM アクセス遅延のもと処理していく．処理の終わったパケットは更にキューへと蓄えられ，CMQ の解放および CMT エントリの削除，キャッシュのアップデートに用いられた後，処理が完了となる．このような処理を 1 単位時間ずつ繰り返すことで，全てのパケットのテーブル検索がシミュレートされる．

本評価では，表 6.2 に示した構成を設定し，シミュレーションを行った．また，ワークロードとして Widely Integrated Distributed Environment[141] のサンプルポイント F において 2009 年 4 月 2 日の 0:00 から 12:00 の間に取得した WIDE トラフィックを使用した．WIDE トラフィックは，日本とアメリカ間を繋ぐ 150Mbps のコアネットワークトラフィックである．本トレースは 15 分間における In&Out パケットおよそ 2500 万パケットが 1 つの pcap ファイルとなっており，この pcap ファイルを断続的に 12 時間分使用し，シミュレーションを行った．この際，1 秒間毎にキャッシュミス率の測定を行っている．

ハードウェアシミュレーション

マルチコンテクストキャッシュは，キャッシュと併せて追加されるモジュールとして低ハードウェアコストであることが望ましく，キャッシュミス削減効果のみで手法を評価することは適当ではない．そこで，各手法を実装する場合の回路規模や最大動作遅延といったハードウェアコストの評価を行うため，ハードウェア記述言語により ASIC を対象とした論理合成シミュレーションを行った．表 6.3 にハードウェアシミュレーションで使用したデザインツールをまとめた．ハードウェア記述には Verilog-HDL（Hardware Description Language）を用いている．本ハードウェア記述言語を用いた論理合成により，ASIC で実装した場合の回路規模や動作の最大遅延を得ることができる．回路規模は組み合わせ回路部分とフリップフロップ回路部分の面積に分けて評価を行った．また，ASIC の論理合成では，合成するソースファイルにメモリを記述していた場合に，メモリに対して誤った論理合成が行われることがあるため，メモリモジュールを外して合成を行っている．メモリは別途，必要なサイズを見積もることで評価を行う．

表 6.3: デザインツールの一覧

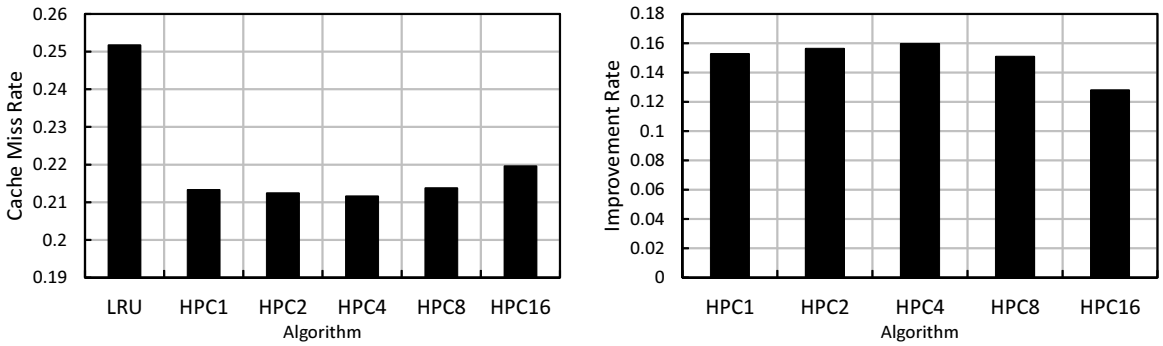
項目	ツール名
言語	Cadence NC Verilog LVD5.2
論理合成ソフト	Synopsys Design Compiler B-2008.09
ライブラリ	FreePDK45 OSU Library （45nm）

6.4.2 キャッシュミスに関する評価

本節では，各手法のソフトウェアシミュレーション結果について示し，得られたキャッシュミス率について評価し，各手法の有用性について議論する．最も簡易的な実装として追加的な処理を一切しない登録ポリシー（Normal ポリシーとする）と，LRU による置換ポリシーを用いた実装を行い，各手法を適用した場合と比較した．以降では，上記の Normal ポリシーと LRU を併せた実装を Normal Cache と呼ぶ．各シミュレーションにおいて特に断りがない場合は，登録ポリシーには Normal ポリシーを，置換ポリシーには LRU を用いている．各シミュレーションの結果は図 6.16 から図 6.23 として本節の最後にまとめている．図中の（a）はシミュレーションによるキャッシュミス率の測定結果を，(b）は Normal Cache と比較した提案手法のキャッシュミス数の割合を示す．なお，この図では，アタックパケット通過時のキャッシュミスの増大を視認しやすくするために，0:00 から 0:15 の結果のみを図示している．図を見ると，この時間において 8 度のアタックが発生していることを確認できる．

6.4.2.1 Hit Priority Cache のミス削減性能

Hit Priority Cache によるキャッシュミス削減効果について述べる．Hit Priority Cache では，通常領域において複数回ヒットしたエントリをヒット領域へと移動させる．そこで，エントリ移動に



（a）キャッシュミス率の測定結果　　（b）LRU に対する提案手法の改善度

図 6.14: HPC におけるエントリ移動に要するヒット数によるミス率の変化

要するヒット回数を 1，2，4，8，16 に設定し，それぞれを HPC1，HPC2，HPC4，HPC8，HPC16 と名づけた．まず，各 HPC と LRU により得られるミス率を 0:00 から 0:15 の間で測定し，比較した結果を図 6.14 に示す．

図 6.14 によると，HPC は LRU よりもキャッシュミス削減効果が高いことがわかる．また，HPC4 まではエントリ移動に要するヒット回数を上げることで，キャッシュミスの削減効果が向上する．しかしながら，HPC8 以降では，ミス削減効果が小さくなっていくことがわかった．4 回のヒットによりエントリ移動を行う HPC4 が最もミス削減効果は高く，16%のミスが削減できている．hpc4 適用時のキャッシュミス率の時間推移を図 6.16 に示している．

次に，HPC4 を用いた場合の 0:00 から 12:00 までのミス率を測定した．HPC では全時間においてキャッシュミスが大きく削減でき，ミス数の削減率は平均 17.9%となることがわかった．また，図 6.16 からもわかるように，HPC はアタックの発生時においても大きくミスを削減できている．これは，新規エントリの登録を最も追い出されやすい位置にしたことで，アタックパケットのエントリがすぐに追い出されるためであると考えられる．このことから，HPC はフローキャッシュにおいて特に有効な手法であることがわかった．

6.4.2.2 動的 Attack Aware Cache のミス削減性能

動的 Attack Aware Cache を適用することによるキャッシュミスの削減効果について述べる．本シミュレーションの結果を図 6.17 に示した．フローキャッシュにおけるキャッシュミス率の急激な増加の大部分は，アタックパケットによる有用なエントリの追い出しが原因となっており，0:00〜0:15 における本シミュレーションにおいてもその影響が確認できた．図 6.17 に示したように動的 Attack Aware Cache を用いることで，0:00〜0:15 間に確認できる 8 度の顕著なアタックに対して，ミスの急激な増加を抑制することができている．これは，図 6.17（b）によると，アタック時には Normal Cache に対して最大で 5.21%，平均して 4.25%キャッシュ性能が向上することを表している．また，動的 Attack Aware Cache を用いることによる，アタックパケットの誤検知の影響について検討すると，キャッシュミスが急激に増加している瞬間以外では，キャッシュミス数は平均して 0.257%増加している程度であり，False Positive の割合はキャッシュ性能に大きな影響を与える程ではないことがわかった．従って，本手法はエントリ数の少ないフローキャッシュにおい

ても十分に効果的であり，ネットワークアタックによるキャッシュエントリの乱れを改善することが可能である．

6.4.2.3 静的 Attack Aware Cache のミス削減性能

静的 Attack Aware Cache を適用した際のキャッシュミスの削減効果について述べる．図 6.18 に本シミュレーションの結果を示した．本手法により，アタックが原因と考えられる 8 度のキャッシュミスの急激な増加において，ミスの増加率を抑制できている．図 6.18（b）は，アタックが発生した際に，Normal Cache と比較してキャッシュ性能が最大 7.96%，平均して 5.53%向上していることを表している．また，動的 Attack Aware Cache はアタックパケットの誤検知により全体的にキャッシュミスが僅かに増加する傾向にあったが，本手法では全体的にも 0.0605%程キャッシュミス率を低減できている．これは，図中で視認できるアタックによるキャッシュミスの増加以外でも，規模の小さなアタックが至る瞬間で起こっており，このような小規模アタックに対しても効果があるからだと考えられる．Attack Aware Cache は全体的なキャッシュミス率を改善する効果は少ないが，アタックによるキャッシュミスの急激な増加を軽減することで，TCAM に急激な負担がかかることを防ぐ．

しかしながら静的 Attack Aware Cache では，シミュレーション中の 2 回目のアタックが発生した時間（トラフィックの時刻 0:01 付近）ではキャッシュミスが改善されていないように，定めたポート番号以外のポート番号を用いたアタックに対しては効果を発揮することができない．本手法ではポート番号によりアタックを判別しているため，未知のアプリケーション脆弱性を用いた攻撃や，アプリケーションによらずポート番号の定まらない攻撃に対しては，これを判断することができない．対策として，動的 Attack Aware Cache と本手法を併用することが考えられる．本手法については後述する．

6.4.2.4 Port Split Cache のミス削減性能

Port Split Cache を適用した際のキャッシュミス率について評価する．Port Split Cache では，表 6.4 に示すエントリ数に従って，キャッシュ領域を 4 分割した．このエントリ比率は，トラフィックにおける各分類の割合をもとに，キャッシュ重要度を考慮し決定した．エントリへのアクセスは，5 タプルのハッシュ値から再計算したインデックスを用いて行った．インデックスの生成手法については次節で述べる．

本シミュレーションの結果を図 6.19 に示した．Port Aware Cache を用いることで，全体的にキャッシュミス率が 2.0%程度低下した．これは，DNS フローに多くのエントリを与えず，HTTP に多くのエントリを与えることで，エントリが有効に活用できたことを示している．更に，本手法では，アタックによるキャッシュ汚染に対しても有効に働いている．アタック時のミス率の増大が平均して 4.1%程度低下した．これは，キャッシュに Well-known 領域と非 Well-known 領域を与えることで，キャッシュ汚染が他の領域にまで及ばなくなったことを示している．結果として，Port Split Cache を適用することで，平均 7.45%のミスが削減されることが示された．

表 6.4: 各領域のエントリ数

	DNS 領域	HTTP 領域	Well-known 領域	非 Well-known 領域
エントリ数	16	624	128	256

6.4.2.5 Look Ahead Cache のミス削減性能

Look Ahead Cache を適用した場合のキャッシュミスの削減効果について述べる．本シミュレーションの結果を図 6.20 に示す．Look Ahead Cache では，全シミュレーション時間においてキャッシュミスが削減できており，平均すると 3.89%のキャッシュミス率低下となった．これは図 6.20（b）に示したように，Look Ahead Cache を用いることで Normal Cache に比べ平均 15.1%のミスが削減されるということである．

本手法によるキャッシュ性能の向上は，衝突性ミスの削減を目的とした他の手法とはその性質が異なる．衝突性ミスの削減手法は，キャッシュ中の有用なエントリの数をできる限り多く保つことで，エントリ数を増加させることに似た効果が得られる．従って，このような手法はキャッシュエントリ数が少ない場合には効果的だが，理論的に可能な最大キャッシュヒット数が増加するわけではない．しかしながら，Look Ahead Cache は本来キャッシュヒットしないはずの，応答フローの 1 パケットからキャッシュヒットさせることを可能とするため，理論的に可能な最大キャッシュヒット数が増加する．

6.4.2.6 Port Split Cache および Attack Aware Cache 複合手法のミス削減性能

本研究の目標は，フローキャッシュの容量を増加させることなくキャッシュミスを削減することである．従って，各提案手法を併せて実装することが望ましい．本節ではこれを複合手法と呼び，検討を行った．複合手法では，手法同士の適用順序や適用範囲によっては効果の重複や阻害が発生し，意図したキャッシュミス削減効果を得られない可能性が考えられる．そこで，各手法の適用順序や効果的となるチューニング方法を検討することで，複合手法において高いパフォーマンスの得られる実装方法の提案を行った．

ポートスキャン攻撃やウィルス，ワームによる攻撃対象は，HTTP や DNS を除くポート番号に多く見られる．このことは，図 6.10 を見てもわかる．一方で，動的な Attack Aware Cache では，全てのパケットを対象として攻撃の検知を行っていた．上記の知見をもとにするならば，HTTP や DNS を検出対象とする必要はない．従って，動的 Attack Aware Cache の適用範囲を Well-known 領域および非 Well-known 領域に限定することで，アタック検知精度が向上できるはずである．

そこで，Attack Aware Cache をフローキャッシュ全体に適用した場合と，Well-known 領域および非 Well-known 領域に限定した場合とで，生じるミス率を測定し比較した．図 6.21 に測定結果を示した．ここで，前者のケースを Normal Case，後者のケースを Best Case と表記している．図 6.21 より，Normal Case の場合には全時間において False Positive が生じることでミスが 2%程度増加しているのに対し，Best Case では False Positive がほぼ発生していないことがわかる．また，僅かではあるが，アタック時のミス数を Normal Case と比較した場合に，Best Case のほうがミス

を削減できていることがわかった．これらのことから，Port Split Cache と Attack Aware Cache は Best Case で実装することにより，アタック検知精度を向上できることが示された．

6.4.2.7 Look Ahead Cache および Attack Aware Cache 複合手法のミス削減性能

Look Ahead Cache と Attack Aware Cache による複合手法の最適な適用順序について検討を行う．Attack Aware Cache が焦点を当てたネットワークアタックは，その性質上，応答パケットが返送されないことが多い．例えばポートスキャン攻撃では，無作為なポート番号に対してパケットを送りその応答パケットの種類により，ポートが開いているかどうかを判別する．しかしながら，大部分のポートスキャンパケットは相手ホストに辿り着く以前にファイアウォール等によりフィルタリングされるため，応答パケットが返送されることはない．また，ポートスキャン攻撃では，ポートが Open 状態にあったとしても相手ホストが応答パケットを返送しないことが起こり得る．従って，アタックに対して応答パケットが返送されることは稀であり，アタックパケットに対して Look Ahead Cache を適用する効果は低いと考えられる．また，ネットワークアタックでは短時間に多量のパケットが送られてくるため，Look Ahead Cache により逆方向フローのエントリを登録すると，多くのキャッシュエントリが追い出されてしまうことが予想される．上記の理由から，まずアタックパケットの判別を行った上でアタックパケットに対しては Look Ahead Cache を適用しないことで，アタックパケットによるキャッシュミスの増大を避ける．

本論文では，Look Ahead Cache と Attack Aware Cache をソフトウェアシミュレータに実装し，ベストケースとワーストケースの処理方法におけるキャッシュミス率の比較を行った．ベストケースでは，キャッシュミスしたトークンに対し，まず Attack Aware Cache（本シミュレーションでは動的 Attack Aware Cache と静的 Attack Aware Cache の複合手法を指す）を適用し，アタックであると判断されたフローには Look Ahead Cache を適用せず，それ以外のフローにのみ Look Ahead Cache を適用する．ワーストケースでは，キャッシュミスした全てのトークンに対して Look Ahead Cache を適用し，キャッシュミストークンと Look Ahead により作成されるトークンの両方を合わせ Attack Aware Cache を適用する．シミュレーション結果を図 6.22 に示した．

シミュレーション結果ではベストケースとワーストケースで共にアタックに対して Look Ahead Cache 単体よりもキャッシュミスを削減できている．しかしながら，各アタックの瞬間におけるキャッシュミス率を比較すると，ワーストケースではベストケースに対してキャッシュミスの削減率が小さいことがわかった．0:00 から 0:15 におけるシミュレーションではキャッシュミス率に影響を与える顕著なアタックが 8 回発生しており，このアタック時にベストケースでは Look Ahead Cache 単体に比べ平均して 5.72%のキャッシュミスを改善しているが，ワーストケースでは 2.24%のキャッシュミス改善となった．これはワーストケースではアタックパケットに対して Look Ahead Cache が適用され，不要なエントリの登録がなされるためである．アタックパケットに対する逆方向フローエントリはヒットする確率が低いため，ワーストケースではキャッシュミスの改善率が低い．更に，顕著なアタック以外の時間におけるキャッシュミス率について観察すると，ベストケースの処理順序では Look Ahead Cache とほぼ同程度となっているが，ワーストケースではキャッシュミスが 1.21 倍に増加した．これは，ワーストケースではキャッシュミスしたトークンと Look Ahead Cache により作成されるトークンの両方に動的 Attack Aware Cache を適用するため，

アタックパケットの誤検知が増加したことが原因だと考えられる．このように，Look Ahead Cache と Attack Aware Cache の複合手法では処理順序を考慮することで，通常時は Look Ahead Cache 単体と同程度のキャッシュミス削減効果が得られ，アタック時には 5.72%のキャッシュミスを改善することが可能であることがわかった．

6.4.2.8 マルチコンテクストキャッシュのミス削減性能

本項では，上述してきたそれぞれの複合手法の最適な実装方法をまとめ，全ての手法を実装したマルチコンテクストキャッシュの評価を行う．まず，マルチコンテクストキャッシュのアーキテクチャを図 6.15 に示す．マルチコンテクストキャッシュでは，まず Port Split Cache モジュールにより，トークンから各領域に合わせたインデックスが生成される．ここで，トークンとは，P-Gear で定義される 5 タプルの値やそのハッシュ値などを含んだデータを表す．Port Split Cache モジュールで得たインデックスをもとにキャッシュの検索が行われ，ヒット時およびミス時のエントリ優先度の管理は Hit Priority Cache モジュールでなされる．ここでキャッシュヒットした場合には当該パケットの処理が完了となる．一方でキャッシュミスした場合は，並列に実装された各 Attack Aware Cache モジュールでアタックの判断が行われ，アタックであると判断されたトークンに対しては，キャッシュ登録および Look Ahead Cache の適用が拒否される．アタックでないと判断された場合は，CMH（Cache Miss Handler）に送られると共に，Look Ahead Cache モジュール，TCAM での処理待ちキューへと送られる．CMH の詳細は 3.1.2.1 項で述べているため，ここでは割愛する．処理待ちキューへと蓄えられたトークンは，TCAM コントローラからのデキュー信号を受信した場合に TCAM へ転送される．TCAM で処理されたトークンのパケットはテーブル検索が完了となり，CMQ の解放およびキャッシュエントリの更新がなされる．一方で，Look Ahead Cache に転送されたトークンは，逆方向トークンへと書き換えられ，キューに蓄えられる．このキュー内のトークンは，上述した TCAM での処理待ちキューと TCAM の処理状態を勘案し，処理待ちキューが空いている時にのみ TCAM へと送られる．本シミュレーションではこのようなアーキテクチャをフローキャッシュシミュレータに実装し，評価を行った．

シミュレーション結果を図 6.23 に示した．図 6.23 ではマルチコンテクストキャッシュのキャッシュミス削減効果をわかりやすくするため，Normal Cache と，Look Ahead Cache によるシミュレーション結果も併せて載せた．シミュレーションでは，Normal Cache での平均のキャッシュミス率が 25.8%であるのに対して，Look Ahead Cache での平均ミス率が 21.9%，マルチコンテクストキャッシュでの平均ミス率が 17.4%となった．マルチコンテクストキャッシュの適用により，通常のフローキャッシュと比べ，平均して 32.6%のミス削減がなされており，本手法の有効性が確認できた．

また，アタック時のキャッシュミスに関しても Attack Aware Cache を併用することで改善できている．本シミュレーションでは，マルチコンテクストキャッシュにより，アタックによる 8 度の急激なミスの増加に対し，通常のフローキャッシュと比べ平均して 18.0%のキャッシュミスを削減可能となった．これは，Attack Aware Cache の動的手法と静的手法の効果が併せられ，更に Hit Priority Cache によってアタックパケットを含めた Mice フローが迅速に追い出されるため，大幅なミスの削減が可能となったと考えられる．アタックに対するミス削減効果が最も大きい場合に

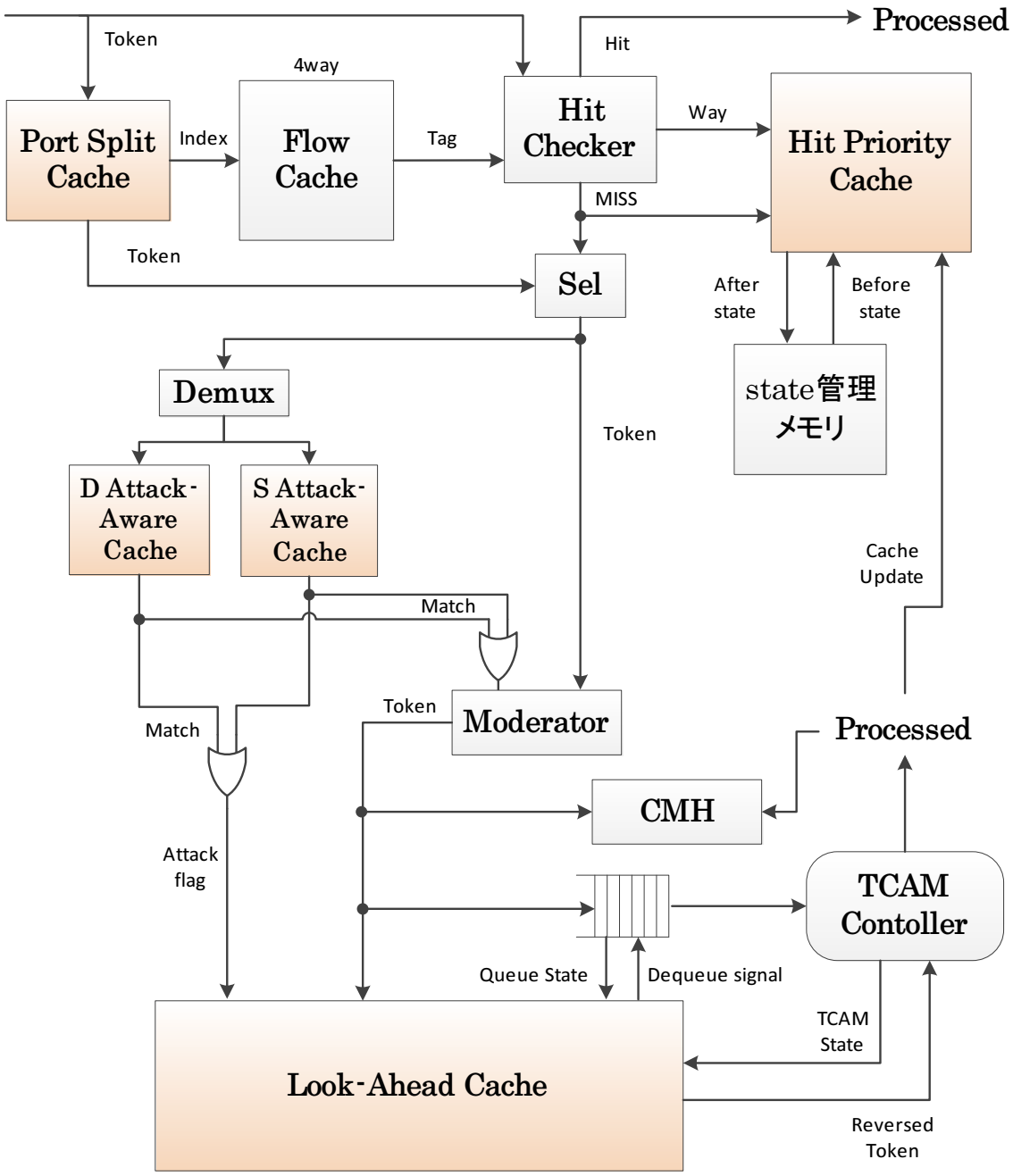
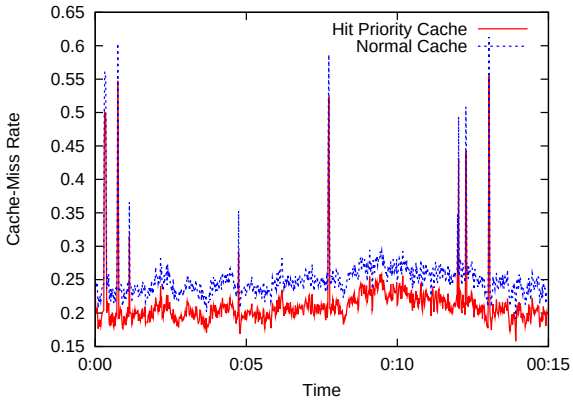
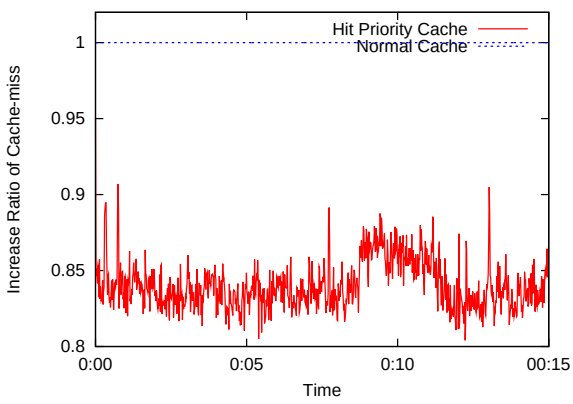


図 6.15: マルチコンテクストキャッシュのアーキテクチャ

は，ミス率を 9.7%低下させていることからも，マルチコンテクストキャッシュのアタックに対する有効性が確認できた．

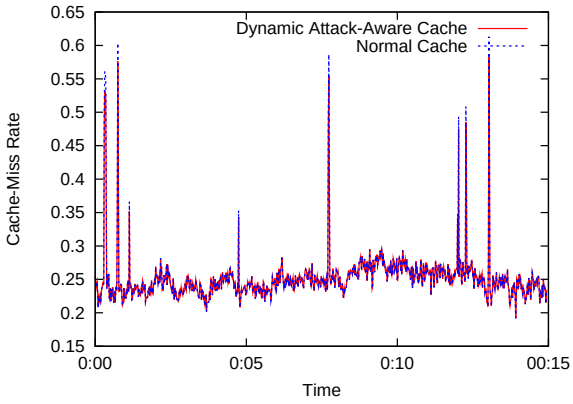


（a）キャッシュミス率の測定結果

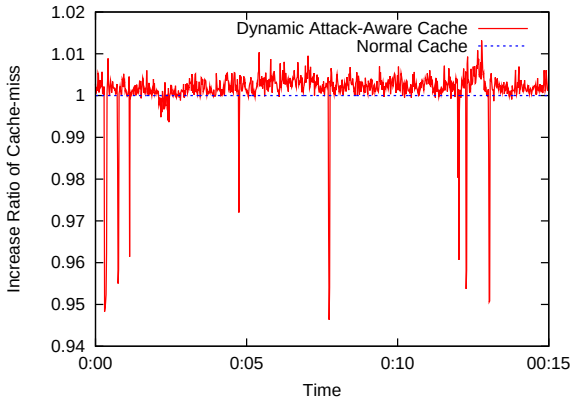


（b）Normal Cache に対するミスの割合

図 6.16: Hit Priority Cache のシミュレーション結果

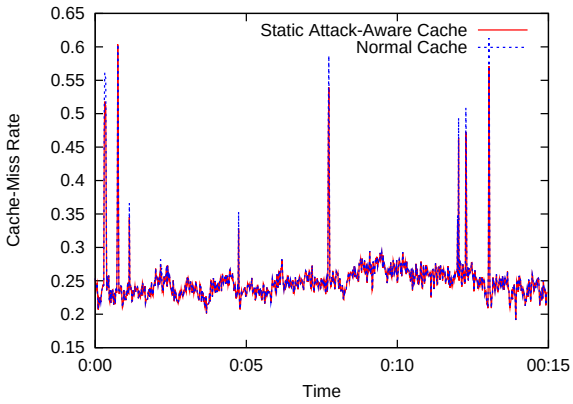


（a）キャッシュミス率の測定結果

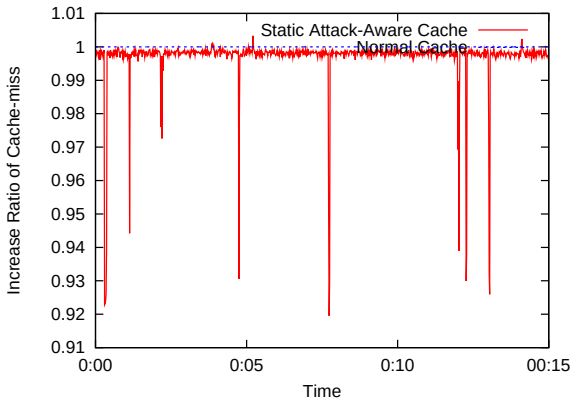


（b）Normal Cache に対するミスの割合

図 6.17: 動的 Attack Aware Cache のシミュレーション結果

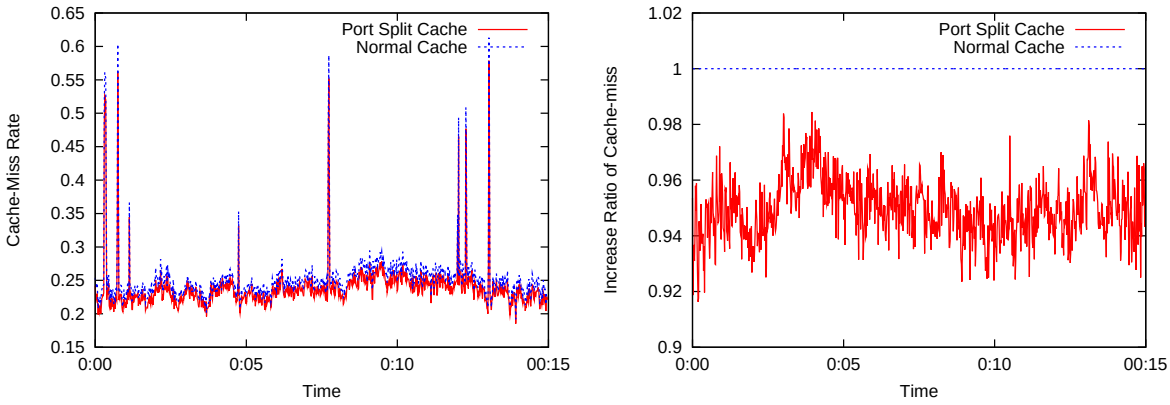


（a）キャッシュミス率の測定結果



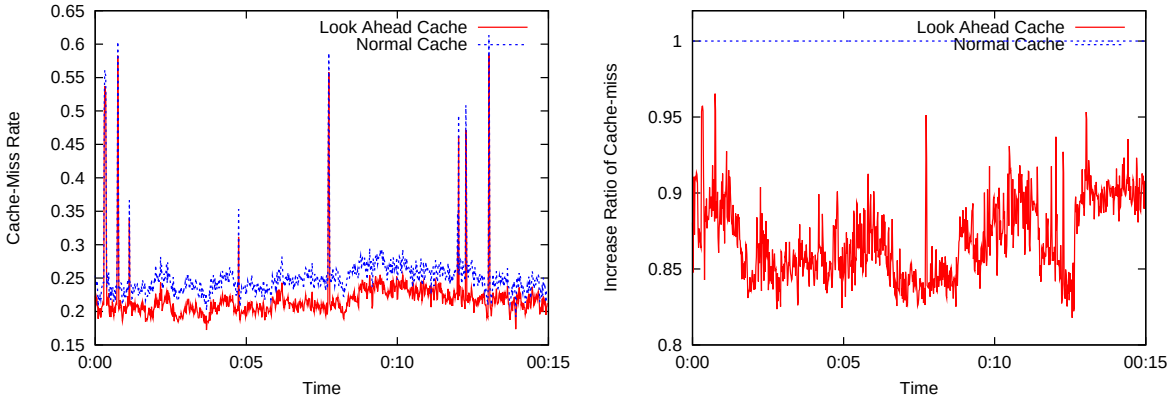
（b）Normal Cache に対するミスの割合

図 6.18: 静的 Attack Aware Cache のシミュレーション結果



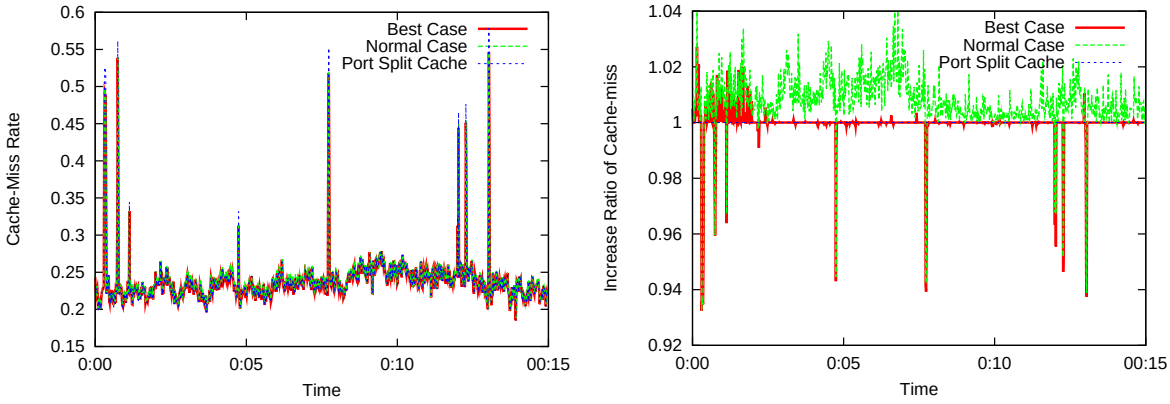
（a）キャッシュミス率の測定結果　　（b）Normal Cache に対するミスの割合

図 6.19: Port Split Cache のシミュレーション結果



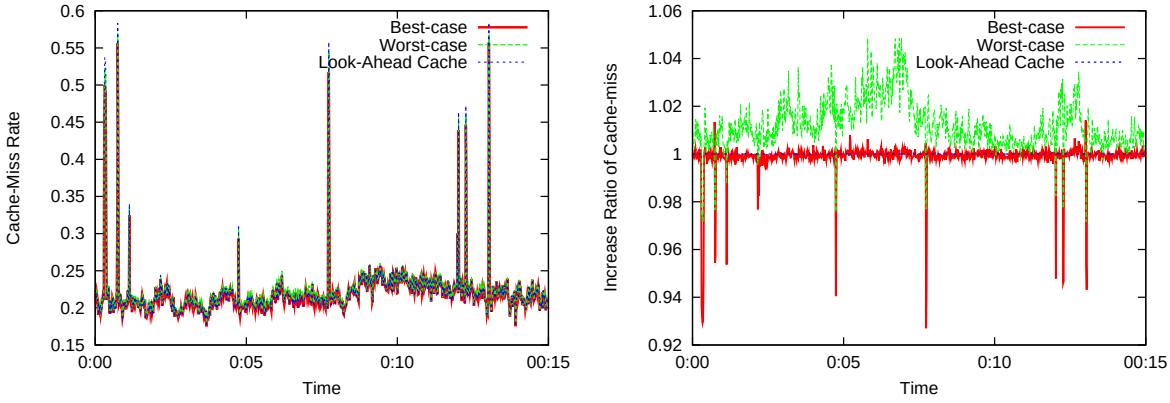
（a）キャッシュミス率の測定結果　　（b）Normal Cache に対するミスの割合

図 6.20: Look Ahead Cache のシミュレーション結果



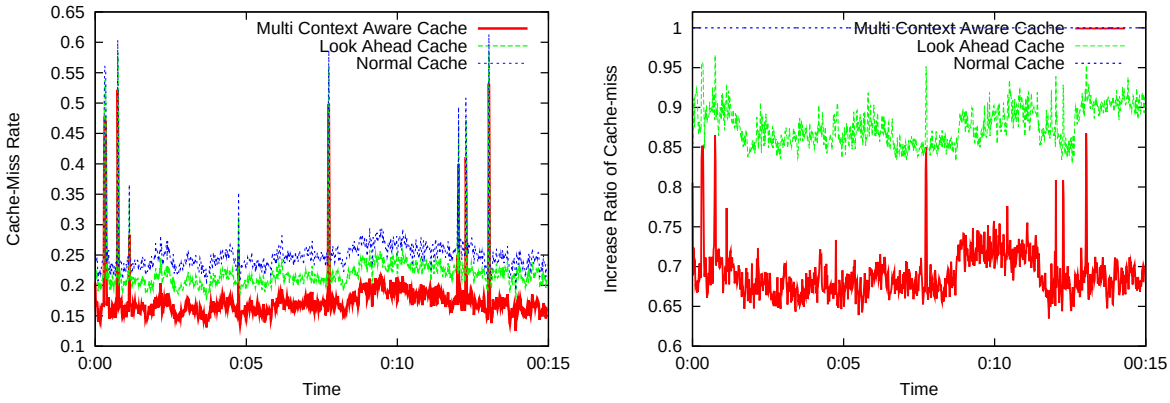
（a）キャッシュミス率の測定結果　　（b）Port Split Cache に対するミスの割合

図 6.21: Port Split Cache と Attack Aware Cache の複合実装によるシミュレーション結果



（a）キャッシュミス率の測定結果　（b）Look Ahead Cache に対するミスの割合

図 6.22: Look Ahead Cache と Attack Aware Cache の複合実装によるシミュレーション結果



（a）キャッシュミス率の測定結果　（b）Normal Cache に対するミスの割合

図 6.23: マルチコンテクストキャッシュによるシミュレーション結果

6.4.3 実装コストに関する評価

本節では，ハードウェア記述言語を用いたハードウェアシミュレーションの結果について述べる．本シミュレーションでは，各アルゴリズムを回路で実現し，論理合成を行うことで回路規模を測定している．ハードウェア論理合成の結果を表 6.5 として本節の最後にまとめている．また，表 6.6 にはシミュレーション結果のキャッシュミス削減率と総回路規模より計算される，各手法の $1\mu m^2$ あたりのキャッシュミス削減率を示している．

6.4.3.1 Hit Priority Cache の実装コスト

置換ポリシーは，キャッシュにおいて必ず一つ実装されるモジュールである．そこで，実装コストの評価は，従来用いられてきた LRU と比較するべきだと考えられる．

まず，LRU の実装に関して説明する．LRU を実現する多くのアーキテクチャでは，各 way の優先度の組み合わせを管理した優先度管理メモリを参照することで追い出しエントリを決定してきた．例えば，4way キャッシュならば 4 つのエントリによる優先度の組み合わせが 24 通りとなるため，これを 5bit のステート値で表現し，ステート管理メモリにインデックス毎に格納する．この値は，エントリのヒット時に，ヒットした way に応じてステート更新モジュールが更新し，再度メモリに書き込まれる．エントリの追い出し時は，まず，ステート管理メモリの当該インデックスから 5bit 値を読み出し，下位 2bit により追い出しエントリを決定すればよい．このような実装は，例えば way 数が 8 になると，考えられる優先度の組み合わせが 40,320 となり，一気に実装コストが膨大となる．このため，LRU の実装は 4way までのキャッシュが主であった．これに対して，HPC は擬似的な 8way キャッシュの挙動を示す．そこで，実装コストが増大しないよう，低実装コストなアーキテクチャを提案する．図 6.24 に HPC の優先度管理の概要を示す．

HPC では，分割された二つの領域をそれぞれ別々の 4way LRU キャッシュと同様に管理する．そして，エントリヒット時には LRU と同様にステート値を更新モジュールにより更新する．また，通常領域のエントリには，ヒット領域への移動に用いるヒットカウンタを持たせる．例えば，

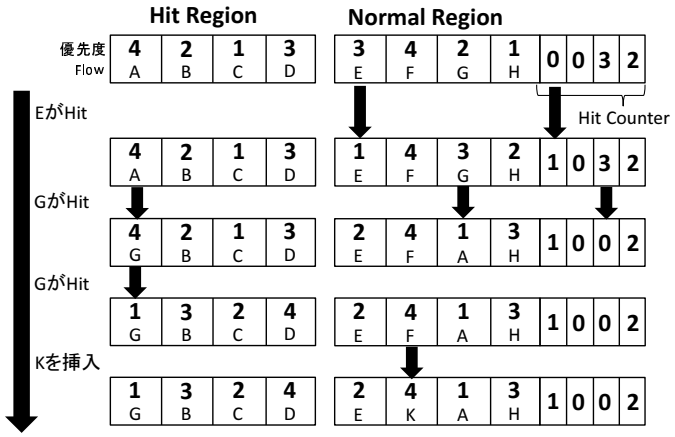


図 6.24: Hit Priority Cache の優先度遷移の例

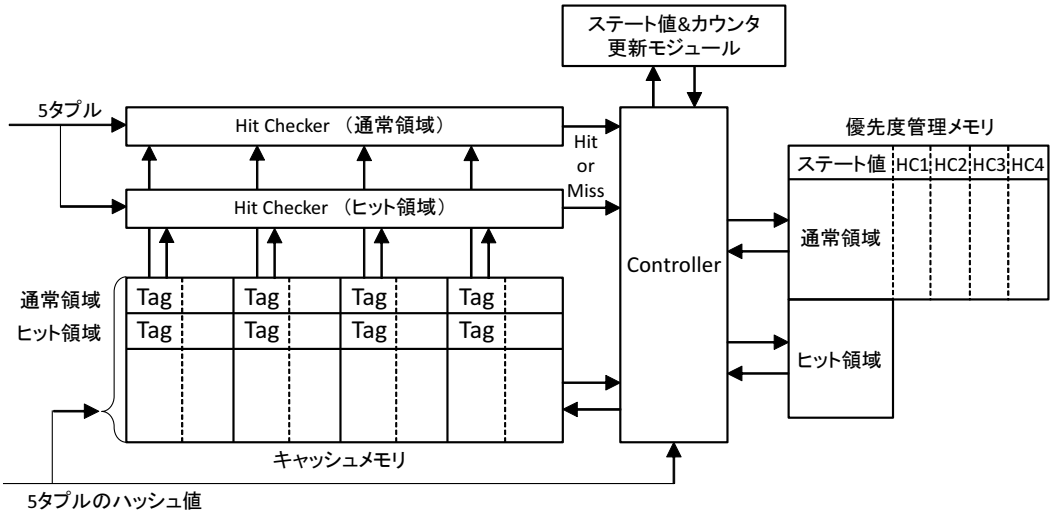


図 6.25: Hit Priority Cache のアーキテクチャ

HPC4 では通常領域のエントリに 2bit の追加が必要となる．ヒットによってヒットカウンタの値が溢れたときに，通常領域の該当データとヒット領域の最も優先度の低いデータを入れ替える．

図 6.24 の例では，まず，通常領域のエントリに対してフロー E および G がヒットする．ここで，それぞれのフローエントリの優先度を LRU と同様に遷移させると共に，ヒットカウンタの値をインクリメントする．フロー G のヒットによりカウンタの値が溢れた時に，ヒット領域のフロー A のエントリとデータを交換する．この場合，優先度は通常領域において遷移させるだけでよく，ヒット領域の優先度は変わらない．そして，次にフロー G が到着しヒットした場合は，ヒット領域において LRU と同様に優先度を遷移させる．新規エントリであるフロー K を挿入する場合は，通常領域に対してエントリを挿入すればよい．このように HPC では，同エントリ数の 4way LRU と全く同様にステート管理が行える．HPC4 では，ステート管理メモリにおける通常領域エントリ分，すなわち総エントリ数の半分のエントリに 2bit を追加するだけでよい．

上記を実現するアーキテクチャを図 6.25 に示した．HPC では，LRU と異なり新規エントリの初期優先度は最も低くなるが，これはステート更新モジュールの更新後の値が変わるだけであり実装コストには影響しない．LRU に対して追加される回路はヒット領域のヒットチェッカとカウンタ更新モジュールのみである．シミュレーションでは，図 6.25 の回路を実装し，回路規模のうち上述した LRU に対する増分のみを表 6.5 に示している．シミュレーション結果では，組み合わせ回路部が $9.25 \times 10 \mu m^2$，フリップフロップ回路部が $2.25 \times 10^2 \mu m^2$ と，LRU に追加されるモジュールは小規模であることが示された．これに対し，キャッシュミスの削減効果は提案手法の中で最も高いことから，フローキャッシュに有効な置換ポリシーであることがわかる．

6.4.3.2 動的 Attack Aware Cache の実装コスト

動的 Attack Aware Cache のハードウェア実装について述べる．図 6.26 にアーキテクチャを示した．動的 Attack Aware Cache では，一つの送信元 IP アドレスから多数の宛先 IP アドレスに対し

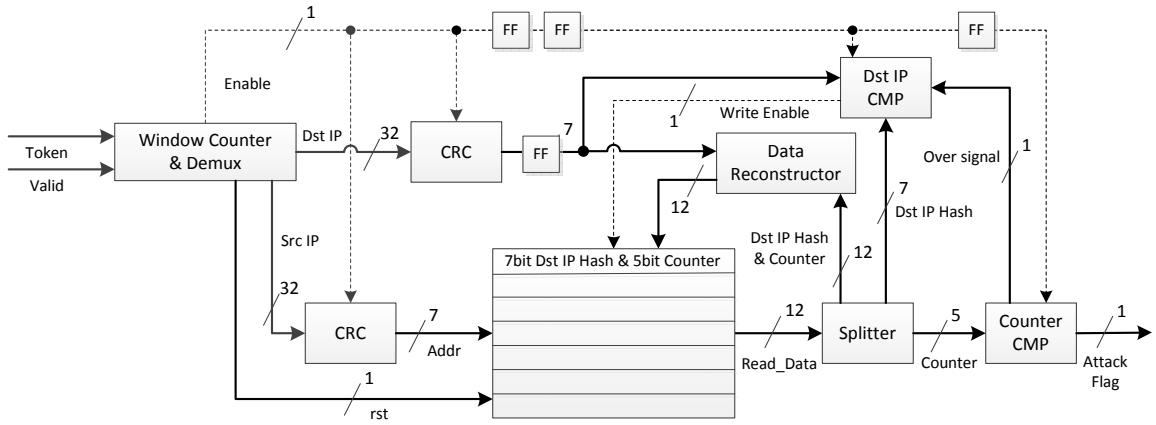


図 6.26: 動的 Attack Aware Cache のアーキテクチャ

てパケットが転送される事実を基に，あるパケット数の解析ウィンドウ内で得られた不正送信元 IP に対して，その解析ウィンドウがあるパケット数のウィンドウサイズに達するまでキャッシュ登録の拒否を行う．

解析ウィンドウ内では，送信元 IP アドレスを CRC ハッシュ化し得られた 7bit のハッシュ値をアドレスとするメモリへの読み書きを行う．本メモリはデータとして宛先 IP アドレスを CRC ハッシュ化し，7bit のハッシュ値としたものと，5bit のカウンタ値を保持している．まず，受信パケットの送信元 IP アドレスのハッシュ値をメモリアドレスとして読み出したデータを，Splitter モジュールにより，7bit のハッシュ値，5bit のカウンタ値，12bit のハッシュ値&カウンタ値の三つに分ける．DstIP Comparator モジュールでは受信パケットの宛先 IP アドレスのハッシュ値と，読み出したメモリデータのハッシュ値を比較し，異なっていた場合にメモリに対して write enable 信号を送る．Data Reconstructor モジュールでは，メモリから読み出した 12bit データのハッシュ値を受信パケットの宛先 IP アドレスのハッシュ値で置き換え，カウンタ値をインクリメントし，新たな 12bit データを構成しメモリへ出力する．メモリでは，DstIP Comparator モジュールからの write enable 信号を受けて，読み出したアドレスのデータを Data Reconstructor モジュールからのデータで書き換える．また，メモリから読み出されたデータのカウンタ値は Counter Comparator モジュールへと送られ，カウンタ値がある閾値を超えていた場合にアタックであると判断し Attack Flag を出力する．カウンタ値が閾値を超えると，over 信号が DstIP Comparator にも送られ，以降同じアドレスのデータが書き換えられることを防ぐ．パケット数が決められたウィンドウ数に達した時，リセット信号がメモリへと送られ，メモリの内容は全部無効となる．なお，本アーキテクチャの動作は 1 サイクルするために 4 クロックを必要とするが，直前に行われるキャッシュのヒット・ミス判定には 4 クロック以上の時間がかかるためキューを実装する必要はない．解析ウィンドウとしては 512 パケット程度，また，ウィンドウ内における攻撃元送信源の判断に用いる宛先 IP アドレスの変化回数は 32 回程度が適していることがシミュレーションによりわかっている．また，この時のメモリエントリ数は 128 程度あれば十分である．

本アーキテクチャは他の手法に比べ，複雑なハードウェア構成となっており，他の手法に比べ回路規模が大きい．Verilog により実装し論理合成した結果，組み合わせ回路規模が $3.84 \times 10^2 \mu m^2$,

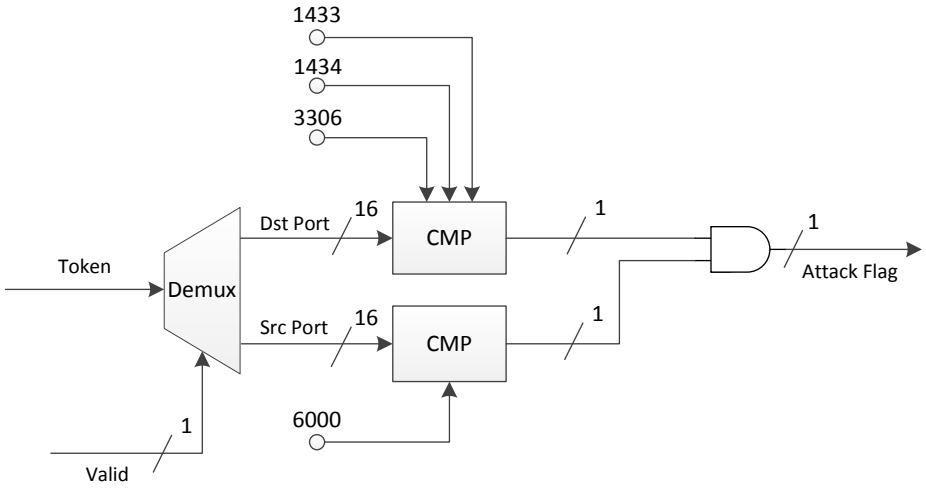


図 6.27: 静的 Attack Aware Cache のアーキテクチャ

フリップフロップ回路規模が $1.98 \times 10^2 \mu m^2$ となっており，マルチコンテクストキャッシュの中では最も組み合わせ回路規模が大きくなる．また，本アーキテクチャの実装にはメモリが必要となる．本メモリは 7bit のハッシュ値及び 5bit のカウンタ値を保存する．エントリ数を前述したように 128 と仮定すると，192Byte のメモリ容量が必要となる．192Byte のメモリや数百 μm^2 の回路規模は，ハードウェアコストとしては十分に小さいものであると言え，キャッシュ性能の向上効果を考慮すると本手法は実用的な手法であると言える．

6.4.3.3 静的 Attack Aware Cache の実装コスト

静的 Attack Aware Cache では，入力パケットのポート番号と 1433，1434，3306，6000 を比較するだけでよい．本手法のアーキテクチャを図 6.27 に示した．アタックを動的に判断することで回路規模の大きくなった動的 Attack Aware Cache と異なり，静的に判断する静的 Attack Aware Cache ではパケットから Demultiplexa モジュールにより送信元ポート番号と宛先ポート番号を抽出し，それぞれに対してポート番号を比較するだけで良いため，回路規模は小さい．表 6.5 に示す論理合成結果では，組み合わせ回路規模が $8.06 \times 10 \mu m^2$，フリップフロップ回路規模が $1.81 \times 10^2 \mu m^2$ となることがわかる．

6.4.3.4 Port Split Cache の実装コスト

本論文では，表 6.4 に示した各領域のエントリ構成をもとに Port Split Cache を実装した．本手法のアーキテクチャを図 6.28 に示した．各領域へのメモリ分割は，各領域に異なるインデックスを割り当てることで容易に実現できる．具体的には，4way のフローキャッシュでは DNS 領域に 0 から 3，HTTP 領域に 4 から 159，Well-known 領域に 160 から 191，非 Well-known 領域に 192 から 255 の値をインデックスとして割り当てることで，それぞれ 16 エントリ，624 エントリ，128 エントリ，256 エントリが確保される．

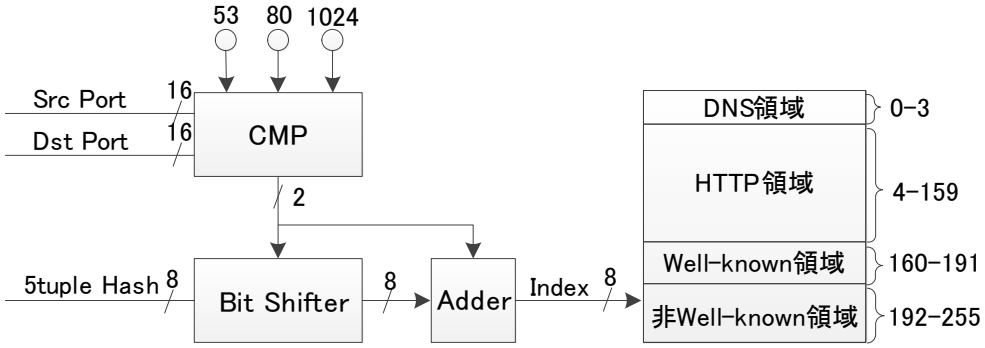


図 6.28: Port Split Cache のアーキテクチャ

まず，入力パケットの宛先ポート番号および送信元ポート番号が Comparator に転送され，DNS，HTTP，Well-known ポート，非 Well-known ポートの判断がなされる．結果として，各領域への分類を示す 2bit の値が得られる．この 2bit の分類値と，5 タプルのハッシュ値は Bit Shifter へと転送される．

Bit Shifter は，5 タプルの 8bit ハッシュ値を 2bit の分類値によって異なる bit 数シフトした後，8bit に整え Adder へと転送する．分類値が DNS を示す場合は，ハッシュ値を 6bit 右にシフトすることで，2bit の値が得られる．Well-known を示す場合は，ハッシュ値を 3bit 右にシフトすることで，5bit の値が得られる．同様に，非 Well-known 領域を示す場合は，ハッシュ値を 2bit 右にシフトすることで，6bit の値が得られる．HTTP 領域の場合はインデックスの範囲が 156 であり，8bit を要するため，シフトせずハッシュ値をそのまま転送する．

Adder では，Bit Shifter で得られた 8bit の値と分類値から，キャッシュのインデックスを生成する．具体的には，分類値が DNS を示す場合には，入力値がそのままインデックスとなる．分類値が Well-known ポートを示す場合は，入力値に 160 を加えた値をインデックスとする．分類値が非 Well-known ポートを示す場合は，入力値に 192 を加えた値をインデックスとする．ここで，分類値が HTTP である場合は，エントリ数が 2 の累乗とならないため，他の領域とは異なった処理が行われる．まず，入力値が 156 以上であるかどうか判断される．入力値が 156 より小さい場合は，入力値に 4 を加えた値をインデックスとする．一方で，入力値が 156 以上である場合は，入力値と 156 の差分に 4 を加えた値をインデックスとする．このような処理では 5 タプル値によっては 4 から 104 のインデックス値が重複する．従って，HTTP フローのエントリ利用効率が多少低下するが，再度 CRC ハッシュ値を計算する手間などを考えると，本手法が適していると考えられる．

本アーキテクチャを実装した結果，表 6.5 に示したように，回路規模は組み合わせ回路として $2.12 \times 10^2 \mu m^2$，フリップフロップ回路として $1.85 \times 10^2 \mu m^2$ であり，比較的実装コストは小さいことがわかった．前節で示した平均 7.45%のキャッシュミス削減効果と併せて考えると，実装コストが小さい分，本手法のパフォーマンスは $1.88 \times 10^{-2}\%/\mu m^2$ と他の手法と比べても高い．

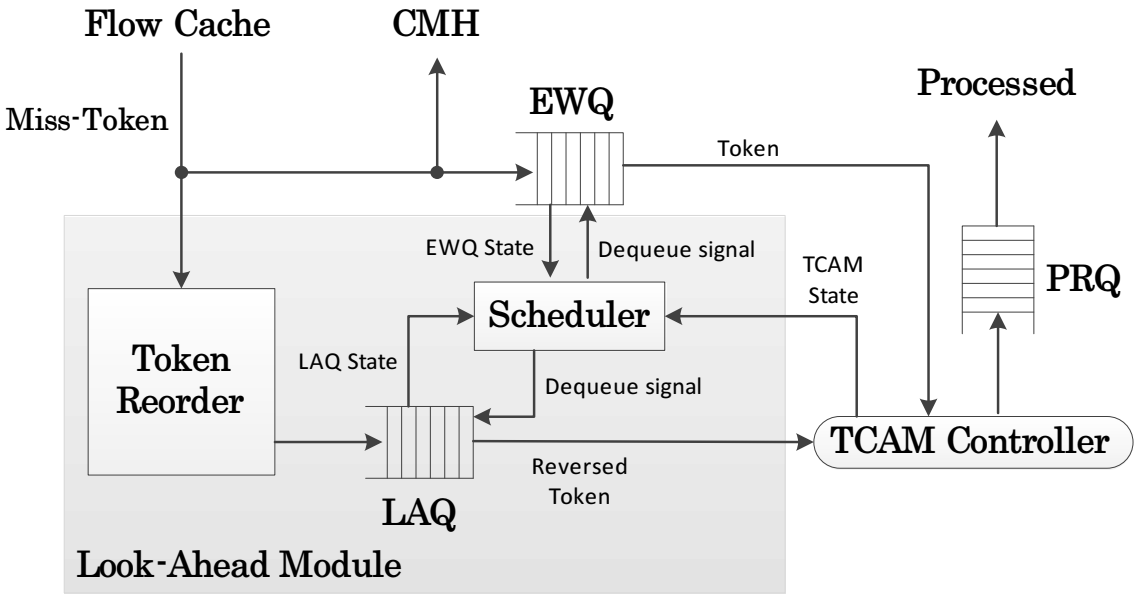


図 6.29: Look Ahead Cache のアーキテクチャ

6.4.3.5 Look Ahead Cache の実装コスト

Look Ahead Cache のハードウェア実装方法について検討を行った．図 6.29 に，実装したアーキテクチャを示す．本手法は前節で検討したように，フローキャッシュのミス時に，キャッシュミスしたフローのテーブル検索と併せて逆方向フローの処理も行いキャッシュする．そこで，Token Reorder モジュールにより逆方向フロー用トークンを生成し，生成されたトークンを Look Ahead Queue（LAQ）へと格納する．逆方向フローは，応答パケットが到着するまでに処理できれば良く，本処理によって通常のパケット処理のパフォーマンスが低下することのないよう TCAM の処理負担が軽い時を選んで処理を行う．そのため，各キューと TCAM の空き状態を Scheduler が管理し，TCAM が使用されておらず，なおかつ EWQ に格納されているトークンが存在しない時に LAQ へデキュー命令を出すことでこれを実現する．

本ハードウェアシミュレーションでは図 6.29 中の Look Ahead Module 部分の回路を記述し論理合成を行った．Look Ahead Module は組み合わせ回路の Token Reorder モジュールと Scheduler モジュール，フリップフロップ回路となる LAQ により構成されており，論理合成結果は表 6.5 に示したように，組み合わせ回路規模が $7.95\times10\mu m^2$，フリップフロップ回路規模が $6.98\times10^2\mu m^2$ となった．また，キャッシュミス多発時の過度な負荷増加を避けるため，LAQ は 1KByte 程度のリングバッファを想定した．本手法によるキャッシュミスの削減効果は非常に高いが，ハードウェアコストも大きいため，$1\mu m^2$ あたりのキャッシュミス削減効果は表 6.6 に示すように $1.94\times10^{-2}\ \%/\mu m^2$ となっており，Hit Priority Cache に次いでパフォーマンスが高い．また，前述したように Look Ahead Cache で削減されるミスは初期参照ミスであるため，本手法の価値は高い．

6.4.3.6 マルチコンテクストキャッシュの実装コスト

最後に，全ての提案手法を組み合わせたマルチコンテクストキャッシュの実装コストについて評価した．マルチコンテクストキャッシュの実装コストは，これまで述べてきた全ての提案手法の実装コストを合わせることで概算できる．そこで，キャッシュミスの削減率および回路規模，そこから計算できるパフォーマンス値を表 6.6 にまとめた．マルチコンテクストキャッシュの回路規模は $2.19 \times 10^3 \mu m^2$ であり，小規模である．それに対して，キャッシュミスの削減率は 32.6%であり，フローキャッシュに本手法を適用することで，低回路コストなハードウェアの追加で大幅なキャッシュミスを削減できることが示された．

表 6.5: ASIC を対象とした論理合成の評価結果

	Hit Priority Cache	動的 Attack Aware Cache	静的 Attack Aware Cache	Port Split Cache	Look Ahead Cache
組み合わせ回路規模	9.25×10 μm^2	3.85×10^2 μm^2	8.06×10 μm^2	2.12×10^2 μm^2	7.95×10 μm^2
フリップフロップ回路規模	2.23×10^2 μm^2	1.98×10^2 μm^2	1.81×10^2 μm^2	1.85×10^2 μm^2	6.98×10^2 μm^2
最大動作遅延	0.39 ns	0.39 ns	0.32 ns	0.42 ns	0.44 ns
最大動作周波数	2.56 GHz	2.56 GHz	3.13 GHz	2.38 GHz	2.27 GHz
メモリ容量	+1bit/entry	192 Byte	0 Byte	0 Byte	1K Byte

表 6.6: 各手法の回路規模に対するキャッシュミス削減効果

名称	ミスの削減率 [%]	回路 [μm^2]	性能 [%/μm^2]
Hit Priority Cache	17.9	3.16×10^2	5.66×10^{-2}
動的 Attack Aware Cache	（全体平均）−0.0485	5.83×10^2	-8.32×10^{-5}
	（アタック時）4.25		7.29×10^{-3}
静的 Attack Aware Cache	（全体平均）0.425	2.62×10^2	1.62×10^{-3}
	（アタック時）5.53		2.11×10^{-2}
Port Split Cache	7.45	3.97×10^2	1.88×10^{-2}
Look Ahead Cache	15.1	7.78×10^2	1.94×10^{-2}
マルチコンテクストキャッシュ	（全体平均）32.6	2.19×10^3	1.49×10^{-2}
	（アタック時）18.0		8.25×10^{-3}

6.4.4 検索性能およびスループットに関する評価

これまで，マルチコンテクストキャッシュを用いることでキャッシュミスが削減できることを述べてきた．本項では，キャッシュミスの削減がテーブル検索の処理性能やスループットに与える影響について定量的に評価する．

まず，従来の TCAM 方式によるテーブル検索の処理性能およびスループットを概算する．テーブル検索のスループットは，メモリの検索性能により決定される．メモリの検索性能は，レイテンシの逆数をとることで計算できる．一般的に，メモリの検索性能の単位は sps（search per second）や，ルータにおいては pps（packet per second）が用いられる．本論文では pps を用いる．最短パケット長 64Byte のパケット処理を想定すると，TCAM 方式のテーブル検索処理スループット T_{tcam} は，TCAM の検索性能 S_{tcam} を用いて (7.1) 式により計算できる．ここで，論文 [77] を参考に TCAM のレイテンシを 5ns とすると，TCAM の検索性能は 200Mpps，テーブル検索スループットは 95Gbps となる．

$$T_{tcam} = S_{tcam} \times 64\text{Byte} \tag{6.1}$$

次に，フローキャッシュと TCAM を併用したテーブル検索における処理性能およびスループットについて検討する．この方式では，テーブル検索をキャッシュと TCAM の 2 ステージで行うため，どちらかが性能ボトルネックとなる．キャッシュが全パケットからアクセスされるのに対し，TCAM はキャッシュミス時にのみアクセスがなされる．従って，キャッシュミス率が高い場合には TCAM が，低い場合にはキャッシュがボトルネックになると考えられる．上記を踏まえ，フローキャッシュを用いたテーブル検索全体における検索性能 $S_{flowcache}$ およびスループット $T_{flowcache}$ を (7.2)，(7.3) 式にそれぞれ表した．

$$S_{flowcache} = \begin{cases} S_{sram} & (S_{sram} \times Miss\ Rate \geq S_{tcam}) \\ \dfrac{S_{tcam}}{Miss\ Rate} & (S_{sram} \times Miss\ Rate < S_{tcam}) \end{cases} \tag{6.2}$$

$$T_{flowcache} = S_{flowcache} \times 64\text{Byte} \tag{6.3}$$

表 6.1 を参考に 32KB SRAM のレイテンシを 1.08ns として検索性能およびスループットを計算した結果を表 6.7 に示した．表 6.7 では，TCAM のみを用いた場合（TCAM のみ），32KB フローキャッシュを併用した場合（Cache(Normal)），更にマルチコンテクストキャッシュを適用した場合（Cache(MCC)）のそれぞれの検索性能およびスループットを概算した．TCAM の検索性能は 200Mpps だが，表では各手法を用いることで処理できる全体的なパケットレートを示している．また，フローキャッシュと TCAM において低いほうのスループットを太字で示している．

従来の TCAM 方式によるテーブル検索性能は 200Mpps 程度であり，最短パケット長のパケット処理を想定した場合，処理スループットは 95Gbps 程度であった．アプリケーションルータでは更にテーブル操作が頻繁に行われるため，100Gbps ネットワークの処理を想定すると，TCAM 方式によるテーブル検索性能は十分であるとは言えない．これに対し，フローキャッシュを併用することで平均 370Gbps のスループットが得られた．更に，フローキャッシュにマルチコンテクストキャッシュを適用した場合には，442Gbps が得られた．これは，近年開発の進められている 400Gbps ネットワークのパケット処理であっても，アプリケーションルータ内で 10 パケットに 1

表 6.7: テーブル検索処理スループットの概算結果

		TCAM のみ	Cache(Normal) + TCAM	Cache(MCC) + TCAM
Cache	検索性能	-	926 Mpps	**926 Mpps**
	スループット	-	442 Gbps	**442 Gbps**
TCAM	検索性能	**200 Mpps**	**775 Mpps**	1149 Mpps
	スループット	**95 Gbps**	**370 Gbps**	548 Gbps

回のテーブル更新を十分に処理可能な性能である．

6.4.5 消費電力量に関する評価

従来のルータにおけるパケット処理では，3.2 節で述べたように，ルーティングテーブルや ARP テーブル，ACL，QoS テーブルなど，複数のテーブル検索を要する．フローキャッシュを用いることで，これらのテーブル検索をたった一度のメモリアクセスにより解決することができる．この場合，複数回のメモリアクセスを一度に省略できる他，消費電力の高い TCAM の使用頻度が軽減されるため，消費電力量が改善される．そこで，本節では，消費電力量の削減効果について評価する．

まず，消費電力量について検討する．論文 [4, 77, 142] を参考にすると，TCAM や SRAM のアイドリング時の消費電力は，エントリ検索時に比べ無視できる程度に小さい．そこで，各メモリの電力消費はエントリ探索時のみとし，1 検索あたりの消費電力量をそれぞれ P_{tcam} と P_{sram} とする．ここで，パケット処理におけるテーブル検索をルーティングテーブルと ARP テーブル，ACL，QoS テーブルの 4 種類とすると，従来の TCAM 方式によるエントリ検索時の消費電力量 $P_{conventional}$ は (7.4) 式により概算することができる．一方で，フローキャッシュを併用した場合には，SRAM へ毎パケットアクセスが行われ，ミス時にのみ TCAM が使用されることから，平均消費電力量 $P_{flowcache}$ は (7.5) 式で表される．

$$P_{conventional} = P_{tcam} \times 4 \tag{6.4}$$

$$P_{flowcache} = P_{sram} + P_{conventional} \times Miss\ Rate \tag{6.5}$$

消費電力量を概算するにあたって，キャッシュおよび TCAM の消費電力量が必要となる．そこで，キャッシュおよび TCAM の消費電力量を，キャッシュメモリシミュレータ CACTI[143, 144] および論文 [142] をもとに見積もった．CACTI は，Hewlett-Packard 社の提供するキャッシュおよびメモリのアクセスタイム，サイクルタイム，エリア，リーク電流等をモデル化したソフトウェアであり，キャッシュに関する研究において広く利用されてきた [4, 142, 145]．CACTI では，メモリサイズやブロックサイズ，タグサイズ，メモリの種類，連想度，CMOS プロセスといった様々

なパラメータを設定することで，メモリの消費電力やエリアコスト，遅延などを見積もることができる．CACTI のバージョンは日々更新されており，2015 年 12 月現在ではバージョン 6.5 が公開されている．バージョン 6.5 では 32nm から 90nm までの CMOS プロセスによる見積りが可能となっている．CACTI を用いて，32KB フローキャッシュの 1 検索あたりの消費電力量 P_{sram} を見積もった結果，0.121nWs/search となった．なお，メモリ構成は 28Byte/エントリ，1k エントリ，4way セットアソシアティブ，90nm プロセスとした．この時のレイテンシの見積り結果は 1.11ns であり，表 6.1 に示した L1 キャッシュのレイテンシに近い値となる．

CACTI は TCAM に対応しておらず，そのままでは TCAM の消費電力量を見積もることはできない．そこで，TCAM の消費電力量を論文 [142] を参考に見積もった．論文 [142] では TCAM の様々なパラメータの測定が行われており，36Byte/エントリ，32k エントリ，180nm プロセスの SRAM と TCAM の消費電力量が比較されている．これによると，1 検索あたりの消費電力量は TCAM が 16.7nWs であるのに対し，SRAM は 1.9nWs であり，TCAM は SRAM の 8.7 倍の消費電力量となる．この比率をもとに，SRAM の消費電力量から TCAM の消費電力量を見積もる．ルータに用いられる 20Mbit の TCAM を想定し，20Mbit の SRAM を 90nm プロセスで見積もった結果，SRAM の 1 検索あたりの消費電力量は 0.640nWs/search となった．従って，TCAM の場合の消費電力量 P_{tcam} は 5.57nWs 程度であると予想される．

これらの値をもとに，従来の TCAM 方式と，フローキャッシュの併用によるハイブリッド方式の消費電力量を概算した結果を表 6.8 に示した．テーブル検索を TCAM のみで行った場合は，1 パケットあたり 22.3nWs の電力量がかかるが，フローキャッシュを用いた場合には 5.87nWs となり，73.7%の消費電力量の削減が期待できる．更にマルチコンテクストキャッシュを用いた場合には，4.00nWs まで削減可能であり，82.1%の消費電力の削減が期待できる．このことから，マルチコンテクストキャッシュを適用したフローキャッシュを用いることで，ルータのテーブル検索における消費電力量を大幅に削減できることが示された．

表 6.8: 消費電力量の概算結果

	TCAM のみ	Cache(Normal) + TCAM	Cache(MCC) + TCAM
平均消費電力量	22.3 nWs/packet	5.87 nWs/packet	4.00 nWs/packet

6.5 本章のまとめ

本章では，パケット処理におけるテーブル検索に焦点を当て，近年の TCAM 方式にフローキャッシュを併用することで高スループットかつ低消費電力にテーブル検索を行う手法を提案した．キャッシュにおける容量とレイテンシのトレードオフの関係から，テーブル検索にはプロセッサの L1 キャッシュ程度の小規模だが高速なメモリを用いることが望ましい．一方で，低容量のキャッシュはキャッシュミスの増大を招く．そこで，本論文では，トラフィックの特徴を基にキャッシュエントリを適切に制御することで，容量を増やすことなくミスを削減するマルチコンテクストキャッ

シュを提案した．

まず，キャッシュエントリ登録時の追い出し優先度とヒット時の追い出し優先度の遷移を適切に制御する Hit Priority Cache を提案した．本手法により，一般的に用いられる LRU と比べ 17.9%のミスが削減できることを示した．Hit Priority Cache は LRU と同様の実装が可能であり，回路として 316μm^2，メモリとして 1bit/entry を追加することで実現できる．

次に，アタック時におけるキャッシュミスの急激な増大を抑制する手法として Attack Aware Cache を提案した．まず，IP アドレスの統計を基に動的にアタックを判断する動的 Attack Aware Cache を提案し，シミュレーションによってアタック時のミスが 4.25%削減できることを示した．本手法の実装コストは 583μm^2 と提案手法の中でも比較的大きいが，未知のアタックに対しても有効に働くことが期待できるため有用である．もう一つの手法として，アタックに多用されるポート番号を用いて，特定ポート番号のキャッシュ登録を拒否する静的 Attack Aware Cache を提案した．本手法により，アタック時のキャッシュミスを 5.53%削減できることを示した．本手法のハードウェア実装は容易であり，262μm^2 程度の回路規模で実装が可能である．

また，フロー分類毎にキャッシュすべきメモリ領域を別個に与える Port Split Cache を提案した．Port Split Cache では，キャッシュを DNS 領域，HTTP 領域，Well-known 領域，非 Well-known 領域の 4 つに分割し，それぞれに適切なエントリ数を与えることで，7.45%のミスを削減できることが示された．Port Split Cache の実装コストは 397μm^2 であり，小規模な回路により実装できる．

フローキャッシュにおける初期参照ミスの削減に焦点を当て，応答フローのキャッシュエントリを予め登録する Look Ahead Cache を提案した．Look Ahead Cache を適用することで，15.1%程度のミスが削減できることを示した．Look Ahead Cache の実装コストは，778μm^2 と提案手法の中で最も大きいが，Look Ahead Cache により削減されるミスは他の手法と性質が異なることから，本手法を実装する意義は大きい．

提案した全ての手法を最適に組み合わせたマルチコンテクストキャッシュでは，通常時に 32.6%程度，アタック時でも 18.0%程度のミスが削減できることを示した．回路規模は 2190μm^2 であり，マルチコンテクストキャッシュの実装コストは小さい．また，マルチコンテクストキャッシュを適用した際のテーブル検索処理スループットは 442Gbps 程度であり，TCAM のみの従来手法に比べ 4.65 倍，何も適用しない場合のフローキャッシュと比べても 1.20 倍のスループットを獲得できることが示された．また，消費電力の点から見ても，マルチコンテクストキャッシュを適用することで，TCAM のみの場合と比べ 17.9%，何も適用しない場合のフローキャッシュと比べ 68.1%の消費電力によりテーブル検索処理を完了できることを示した．

第7章　結論

本論文では，近年のネットワークの高度化に伴う高機能なサービスへの需要の高まりに対し，ネットワーク経路上のパケットをレイヤ 7 まで含めて解析し，その結果を用いてアプリケーションを提供するアプリケーションルータが広く着目されていることを第 1 章および第 2 章で述べた．従来のアプリケーションルータは，レイヤ 7 のペイロードまで含めた情報抽出を，パケット断片化を考慮して行う場合，その処理の複雑さから，1Gbps 程度を達成することが限界であり，広帯域ネットワークへの適用は困難であった．更に，パケット転送処理の観点から見ると，アプリケーションルータでは情報抽出結果をもとにルーティングテーブルや ACL といった各種テーブルエントリが追加，更新されることで，テーブル検索の処理負荷が増大する．従来のルータにおけるテーブル検索は消費電力の高い TCAM を潤沢に用いており，消費電力の増大が問題となっていることからも，これ以上の TCAM 負担の増加は避けれなればならない．このように，アプリケーションルータの広帯域ネットワークへの対応において，情報抽出およびテーブル検索それぞれの改善が必要不可欠であった．そこで，本論文では，100Gbps ネットワークを目標としたアプリケーションルータにおける情報抽出およびテーブル検索手法を提案した．

まず，第 5 章では，近年普及が進んでいる HTTP データの GZIP 圧縮化に対し，そのままでは情報抽出ができないことから，ネットワーク経路上で逐次に HTTP 圧縮パケットの GZIP 展開を行う高速なハードウェアアーキテクチャを提案し，実ネットワークトレースを用いて評価した．パケット断片化に対応するためのコンテクストスイッチ機構，GZIP 展開処理高速化のためのキャッシュおよびモジュール並列化を用いた本アーキテクチャによって，符号表作成モジュール数 1，復号モジュール数 4 とした場合に 6.37Gbps のスループットが得られることを示した．これは，100Gbps ネットワーク環境における GZIP 展開のワイヤレート処理に必要となる 5.5Gbps のスループットを十分に達成している．更に，今後の GZIP 圧縮の普及を想定し，トラフィックに占める GZIP 圧縮データの割合が現在の 2 倍となる 11%になったとしても，符号表作成モジュール数 2，復号モジュール数 8 のハードウェア構成により 11Gbps が達成できることを示した．本アーキテクチャは，モジュール並列度を増やすことで更にスループットを向上させることが可能であり，符号表作成モジュール数 3，復号モジュール数 20 とした場合には 25Gbps 程度のスループットが獲得できる．これは，GZIP 圧縮の普及率が現在のままであるならば，400Gbps 程度までは対応可能であることを示す．更に，本論文では，複数台のアプリケーションルータがネットワークに存在する場合に，ピギーバックパケットを用いることで，アプリケーションルータ間で重複する符号表作成処理をバイパスさせる手法を提案した．本手法によって，アプリケーションルータが複数台存在するネットワーク環境では，GZIP 展開処理のスループットが 1.3 倍程度向上できることを示した．

次に，第 6 章では，アプリケーションルータの情報抽出機構の中核を担う文字列探索処理が性能ボトルネックとなっていることから，Rabin-Karp 法をもとに，アプリケーションルータの抽出

ルールに基づいて文字列探索処理負荷をオフロードするハードウェアアーキテクチャを提案し評価した．本手法を用いることで，Rabin-Karp 法を従来の 36.0%程度のハードウェアコストで実装でき，なおかつ文字列探索処理負荷を 96%削減できることを示した．これは即ち，100Gbp トラフィックに本手法を適用した場合，文字列探索に求められるスループットを 4Gbps まで削減できることを示している．近年の文字列探索技術はハードウェア ASIC や FPGA，メニーコア技術を用いることで 8Gbps 程度の性能が得られることから，本手法と併せることで，200Gbps 程度のネットワーク環境ならばワイヤレートに文字列探索が行える．本論文では，単純な文字列探索処理に焦点を当てたが，将来的には正規表現を含むより複雑な情報抽出を行えることが望ましい．そこで，本ハードウェアアーキテクチャの正規表現への対応が今後の課題となる．

第 7 章では，アプリケーションルータにおけるテーブル検索処理負荷の増大によってテーブル検索処理が高消費電力化し，なおかつ性能ボトルネックとなることから，近年の TCAM を用いたテーブル検索手法にフローキャッシュ機構を併せて実装した，高速かつ低消費電力なテーブル検索手法を提案した．本論文では，更に，フローキャッシュのテーブル検索性能を決定づけるキャッシュミスを改善するために，ネットワークトラフィックの特徴を用いてキャッシュエントリの登録，追い出しを最適化するマルチコンテクストキャッシュを提案し，実ネットワークトレースを用いて評価した．マルチコンテクストキャッシュを用いることで，キャッシュミスを平均して 32.6%程度改善でき，なおかつ，アタック時の急激なキャッシュミスの増加を 18.0%程度抑えられることを示した．これによって得られるテーブル検索スループットは 442Gbps であり，マルチコンテクストキャッシュを適用しない場合と比べ 1.20 倍，TCAM のみを用いた従来手法と比べ 4.65 倍のスループットが得られることを示した．更に，消費電力量の点から見ても，マルチコンテクストキャッシュを適用したテーブル検索は 1 エントリあたり平均 4.00nWs であり，マルチコンテクストキャッシュを適用しない場合と比べ 68.1%，TCAM のみの従来手法と比べ 17.9%の消費電力によって処理できることを示した．また，マルチコンテクストキャッシュ全体の実装回路規模は 2190μm^2 程度であり，キャッシュや TCAM に比べ小さいハードウェアコストによって実装できる．提案手法によって，既存の TCAM 方式に変更を加えることなく，100Gbps ネットワーク環境におけるアプリケーションルータのテーブル検索処理に対応することが可能となり，更には従来のテーブル検索の問題点であった消費電力の増大を解決できることを示した．

従来のアプリケーションルータにおいて，パケット断片化を考慮し，なおかつ GZIP 展開まで対応した場合に得られるスループットは数 Gbps 程度であった．これに対し，本論文では，アプリケーションルータの性能ボトルネックを検討し，100Gbps のワイヤレート処理の実現において GZIP 展開処理，文字列探索処理，テーブル検索処理の 3 つの処理が問題であることから，それぞれを解決するアーキテクチャを提案し，100Gbps 以上の実効スループットを得た．これにより，100Gbps コアネットワークであってもアプリケーションルータにおいてワイヤレートにパケットの転送および情報抽出を行うことが可能となった．

本論文の意義は，今後のネットワークの動向に注目した上でも大きい．近年，ネットワーク技術は従来の電気ベースによる情報の伝送から，光ベースへと移り変わりつつある．それに伴い，光によってパケット転送処理を行う光スイッチや，データの転送経路を光の波長によって決定する光スイッチング技術が提案されている．初期に提案された光回線交換スイッチングと呼ばれる技術では，まず通信の端点において通信経路を決定し，その経路に対応した波長の光へデータを変

換することで，ルータが処理することなく光の波長を基に宛先へとデータが転送される．しかしながら，光回線交換スイッチングでは，通信路上においてデータに介入する術がなく，QoS，カプセリングといった需要の増加しているプロセッシング機能を提供することが困難となる．従って，光回線交換スイッチングは今後のネットワークへのニーズに一致しないと考える．一方で，近年研究の進められている光パケットスイッチング技術は，従来の電気素子によるルータアーキテクチャを光により実現することを目標としている．この技術では，従来のクロスバースイッチやバッファを光化した光スイッチや光バッファが提案されているが，パケットペイロードの処理やヘッダのラベル処理といったプロセッシング機能に関しては従来の電気処理によって行われるのが一般的である [146]．このことからも，今後のネットワーク技術の光化を想定しても，ルータのプロセッシング機能に関しては本論文の提案手法が有効であると考えられる．

今後の発展としては，まず，アプリケーションルータに各提案機構を組み合わせた際の，実装の最適化を検討したい．本論文では，フローキャッシュやコンテクストキャッシュ，コンテクスト管理メモリといった，パケットの 5 タプル情報をタグとして用いるメモリ機構が多用されている．これらのメモリはまとめることが可能であると考えられる．例えば，フローキャッシュとコンテクストキャッシュを一つにまとめ，L1 キャッシュとして用い，コンテクスト管理メモリにテーブル検索結果を追加して L2 キャッシュとして用いることで，メモリ資源の有効活用が可能となる．このような，アプリケーションルータにおける処理の最適化は，まだ検討の余地が残っている．次に，更なるスループットの向上が考えられる．本論文で提案した各アーキテクチャは最低でも 200Gbps 程度の実効スループットが得られている．従って，今後，アプリケーションルータが 100Gbps 以上のネットワークに対応するためには，第 3 章の表 3.18 を参考にすると，データベースエンジンの高速化が課題となると考えられる．

また，アプリケーションルータのサービスについても検討する必要がある．第 2 章では，アプリケーションルータにより実現される可能性のある様々なサービスを紹介したが，NIDS 以外のサービスに関しては，まだまだ既存研究が不十分であり，実用には及んでいない．そこで，第 2 章で紹介したサービスを実装し，実際のネットワークにおいて運用を試みる必要があると考える．更に，我々が将来的に目指すのは，アプリケーションルータに蓄えられた情報を，共通の API のもとにユーザが自由に加工し，ユーザ自身が新たなサービスを作り出すオープンなネットワークの実現である．近年，ネットワークを流れる情報はビッグデータとして，その価値を高く評価されているが，このような有用なデータは全てのユーザにとって利用可能であることが望ましい．そのためには，ユーザが抽出ルールの決定やデータベースに蓄えられた情報の取得を可能とするための共通 API，抽出された情報に対しプライバシーを特定できないように匿名化した上でデータベースに保存する機構，複数のアプリケーションルータ間でデータベースの共有を行う分散データベース機構など，様々な機能を実現する必要がある．このような機能をアプリケーションルータに実装し，複数台のアプリケーションルータによるサービス指向なネットワークを構築することが今後の目標である．

最後に，アプリケーションルータと電気通信事業法との関わりについて意見を述べる．アプリケーションルータによるサービスの提供は，通信内容にまで踏み込んで情報抽出を行うことから，電気通信事業法 4 条の「通信の秘密」に触れていると判断され，違法性を指摘されることが多い．現状ではこのような法律的観点から，アプリケーションルータを導入するネットワークでは，予め

ユーザに承認を得るオプトイン方式を採らなければならない．電気通信事業法は，インターネットが大々的に普及する以前の 1984 年に制定された，主に電話の取り決めを目的とした法律である．従って，同法 4 条の「通信の秘密」は，電話に対する傍受を禁止する目的が大きい．インターネットも電気通信事業法により取り決められているが，本論文の第 1 章，第 2 章でも述べたように，現在のインターネットは初期のインターネットと異なり，様々な用途で利用されるようになってきた．このことから，電話とインターネットを同一の法律で扱うべきではないと考える．アプリケーションルータの出現は，このような，インターネットとそれ以前の電気通信との役割の違いを明確化し，見直す機会となる．まず，オプトイン方式によって，アプリケーションルータの提供する様々なサービスの有用性と安全性を示すことが，その一歩になると考える．本論文によって，アプリケーションルータが今後より広く有効に用いられることを期待する．

謝辞

本論文の主査を快く引き受けてくださり，また，本論文の礎となるインターネットアーキテクチャをはじめとする IT 基盤の基礎から応用まで深いレベルで絶えずご指導いただいた慶應義塾大学理工学部システムデザイン工学科 西 宏章教授に深く感謝いたします.

本論文の副査を快く引き受けてくださり，数多くの有益なアドバイスを頂いた慶應義塾大学理工学部情報工学科 山中 直明 教授，同情報工学科 重野 寛 教授，同情報工学科 松谷 宏紀 専任講師，国立情報学研究所 鯉渕 道紘 准教授に深く感謝いたします.

そして最後に，これまでずっと支えてくれた両親と兄，友人に感謝します.

2016 年 2 月 慶應義塾大学理工学部西研究室にて

参考文献

[1] 総務省. 我が国のインターネットにおけるトラヒックの集計・試算. http://www.soumu.go.jp/menu_news/s-news/01kiban04_02000042.html.

[2] 石田慎一, 原島真吾, 鯉渕道紘, 川島英之, 西宏章. トラヒックからアプリケーションレイヤ情報の検索・抽出を可能とするソフトウェアの実装と評価. コンピュータソフトウェア, Vol. 29, No. 4, pp. 59–73, oct 2012.

[3] B. Talbot, T. Sherwood, and B. Lin. Ip caching for terabit speedrouters. In *Global Communication Conference (Globecom'99)*, pp. 1565–1569, 1999.

[4] G. Liao, H. Yu, and L. Bhuyan. A new ip lookup cache for high performance ip routers. In *DAC'10 Proceedings of the 47th Design Automation Conference*, pp. 338–343, 2010.

[5] N. Kim, S. Jean, J. Kim, and H. Yoon. Cache replacement schemes for data-driven lable switching networks. In *High Performance Switching and Routing, 2001 IEEE Workshop on*, pp. 223–227, 2001.

[6] M. Okuno, S. Ishida, and H. Nishi. Low-Power Network-Packet-Processing Architecture Using Process-Learning Cache for High-end Backbone Router. *IEICE Trans. Electron.*, Vol. E88-C, No. 4, pp. 536–543, Apr. 2005.

[7] M. Okuno and H. Nishi. Network-Processor Acceleration-Architecture Using Header-Learning Cache and Cache-Miss Handler. In *Proceedings of The 8th World Multi-Conference on Systemics, Cybernetics and Informatics, SCI2004*, Vol. III, pp. 108–113, Jul. 2004.

[8] Sandy bridge がやってきた！プロセッサの基本性能は順当に向上. http://www.5gamer.net/games/098/G009883/20110102001/, January 2011.

[9] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng. Fpga implementation of gzip compression and decompression for idc services. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 265–268, 2010.

[10] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel. Hardware acceleration in the ibm poweren processor: Architecture and performance. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 389–400, New York, NY, USA, 2012. ACM.

[11] 西野清志. ネットワーク境界のワークロード処理へのチャレンジ - the ibm power edge of networktm processor. In *Provision*, 67 巻, pp. 58–63, 2010.

[12] M. Akil, L. Perroton, and T. Grandpierre. Fpga-based architecture for hardware compression/decompression of wide format images. *Journal of Real-Time Image Processing*, Vol. 1, No. 2, pp. 163–170, 2006.

[13] C.-T.D. Lo, Y.-G. Tai, and K. Psarris. Hardware implementation for network intrusion detection rules with regular expression support. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pp. 1535–1539, New York, NY, USA, 2008. ACM.

[14] 総務省. 電気通信サービスの契約数及びシェアに関する四半期データの公表（平成24年度第1四半期（6月末））. http://www.soumu.go.jp/menu_news/s-news/01kiban04_02000044.html.

[15] 総務省. インターネット接続サービスの利用者数等の推移 [平成 14 年 7 月末現在]（速報）. http://www.soumu.go.jp/menu_news/s-news/daijinkanbou/020902_2.pdf.

[16] IEEE802.3ba Task Force. http://grouper.ieee.org/groups/802/3/ba/index.html.

[17] K. Ashton. That ' internet of things ' thing. *RFiD Journal*, Vol. 22, No. 7, pp. 97–114, 2009.

[18] 総務省. 情報通信白書平成 27 年版. http://www.soumu.go.jp/johotsusintokei/whitepaper/h27.html.

[19] P. Borgnat, G. Dewaele, K. Fukuda, P. Abry, and K. Cho. Seven years and one day: Sketching the evolution of internet traffic. In *INFOCOM 2009, IEEE*, pp. 711–719, April 2009.

[20] V. Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, Vol. 31, No. 23-24, pp. 2435–2463, December 1999.

[21] H.J. Wang, C. Guo, D.R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pp. 193–204, New York, NY, USA, 2004. ACM.

[22] M. Hanaoka, M. Shimamura, and K. Kono. Tcp reassembler for layer7-aware network intrusion detection/prevention systems (dependable computing). *IEICE transactions on information and systems*, Vol. 90, No. 12, pp. 2019–2032, dec 2007.

[23] 総務省｜平成 24 年版 情報通信白書. http://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h24/html/nc121410.html.

[24] 経済産業省. ルーター・スイッチの現状. http://www.meti.go.jp/committee/materials/downloadfiles/g80221b03j.pdf.

[25] Juniper Networks. http://www.juniper.net/jp/jp/products-services/routing/t-tx-series/.

[26] Juniper Networks. Juniper Networks Introduces the Next-Generation Service-Aware Core Network(Press Release), 2007.

[27] Juniper Networks. *T Series Core Routers Architecture Overview (White Paper)*, 2010.

[28] Juniper Networks. Juniper Demonstrates Industry's First Live 100G Traffic from the Network Core to Edge and Unveils Next-Generation Core Router(Press Release), 2010.

[29] Cisco Systems. `http://www.cisco.com`.

[30] Cisco Systems. *Cisco CRS-3 16-Slot Single-Shelf System(Data Sheet)*, 2010.

[31] AlaxalA Networks Corporation ハイエンドルータ AX7800R シリーズ. `http://www.alaxala.com/jp/products/AX7800R/index.html`.

[32] AlaxalA Networks Corporation ハイエンドルータ AX8600R シリーズ. `http://www.alaxala.com/jp/products/AX8600R/index.html`.

[33] White paper: Cisco 4000 series integrated services routers: Architecture for branch-office agility. http://www.cisco.com/c/dam/en/us/products/collateral/routers/4000-series-integrated-services-routers-isr/whitepaper_c11-732909.pdf.

[34] A. Ooka, S. Atat, K. Inoue, and M. Murata. Design of a high-speed content-centric-networking router using content addressable memory. In *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pp. 458–463, April 2014.

[35] 大岡睦, 吾多阿信, 井上一成, 村田正幸. 連想メモリを用いた高速なコンテンツセントリックネットワークルータのハードウェア設計と評価 (分散システム・nw). 電子情報通信学会技術研究報告. IN, 情報ネットワーク, Vol. 113, No. 473, pp. 105–110, feb 2014.

[36] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pp. 1–12. ACM, 2009.

[37] B. Bou-Diab, K. Masood, and A. Matrawy. Xml router and method of xml router network overlay topology creation, apr 2009. US Patent App. 11/905,246.

[38] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. Xquery 1.0 and xpath 2.0 data model. *W3C working draft*, Vol. 15, , 2002.

[39] Data sheet: Brocade 5600 vrouter. http://www.brocade.com/ja/backend-content/pdf-page.html?/content/dam/common/documents/content-types/datasheet/brocade-5600-vrouter-ds.pdf.

[40] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, p. 19. ACM, 2010.

[41] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, Vol. 38, No. 2, pp. 69–74, 2008.

[42] K. Inoue, D. Akashi, M. Koibuchi, H. Kawashima, and H. Nishi. Semantic router using data stream to enrich services. *Proc. CFI*, pp. 20–23, 2008.

[43] J. Sawada and H. Nishi. Hardware acceleration and data-utility improvement for low-latency privacy preserving mechanism. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 499–502. IEEE, 2012.

[44] F. Yamaguchi, K.e Matsui, and H. Nishi. Ram-based hardware accelerator for network data anonymization. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–4. IEEE, 2014.

[45] 増田和紀, 石田慎一, 西宏章. ネットワークルータにて取得したコンテンツに基づくサイト横断型リコメンデーションサービスの提案 (クラウド,2012 年並列/分散/協調処理に関する『鳥取』サマー・ワークショップ (swopp 鳥取 2012)). 電子情報通信学会技術研究報告. CPSY, コンピュータシステム, Vol. 112, No. 173, pp. 127–132, jul 2012.

[46] K. Masuda, S. Ishida, and H. Nishi. Cross-site recommendation application based on the viewing time and contents of webpages captured by a network router. In *Internet Computing and Big Data (ICOMP'2013), The 2013 International Conference on*, pp. 92–98, July 2013.

[47] E. Harahap, J. Wijekoon, R. Tennekoon, F. Yamaguchi, S. Ishida, and H. Nishi. Article: Modeling of router-based request redirection for content distribution network. *International Journal of Computer Applications*, Vol. 76, No. 13, pp. 37–46, August 2013.

[48] J. Wijekoon, E. Harahap, K. Takagiwa, R. Tennekoon, and H. Nishi. Effectiveness of a service-oriented router in future content delivery networks. In *Ubiquitous and Future Networks (ICUFN), 2015 Seventh International Conference on*, pp. 444–449, July 2015.

[49] N. Das and T. Sarkar. Survey on host and network based intrusion detection system. *International Journal of Advanced Networking and Applications*, Vol. 6, No. 2, Sep 2014.

[50] Jubatus: Distributed online machine learning framework. `http://jubat.us/en/`.

[51] Google caching overview. `http://peering.google.com/about/ggc.html`.

[52] R. Giladi. *Network processors: architecture, programming, and implementation*. Morgan Kaufmann Pub, 2008.

[53] M.T. Rose. *The Open Book: A Practical Perspective on OSI*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[54] Rfc 826 - ethernet address resolution protocol: or converting network protocol addresses to 48.bit ethernet address for transmission on ethernet hardware. `http://tools.ietf.org/html/rfc826`, November 1982.

[55] Rfc 791 - internet protocol. `http://tools.ietf.org/html/rfc791`, September 1981.

[56] Rfc 1071 - computing the internet checksum. `http://tools.ietf.org/html/rfc1071`, September 1988.

[57] Rfc 793 - transmission control protocol. `http://tools.ietf.org/html/rfc793`, September 1981.

[58] Douglas E. Comer . *Network Systems Design using Network Processors (IXP1200Version)*. Pearson Prentice Hall, 2004.

[59] Y. Rekhter and T. Li. An Architecture for IP Address Allocation with CIDR. RFC 1518, RFC Editor, September 1993.

[60] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, RFC Editor, January 2006.

[61] Edward Fredkin. Trie memory. *Commun. ACM*, Vol. 3, No. 9, pp. 490–499, sep 1960.

[62] D.R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, Vol. 15, No. 4, pp. 514–534, oct 1968.

[63] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, Vol. 41, No. 3, pp. 712–727, March 2006.

[64] X. Chen. Effect of caching on routing-table lookup in multimedia environment. In *INFOCOM '91. Proceedings. Tenth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking in the 90s., IEEE*, pp. 1228–1236 vol.3, Apr 1991.

[65] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *(INFOCOM)'98*, pp. 1240–1247, 1998.

[66] T.-C. Chiueh and P. Pradhan. High-performance ip routing tablelookup using cpu caching. In *INFOCOM'99*, pp. 1421–1428, 1999.

[67] T.-C. Chiueh and P. Pradhan. Cache memory design for network processors. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pp. 409–418, Jan 2000.

[68] J.J. Rooney, J.G. Delgado-Frias, and D.H. Summerville. Associative ternary cache for ip routing. In *IEE Proceedings of Computers and Digital Techniques*, pp. 409–416, Nov. 2004.

[69] R. Jain. Characteristics of destination address locality in computer networks: A comparison of caching schemes. *Computer Networks and ISDN Systems*, Vol. 18, pp. 243–254, 1989.

[70] E.G. Coffman, Jr. and P.J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[71] W. Peterson and D. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, Vol. 49, No. 1, pp. 228 – 235, 1961.

[72] K. Li, F. Chang, D. Berger, and W.-C. Feng. Architectures for packet classification caching. Networks（ICON）, 2003 11th IEEE International Conference on, pp. 111–117, Sep. 2003.

[73] K.-Y. Ho and Y.-C. Chen. Performance evaluation of ipv6 packet classification with caching. In *Proceedings of ChinaCom*, pp. 669–673, 2008.

[74] S. Ata, M. Murata, and H. Miyahara. Efficient Cache Structures of IP Routers to Provide Policy-Based Services . In *The 2001 IEEE International Conference on Communications (ICC'2001)*, pp. 1561–1565, 2001.

[75] 西宏章, 奥野通貴. 次世代インターネットを支えるネットワークプロセッサアーキテクチャ P-Gear の提案. 電子情報通信学会 CS 研究会 CS2003-96, pp. 81–84, Sep. 2003.

[76] Catalyst 3750 シリーズスイッチの switching database manager の説明と設定. http://www.cisco.com/cisco/web/support/JP/100/1007/1007880_swdatabase%_3750ss_44921-j.pdf.

[77] S. Gamage and A. Pasqual. High performance parallel packet classification architecture with popular rule caching. In *Networks (ICON), 2012 18th IEEE International Conference on*, pp. 52–57, Dec 2012.

[78] 石田慎一, 原島真悟, 川島英之, 鯉渕道紘, 西宏章. パケットデータ管理基盤における情報抽出処理の効率化技法 (コンピュータシステム技術 2, 組込み技術とネットワークに関するワークショップ etnet2010). 電子情報通信学会技術研究報告. CPSY, コンピュータシステム, Vol. 109, No. 474, pp. 309–314, mar 2010.

[79] 牧野友昭, 辻良繁, 川島英之, 鯉渕道紘, 西宏章. サービス指向ルータにおける高速なデータ書き込み機構の提案 (ネットワーク技術,2009 年並列/分散/協調処理に関する『仙台』サマー・ワークショップ (swopp 仙台 2009)). 電子情報通信学会技術研究報告. CPSY, コンピュータシステム, Vol. 109, No. 168, pp. 79–84, jul 2009.

[80] Snort official web site. http://www.snort.org.

[81] Y. Sugawara, M. Inaba, and K. Hiraki. High-speed and memory efficient tcp stream scanning using fpga. In *Field Programmable Logic and Applications, 2005. International Conference on*, pp. 45–50, Aug 2005.

[82] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, RFC Editor, May 1996.

[83] 八巻隼人, 中村優一, 高際兼一, 松井加奈絵, 西宏章. インターネットルータにおける http 圧縮ストリームの高速展開処理機構の提案. 電子情報通信学会論文誌. B, 通信, Vol. J98-B, No. 10, pp. 1104–1114, Oct. 2015.

[84] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berbers-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, June 1999.

[85] zlib home site. http://zlib.net/, April 2010.

[86] P. Deutsch. Gzip file format specification version 4.3, 1996.

[87] P. Deutsch. Deflate compressed data format specification version 1.3, 1996.

[88] P. Erragina and G. Manzini. An experimental study of an opportunistic index. In *In SODA*, pp. 269–278, 2001.

[89] J. Storer and T. Szymanski. Data compression via textual substitution. *J. ACM*, Vol. 29, pp. 928–958, 1982.

[90] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, Vol. 40, No. 9, pp. 1098–1101, Sept 1952.

[91] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, Vol. 23, No. 3, pp. 337–343, May 1977.

[92] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, Vol. 24, No. 5, pp. 530–536, September 1978.

[93] 福島荘之介. ファイル圧縮ツール gzip のアルゴリズム, 28 巻, pp. 30–37. bit, 3, March 1996.

[94] Rfc 1950 - zlib compressed data format specification version 3.3. `http://www.ietf.org/rfc/rfc1950.txt`, May 1996.

[95] B.W.Y. Wei and T.H. Meng. A parallel decoder of programmable huffman codes. *Circuits and Systems for Video Technology, IEEE Transactions on*, Vol. 5, No. 2, pp. 175 –178, apr 1995.

[96] Bro. `http://www.bro-ids.org`.

[97] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Commun. ACM*, Vol. 20, No. 10, pp. 762–772, October 1977.

[98] D.M. Sunday. A very fast substring search algorithm. *Commun. ACM*, Vol. 33, No. 8, pp. 132–142, August 1990.

[99] A.V. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, Vol. 18, No. 6, pp. 333–340, June 1975.

[100] A. Kennedy, X. Wang, Z. Liu, and B. Liu. Ultra-high throughput string matching for deep packet inspection. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pp. 399–404, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[101] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, Vol. 4, pp. 2628–2639 vol.4, March 2004.

[102] D. Pao, W. Lin, and B. Liu. A memory-efficient pipelined implementation of the aho-corasick string-matching algorithm. *ACM Trans. Archit. Code Optim.*, Vol. 7, No. 2, pp. 10:1–10:27, October 2010.

[103] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, Vol. 31, No. 2, pp. 249–260, March 1987.

[104] M. Attig S. Dharmapurikar and J. Lockwood. Implementation results of bloom filters for string matching. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[105] M. Attig, Sarang Dharmapurikar, and J. Lockwood. Implementation results of bloom filters for string matching. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pp. 322–323, April 2004.

[106] G. Papadopoulos and D. Pnevmatikatos. Hashing + memory = low cost exact pattern matching. In *Proc. Int. Conf. Field Program. Logic Appl*, pp. 39–44, 2005.

[107] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis. A reconfigurable perfect-hashing scheme for packet inspection. In *15th Int. Conference on Field Programmable Logic and Applications*, 2005.

[108] K. Huang, L. Ding, G. Xie, D. Zhang, A.X. Liu, and K. Salamatian. Scalable tcam-based regular expression matching with compressed finite automata. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, pp. 83–93, Oct 2013.

[109] M.H. Hajiabadi, H. Saidi, and M. Behdadfar. Scalable, high-throughput and modular hardware-based string matching algorithm. In *Information Security and Cryptology (ISCISC), 2014 11th International ISC Conference on*, pp. 192–198, Sept 2014.

[110] ORACLE. Oracle TimesTen In-Memory Database on Oracle Exalogic Elastic Cloud. `http://www.oracle.com/technetwork/database/database-technologies/timesten/overview/timesten-exalogic-wp-july2011-487843.pdf`.

[111] Y. Nishida and H. Nishi. Implementation of a hardware architecture to support high-speed database insertion on the internet. In *Engineering of Reconfigurable Systems and Algorithms (ERSA), International Conference on*, pp. 1–6, July 2012.

[112] Empress Software Japan Inc. 株式会社 Empress Software Japan コンポーネントフレームワーク. `http://www.empressjapan.co.jp/produced/component_framework.php`.

[113] ALTIBASE Corporation. ハイブリッドメモリデータベース ALTIBASE Altibase Technical White Paper 2007/10/01. `http://www.science-arts.com/altibase/area/downloads/wp_002.pdf`.

[114] M. Iwazume, T. Iwase, K. Tanaka, and H. Fujii. Big data in memory: Benchmarking in memory database using the distributed key-value store for constructing a large scale information infrastructure. In *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, pp. 199–204, July 2014.

[115] L. Ceuppens, A. Sardella, and D. Kharitonov. Power saving strategies and technologies in network equipment opportunities and challenges, risk and rewards. In *Applications and the Internet, 2008. SAINT 2008. International Symposium on*, pp. 381–384, July 2008.

[116] 阿多信吾, 黄恵聖, 山本耕次, 井上一成, 村田正幸. 低コスト・低消費電力 tcam における効率的なルーティングテーブル管理法 ((フォトニック)ip ネットワーク技術,(光) ノード技術,wdm 技術, 信号処理技術, 一般). 電子情報通信学会技術研究報告. NS, ネットワークシステム, Vol. 107, No. 443, pp. 7–12, jan 2008.

[117] N.B. Guinde, R. Rojas-Cessa, and S.G. Ziavras. Packet classification using rule caching. In *Information, Intelligence, Systems and Applications (IISA), 2013 Fourth International Conference on*, pp. 1–6, July 2013.

[118] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting route caching: The world should be flat. In *Passive and Active Network Measurement (PAM) 2009, 10th International Conference on*, pp. 3–12, April 2009.

[119] W. Jiang, Q. Wang, and V.K. Prasanna. Beyond tcams: An sram-based parallel multi-pipeline architecture for terabit ip lookup. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, April 2008.

[120] 環境負荷低減に資する ict システム及びネットワークの調査研究会報告書. http://warp.da.ndl.go.jp/info:ndljp/pid/283520/www.johotsusintokei.soumu.go.jp/field/data/gt070406.pdf.

[121] グリーン it 推進協議会 調査分析委員会 総合報告書（2008 年度〜2012 年度）. http://home.jeita.or.jp/greenit-pc/activity/reporting/110628/pdf/survey01.pdf.

[122] 低消費電力型通信技術等の研究開発（エコインターネットの実現）基本計画書 - 総務省. http://www.soumu.go.jp/main_content/000020284.pdf.

[123] ミック経済研究所. データセンター市場と消費電力・省エネ対策の実態調査 2015 年度版, 2015.

[124] A. Appleby. Murmurhash first announcement. http://tanjent.livejournal.com/756623.html, March 2008.

[125] 山口史人, 石田慎一, 西宏章. ネットワークアプリケーションにおけるハードウェアハッシュの有用性検証. Technical Report 17, 研究報告計算機アーキテクチャ(ARC), Jul 2013.

[126] F. Yamaguchi and H. Nishi. Hardware-based hash functions for network applications. In *Networks (ICON), 2013 19th IEEE International Conference on*, pp. 1–6, Dec 2013.

[127] B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, Vol. 22, No. 9, 1997.

[128] B. Shirazi and Y.-D. Song. A parallel exchange sort algorithm. In *Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on*, pp. 366 –370, mar 1989.

[129] J. Moy. Ospf version 2, 1998.

[130] North Carolina State University. FreePDK45. http://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[131] T. Heil, A. Krishna, N. Lindberg, F. Toussi, and S. Vanderwiel. Architecture and performance of the hardware accelerators in ibm's poweren processor. *ACM Trans. Parallel Comput.*, Vol. 1, No. 1, pp. 5:1–5:26, May 2014.

[132] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual September 2015. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[133] K. Papagiannaki, N. Taft, S. Bhattacharyya, P. Thiran, K. Salamatian, and C. Diot. A pragmatic definition of elephants in internet backbone traffic. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, pp. 175–176, New York, NY, USA, 2002. ACM.

[134] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. *SIGCOMM Comput. Commun. Rev.*, Vol. 32, No. 4, pp. 309–322, 2002.

[135] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto. Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pp. 115–120, New York, NY, USA, 2004. ACM.

[136] S. Urushidani, S. Abe, K. Fukuda, J. Matsukata, Y. Ji, M. Koibuchi, and S. Yamada. Architectural Design of Next-generation Science Information Network. *IEICE Transaction*, Vol. E90-B, No. 5, pp. 1061–1070, May 2007.

[137] Sinet4 学術情報ネットワーク［サイネット・フォー］ sience information network 4. http://www.sinet.ad.jp/.

[138] Ipa 独立行政法人 情報処理推進機構：tcp/ip に係る既知の脆弱性に関する調査報告書. http://www.ipa.go.jp/security/vuln/vuln_TCPIP.html.

[139] コンピュータ緊急対応センター（JPCERT/CC）. http://www.jpcert.or.jp/at/2003/at030001.txt.

[140] Internet Storm Center. http://isc.sans.edu/diary.html?storyid=7924.

[141] WIDE プロジェクト. http://www.wide.ad.jp/.

[142] B. Agrawal and T. Sherwood. Modeling tcam power for next generation network devices. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, pp. 120–129, March 2006.

[143] S.J.E. Wilton and N.P. Jouppi. Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, Vol. 31, No. 5, pp. 677–688, May 1996.

[144] CACTI An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. http://www.hpl.hp.com/research/cacti/.

[145] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann (ISBN 978-0-12-383872-8), 2014.

[146] 上田悠太. 次世代光ネットワーク用半導体高速光スイッチに関する研究. PhD thesis, 早稲田大学, 2011.

論文目録

定期刊行誌掲載論文（主論文に関連する原著論文）

[1] 八巻隼人，中村優一，高際健一，松井加奈絵，西宏章：インターネットルータにおける HTTP 圧縮ストリームの高速展開処理機構の提案，電子情報通信学会論文誌 B，Vol.J98-B，No.10，pp.1104-1114，Oct. 2015.

[2] H. Yamaki, H. Nishi: An Improved Cache Mechanism for a Cache-based Network Processor, Journal of Communication and Computer, Vol.10, No.3, pp.277-286, Mar. 2013.

[3] H. Yamaki, Y. Nagatomi, H. Nishi: Effective Hash-Based Filtering Architecture for High-throughput Regular-Expression Matching, International Journal of Information and Electronics Engineering, Vol.2, No.5, pp.672-677, Sep. 2012.

国際会議論文（査読付きの full-length papers）

[1] H. Yamaki, H. Nishi: An Improved Cache Mechanism for a Cache-based Network Processor, The 2012 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2012 in WORLDCOMP2012), pp.428-434, Jul. 2012.

国内学会発表

[1] 八巻隼人，西宏章：HTTP 圧縮ストリームの GZIP 展開逐次処理におけるメモリ容量の削減，第 8 回インターネットと運用技術シンポジウム, Nov. 2015.

[2] 八巻隼人，西宏章：パケット処理キャッシュに特化した低回路コストなキャッシュエントリ制御手法の実現，計算機アーキテクチャ研究会, Jul. 2014.

[3] 八巻隼人，西宏章：ネットワークプロセッサにおけるトラフィックに基づいたフローキャッシュポート分割手法の提案，コミュニケーションクオリティ研究会，Sep. 2013.

[4] 八巻隼人，西宏章，新世代キャッシュ搭載ネットワークプロセッサにおけるキャッシュアルゴリズムの提案，先進的計算基盤システムシンポジウム，May 2012.

[5] 八巻隼人，原島真悟，鯉渕道紘，西宏章，アプリケーションゲートウェイにおける処理オフロードの実現，計算機アーキテクチャ研究会（ARC2011），高知，Mar. 2011.