情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 1　1–11 (June 2008)

高速パケットフィルタの実装と評価

山　下　義　行[†1]　　鶴　　正　人[†2]

パケットフィルタリング処理はあらゆる種類のネットワーク機器に必要な機能になってきている．ハイエンドのルータやファイアウォールであればハードウェアベースの実装も可能である．さもなくば柔軟かつ安価な実現のために汎用 CPU を使ってソフトウェア的に実装されるが，その場合には処理の高速性に欠点がある．そこで本研究ではパケットフィルタプログラムにコード最適化手法，特に条件分岐を含むループのためのソフトウェア・パイプライン化手法を適用し，インテル IA-64 Itanium 2 プロセッサ上での高速化を試みる．著者らはすでにパケットモニタ・ツール `tcpdump` について高速化の効果を確認している．本研究ではその手法を一部変更して適用し，商用 C コンパイラによって最適化した場合の 4 倍の高速化，ソフトウェア・パイプライン化を用いない最適化の 2 倍の高速化を達成した．今回開発した最も高速なフィルタプログラムは Itanium 2 プロセッサの上限性能で動作する．

Implementation and Evaluation of Fast Packet Filters

Yoshiyuki Yamashita[†1] and Masato Tsuru[†2]

Packet filters are essential for most areas of recent network technologies. While high-end expensive routers and firewalls are implemented in hardware-based, flexible and cost-effective ones are usually in software-based solutions using general-purpose CPUs but have less performance. In order to solve this performace problem, we apply code optimization techniques to packet filter implementations, in particular the software pipelining techniques for a loop with conditional branches, on Intel IA-64 Itanium 2 processor. The authors have studied the method of applying the techniques to the packet monitoring tool `tcpdump` and reported their high effects. Using the revised method, we can obtain a software-pipelined packet filter implemetation which is four times faster than a C compiler based one and two times faster than an optimized code without software pipelining. The fastest filter program developed in this research can execute at the maximum speed of Itanium 2 processor.

1. はじめに

パケットフィルタは，与えられた一連の規則，条件（フィルタルール）に従って個々の入力パケットのヘッダまたはペイロードを検査する．そしてパケットが条件を満たせば，それに対応する処理（通過，廃棄，印付け，記録など）を行う．このような処理はネットワークにおいて重要性を増しているトラヒック管理やセキュリティ管理に不可欠な基本機能であり，それゆえ，IP ルータやサーバ，ファイアウォールだけでなく，最近ではあらゆる規模・種類のネットワーク機器に実装されてきている．ハイエンドのルータやファイアウォールであればハードウェアベースの実装（ASIC や FPGA を用いる実装[9),13)]）も可能であるが，一般に高コストである．一方，柔軟かつ安価な実現のためには汎用 CPU を使ってソフトウェア的に実装されるが，その場合には処理の高速性に欠点がある．

近年のブロードバンド化による通信帯域幅増大に対応するためにはパケットフィルタの高速化が急務である．また，パケットフィルタの利用形態の多様化・適用範囲の拡大にともない，状況に応じ実時間で動的に変更できるなどの柔軟性・拡張性が要請されている．このような機能・性能をコスト効率良く実現するためには，ソフトウェアベースのパケットフィルタの高速化手法の確立が不可欠と考えられる．さらには高位レベルの高速化（フィルタ・ルール構造の最適化）と下位レベルの高速化（機械語レベルの実行コードの最適化）の両方を実現し，それを効果的に組み合わせる必要がある．前者の例として入力パケット列の特性に応じてパターンマッチの適用順序などの構造を最適化する研究がなされており，後者の例として最適化コンパイラのための様々な技法を応用することができる．

本論文では，パケットフィルタにおける下位レベルの高速化に焦点を絞り，コード最適化，特に条件分岐を含むループのためのソフトウェア・パイプライン化手法を適用する．これまでにもパケットフィルタの実行処理系の高速化の研究[2),4),8)] は存在するが，著者らが知る限り，ソフトウェア・パイプライン化に基づいてコード最適化技法を追求した研究はない．ここでの課題は，処理が多数の条件分岐を含むために通常のソフトウェア・パイプライン化技法がそのままでは適用困難な点にある．著者らは，今回のパケットフィルタ高速化

†1 佐賀大学理工学部知能情報システム学科
Department of Information Science, Saga University
†2 九州工業大学大学院情報工学研究院電子情報工学研究系
Department of Computer Science and Electronics, Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology

の準備研究として tcpdump [6] のパケットフィルタ処理を高速化することを試み，predicate execution [5] や enhanced modulo scheduling [14] を多数の条件分岐にも適用できるように拡張し，成果を得ている[11),12)]．本論文ではそれらを一部変更して適用し，一般的なパケットフィルタにおいても十分な高速化が実現可能であることを示す．

本論文の構成は以下のとおりである．2 章では，以後の説明の準備と本論文の位置づけの明確化のために，著者らの先行研究を紹介する．3 章ではフィルタルールをモデル化し，4 章ではルール・マッチングを実行するフィルタプログラムの複数の実装方法を提案・説明する．5 章では複数の実装の性能比較のベースとなるルール・マッチングの実行時間を数理モデル化し，6 章では性能比較の実験とその結果を説明する．

2. 先行研究（tcpdump の高速化）

この章では著者らが先行研究[11),12)] として手がけた tcpdump の高速化についてその手法と効果を概説する．この章の手法を応用し，**図 1** のような実際のルータやファイアウォールで採用している実用レベルのフィルタルールを処理する高速パケットフィルタを構築することが本研究の目的である．

2.1 フレームワーク

パケットフィルタ処理は次の 2 ステップからなる．

ステップ 1　入力パケットがフィルタルールにマッチするか否かを判定する[⋆1]．

ステップ 2　その判定結果に従い，入力パケットを処理する．

この 2 ステップを C プログラムで記述すれば以下のとおりである．

```
result = filter();
action(result);
```

ここに filter() がステップ 1 を，action() がステップ 2 を実行する関数である．

さて，たとえば DOS 攻撃を受けているときには短時間に大量のパケットが押し寄せるから，それらパケットは以下のようなループで処理することになる．

```
for(i = 0; i < n_packets; i++){
    result[i] = filter(i);
    action(result[i]);
}
```

⋆1 正確にいえば，フィルタルールが表現するパケット集合に入力パケットが含まれるか否かを判定する．

```
ip filter 1 reject X.X.X.0/24 * * * *
ip filter 2 pass * X.X.X.0/24 established * *
ip filter 3 pass X.X.X.X/29 X.X.X.X tcp * smtp
ip filter 4 pass X.X.X.0/24 X.X.X.X tcp * 5000-6000
ip filter 5 pass * * udp * domain
ip filter 6 pass X.X.X.X/29 X.X.X.X tcp * pop3
...
```

図 1　フィルタルールの例（一部抜粋，上記 X.X.X.X には実際には特定のアドレス値が入る）
Fig. 1　An Example of a set of filter rules (an extract, replacing each of X.X.X.X with a concrete address value in real rules).

ここに n_packets は入力パケット数である．しかし，上のループは以下のように 2 つのループに分割する方が高速処理に向いている．

```
for(i = 0; i < n_packets; i++){ //ループ 1
    result[i] = filter(i);
}
for(i = 0; i < n_packets; i++){ //ループ 2
    action(result[i]);
}
```

前半の for ループ（ループ 1）はパケットがルールにマッチするか否かを判定する処理であるから，論理/算術演算のみで構成されており，これには様々なコード最適化技法が適用可能である．これに対して，後半の for ループ（ループ 2）はシステムコールなどを含むため，このループにコード最適化を適用することは困難と考えられる．しかし大量に届く DOS パケットはすべて廃棄されるだけであるから，ループ 2 は処理時間のネックにはならない．むしろ大量の DOS パケットがルールにマッチしないことを高速に判定するために，ループ 1 の高速化が重要である．そこで先行研究[11),12)] ではループ 1 にコード最適化技法，特に高度に洗練されたループ最適化手法の 1 つであるソフトウェアパイプライン化技法[1)] を適用することを試みた．

なお，この先行研究のフレームワークは本研究のフレームワークとは以下の点で異なることを注意する．

(1)　先行研究のフィルタルールは tcpdump の比較的単純なルール（たとえば終点ポート番号が 3000 であることを表すルール"dst port 3000"など）を対象としているが，

本研究のルールは図 1 のような，定型的フィルタパターン（3.2 節参照）を多数（しばしば数百行以上も）有するルールを対象とする．

(2)　先行研究では短時間に大量に届く入力パケットを想定しているが，本研究では入力パケットが 1 個であってもコード最適化の効果が出る状況を想定している．

つまり，先行研究では大量のパケットに関して繰り返すループをコード最適化の対象とするが，本研究では大量のフィルタパターンに関して繰り返すループを対象とする．そのような違いはあるものの，適用できる最適化技法は両者で共通しており，後に示すように最適化の効果も類似している．

2.2　コード最適化技法

先行研究[11),12)] の最適化技法を概説する．

2.2.1　コンパイリング

`tcpdump`[6)] では，フィルタルールは `bpf` コンパイラによって中間コードに変換され，その中間コードは仮想計算機によって実行される．これはいわゆるインタプリタ方式であり，処理速度は非常に遅い．そこでフィルタルールを C プログラムへ変換し，その C プログラムをコンパイラで実行可能コードへ翻訳した．それだけでも高速化できる．

2.2.2　単純なコード最適化

さらなる高速化を目指し，ループ 1 に特化されたコード最適化器を開発した．この最適化器はフィルタルールから実行可能コードを直接生成する．ただし適用する最適化技法は不要命令削除，基本ブロック内リスト・スケジューリングなどの初歩的な技法のみに限定した．このようにして生成されたコードを，次に述べるソフトウェアパイプライン化技法が適用されていないという意味で，以下「非ソフトウェアパイプライン化コード」と呼ぶ．

2.2.3　ソフトウェアパイプライン化

ソフトウェアパイプライン化[1)] は最も洗練されたループ最適化技法の 1 つである．しかしループ 1 の本体（=関数 `filter()` の処理内容）は多くの条件分岐を含むため，この技法をループ 1 へ適用することは容易ではない．

条件分岐を含むループのためのソフトウェアパイプライン化技法として述語付き実行[5)]（Predicated Execution，以下 PE）と Enhanced Modulo Scheduling[14)]（以下 EMS）が知られている．先行研究ではこの 2 つの技法をループ 1 向けに改良し，専用のコード最適化器を開発した．以下ではそれぞれの技法で生成されたコードを「PE ソフトウェアパイプライン化コード」，「EMS ソフトウェアパイプライン化コード」と呼ぶ．

表 1　先行研究[12)] の実験結果
Table 1　The experimental results of the previous work [12)].

	実装方式	加速率	18.6/加速率
1	`tcpdump` インタプリタ	1.0	18.6
2	非構造化 C プログラム（`gcc` 適用）	5.7	3.3
3	構造化 C プログラム（`gcc` 適用）	5.6	3.3
4	非構造化 C プログラム（`icc` 適用）	8.3	2.2
5	構造化 C プログラム（`icc` 適用）	11.3	1.6
6	非ソフトウェアパイプライン化コード	8.3	2.2
7	PE ソフトウェアパイプライン化コード	18.6	1.0
8	EMS ソフトウェアパイプライン化コード	16.3	1.1

2.3　実験結果と評価

表 1 は文献 12) から転載した実験結果である．ここに 2 行目から 5 行目の「非構造化/構造化 C プログラム」は 2.2.1 項の C プログラムであり，前者はラベルと `goto` 文からなるプログラム，後者は if 文のネストからなるプログラムである．GNU C コンパイラ `gcc` ではこの 2 種類のプログラムの実行速度に差がないが，インテル C コンパイラ `icc`[⋆1] では構造化プログラムにソフトウェアパイプライン化を適用する場合があるので，両者の実行速度に差が出る．

「加速率」の列は `tcpdump` インタプリタの実行速度を基準とした高速化率を示している．「18.6/加速率」の列は，PE ソフトウェアパイプライン化コードの実行速度を基準とした減速率を示している．この表から以下の事実を確認しておく．

(1)　PE ソフトウェアパイプライン化コード（表の 7 行目）は，`gcc` でコンパイルした C プログラム（2，3 行目）よりも 3 倍以上高速である．

(2)　PE ソフトウェアパイプライン化コードは，`icc` でコンパイルした C プログラム（4，5 行目），非ソフトウェアパイプライン化コード（6 行目）よりも約 2 倍高速である．

著者らはこの先行研究を本研究の予備的研究と位置づけ，本研究への適用可能性を詳細に調べてきた．本論文では先行研究の手法および実験結果が本研究においても有効であることを示す．

⋆1 Itanium 2 プロセッサ向けの様々な高度なコード最適化を実装している．

3. フィルタルール

本研究では一般的な静的 IP フィルタルール[3),10)] に準じた記述力を持つルールを想定している．この章ではその機能を概説する．

3.1 フィルタルールの全体構造

フィルタルールはフィルタパターンの 0 個以上の並びからなり，テキスト・ファイルに事前に編集されているものとする．たとえば図 1 はフィルタルールの記述例の一部である．ルールの各行が個々のフィルタパターンである．

フィルタプログラムは，プログラム起動後，フィルタルールが格納されたテキスト・ファイルを読み込み，それを内部表現へ変換し，メモリ上に配列する．そしてフィルタルールの先頭（一番上の行）のフィルタパターンから下に向かって順に，入力パケットがそのパターンにマッチするか否かをチェックし，あるパターンにマッチした時点で判定処理は終了し，対応するパケット処理を行う．もし入力パケットがすべてのパターンにマッチしなかったならば，そのパケットを廃棄する（これを「デフォルトルール」と呼ぶ）．

以下では単に「ルール」と呼ぶときにはルール全体を意味する．個々のフィルタパターンは「パターン」と呼ぶこともある．

3.2 フィルタパターン

1 つの（1 行の）フィルタパターンの記述は以下の構文に従うとする．

`ip filter` *n type sip dip proto spt dpt*

ここに各フィールド *n*，*type*，*sip*，... の意味は以下のとおりである．

n ...　パターン識別番号（16 bit 符号なし整数値）．ルール中の複数のフィルタパターンは識別番号の昇順に並んでいるものとする．

type ...　パケットの処理方法．`pass`（通過）または `reject`（廃棄）を指定する．

sip ...　始点 IP アドレス．以下の構文で指定する．

`*` ...　任意のアドレス．

$x_1.x_2.x_3.x_4$ ...　特定のアドレス．各 x_i は 8 bit 符号なし整数値．

$x_1.x_2.x_3.x_4/m$ ...　マスク付きアドレス．m は 32 以下の正整数値．

dip ...　終点 IP アドレス．*sip* と同じ構文で指定する．

proto ...　プロトコル識別子．以下の構文で指定する．

`*` ...　任意のプロトコル．

`tcp` ...　tcp プロトコル．

`udp` ...　udp プロトコル．

`established` ...　tcp コネクションが確立された tcp パケットを表す．具体的な手順では tcp パケットのフラグ・フィールドの ack ビットと rst ビットを調べる．

spt ...　始点ポート番号（tcp/udp パケットでのみ有効）．以下の構文で指定する．

`*` ...　任意のポート番号．tcp/udp プロトコル以外は必ずこの指定を行う．

p ...　特定のポート番号（16 bit 符号なし整数値）．

p_1-p_2 ...　ポート番号の範囲指定．$p_1 \leq p_2$．

文字列 ...　ニーモニック指定のポート番号．`smpt`，`www`，`domain` など．

dpt ...　終点ポート番号（tcp/udp パケットでのみ有効）．*spt* と同じ構文で指定する．

4. 実 装 方 法

フィルタルールが 3 章のように定義されるとき，パケットフィルタは以下の 3 つの処理を行う．

(1)　ルール中の各フィルタパターンを計算機の 2 進数内部表現に変換し，パターンの並びをその内部表現の配列としてメモリ上に配置する．

(2)　入力パケットが各内部表現とマッチするか否かの判定を配列要素順に繰り返す．

(3)　その判定結果に従って入力パケットを処理する．

上記 (1) はいわば前処理であり，この前処理の実行速度はフィルタの実行速度には関係しない．本研究ではこの前処理部を `lex`，`yacc` を用いて実装している．上記 (2) はフィルタ本体の処理であり，2.1 節の関数 `filter()` に対応する．上記 (3) はフィルタの出口でのシステム依存の処理であり，2.1 節の関数 `action()` に対応する．本研究は (2) の高速化に関するものであり，(3) には触れない．ルールが複雑化しパターン数が増大したときに，各入力パケットに対して発生する処理は (2) に関しては増大するが (3) に関しては変わらない．よって (2) に関する高速化はそれ単独でも重要である．本研究では (2) について以下に述べる数種類の実装を行い，実行時間を比較する．

4.1 C コンパイラによる実装

4.1.1 素朴な C プログラム

実行時間を強く勘案しなければ，上記 (2) の処理を行う C プログラムを開発することはきわめて容易であり，ほぼすべてのプログラマが同様のプログラムを作るであろう．それを「素朴なプログラム」と呼ぼう．そのようなプログラムの外形を**図 2** に示す．このプログラムには (1)〜(9) の 9 個の条件分岐が含まれる．なお簡単のため，現在は tcp プロトコルお

```
for(int i = 0; i < フィルタパターン数; i++){
  IP = [入力パケットの始点 IP アドレス] & [i 番目のパターンの sip の bit マスク];
  if(IP == [i 番目のパターンの sip]){                                    //(1)
    IP = [入力パケットの終点 IP アドレス] & [i 番目のパターンの dip の bit マスク];
    if(IP == [i 番目のパターンの dip]){                                  //(2)
      if([i 番目のパターンの proto] == tcp){                             //(3)
        if([入力パケットの始点ポート番号] >= [i 番目のパターンの spt の下限値]){   //(4)
          if([入力パケットの始点ポート番号] <= [i 番目のパターンの spt の上限値]){   //(5)
            if([入力パケットの終点ポート番号] >= [i 番目のパターンの dpt の下限値]){   //(6)
              if([入力パケットの終点ポート番号] <= [i 番目のパターンの dpt の上限値]){   //(7)
                FLAGS = [入力パケットの tcp フラグ・フィールド] &
                          [i 番目のパターンの tcp フラグ・フィールドの bit マスク];
                if(FLAGS == [i 番目のパターンの tcp フラグ・フィールド]){          //(8)
                  return [i 番目のパターンの type]; //全ての条件判定に成功した!
                }
              }
            }
          }
        }
      } else if([i 番目のパターンの proto] == *){                        //(9)
        return [i 番目のパターンの type]; //全ての条件判定に成功した!
      }
    }
  }
}
```

図 2　パケットフィルタ中核部を記述する素朴な C プログラム
Fig. 2　A naive C program that defines the core part of the packet filter.

よび任意プロトコル（*proto* = tcp or *proto* = ∗）のみを扱っている．その他のプロトコル（たとえば udp プトロコル）を扱うには図 2 と同等のループ・プログラムを別に用意し，入力パケットのプロトコルに応じて実行するループを 1 つ選ぶような前処理プログラムを追加すればよい．その前処理による性能低下は，プロトコルを 1 個追加するごとに 1 個の条件判定（比較命令 1 個，分岐命令 1 個）が追加されるにすぎない．

本研究では，作成された C プログラムを GNU コンパイラ gcc（ver. 2.96）とインテルコンパイラ icc（ver. 9.1）でコンパイルする．商用 C コンパイラのコード最適化機能のみを用いてフィルタプログラムの高速化を試みる方法を，以下では単に「C コンパイラによる実装」と呼ぶこととする．

4.1.2　アラインメント調整

パケット中の始点/終点 IP アドレスの格納開始位置は入力パケット先頭からそれぞれ 26 バイト目[⋆1]，30 バイト目にある．そのため，もしパケット先頭アドレスがメモリの 4 バイト境界にあるならば，IP アドレス格納の先頭アドレスは 4 バイト境界をまたぐ．このことがフィルタプログラムの実行性能を下げると予想されるが，先行研究では対策が不十分であった．

本研究では解決策としてパケットを格納する先頭アドレスをプログラム上で調整し，始点 IP アドレスの開始アドレスが 8 バイト境界[⋆2]になるように工夫する．

4.2　コード最適化による実装

4.2.1　マルチメディア命令と単純な最適化

複数の整数演算を同時に実行するマルチメディア命令はプログラム実行速度の向上に効果的である．本研究の場合，始点 IP アドレスと終点 IP アドレスの取扱い，および始点ポート番号と終点ポート番号の取扱いがまったく同じ演算内容である（図 2 参照）から，マルチメディア命令を利用することでその部分の演算数を半分に削減できる．ただし C コンパイラがプログラム中からマルチメディア命令利用可能部分を自動抽出することは一般には非常に困難であり，実際，本研究で用いた gcc，icc では不可能であった．そこでマルチメディア命令を含むコードを人手によるコンパイルを行って作成した．そしてそのコードに 2.2.2 項で述べたコード最適化器の技術を応用し，比較的単純なコード最適化を行った．マルチメディア命令は先行研究では使用しなかった．

リスト 1 は，Itanium 2 プロセッサ向けの実際のアセンブリ・コードである．Ltop がループの先頭，Laccept はパケットがあるファルタパターンにマッチした場合の処理先，Lexit はすべてのパターンにマッチしなかった場合のループ脱出先を表す．このコードでは，ロード命令のレーテンシ（latency）を考慮し，各パターンのすべての情報を初めにロードし，その後に比較演算を実行するように工夫している．各 cmp 命令の後ろに，図 2 の中の対応する if 文の番号をコメントとして付けた．その中で"//(1),(2)"は，2 つの if 文の判定を 1 つの cmp 命令で実現していることを表す．"//(4),(5)"，"//(6),(7)"ではマルチメディ

⋆1 始点 IP アドレスの格納開始位置は IP ヘッダの先頭からは 12 バイト目であるが，研究ではその前に Mac ヘッダ 14 バイトを含む実装を採用している．
⋆2 64 bit マシンの場合には 8 バイト境界にしておくと後に述べるマルチメディア命令の使用時にさらに効果的である．

リスト 1:非ソフトウェアパイプライン化コード

```
Ltop:
    ld8 r31 = [r105],8
    ld8 r30 = [r104],8
    ld1 r29 = [r103],1
    ld8 r28 = [r102],8
    ld8 r27 = [r99],8
    ld1 r24 = [r98],1
    ld2 r23 = [r97],2
;;
    and r31 = r31, r109
;;
    pcmp4.gt r28 = r28,r108
    cmp.eq p6, p7 = r30,r31        //(1),(2)
    (p6)br.cond.dptk L1
    br.ctop.dptk Ltop
    br Lexit
;;
L1:
    pcmp4.gt r27 = r108,r27
    cmp4.eq p6, p7 = 6,r29             //(3)
    (p6) br.cond.dptk L2
;;
    cmp4.eq p6, p7 = 0,r29             //(9)
    (p6)br.cond.dptk Laccept
    br.ctop.dptk Ltop
    br Lexit
;;
L2:
    cmp.eq p6,p7 = r28,r0          //(4),(5)
    (p6)br.cond.dptk L3
    br.ctop.dptk Ltop
    br Lexit
;;
L3:
    and r22 = r24, r106
    cmp.eq p6, p7 = r27,r0         //(6),(7)
    (p6)br.cond.dptk L4
    br.ctop.dptk Ltop
    br Lexit
;;
L4:
    cmp.eq p6, p7 = r24,r22           //(8)
    (p6)br.cond.dptk Laccept
    br.ctop.dptk Ltop
    br Lexit
;;
Laccept:
    st4 [r110] = r23
Lexit:
```

ア命令 pcmp4.gt（4 バイト整数の大小比較の 2 並列実行）の結果を利用している．ダブルセミコロン ";;" で囲まれた命令のグループは同時に発行可能である．Itanium 2 プロセッサは最大 6 命令同時発行可能であるが，実際に同時発行されるか否かは命令の種類，プロセッサ・リソースなどに強く依存する．

4.2.2　ソフトウェアパイプライン化

パケットフィルタにソフトウェアパイプライン化を適用する効果は大きいと予想される．そこで，2.2.3 項で述べた開発済みのコード最適化器を本研究向きに改造した．

PE ソフトウェアパイプライン化コードを得るには，まず，述語レジスタを用いてリスト 1 のコードから分岐命令を削除する．そのコードがリスト 2 である．リスト 2 は，ループ脱出（Laccept への分岐）以外の分岐命令を含まないから通常のソフトウェアパイプライン化手法が適用できる．実際に生成されるコードはコードに含まれる各ロード命令のレーテンシの見積りに応じて様々に変化するが，1 つの例がリスト 3 である．高速化を追求するため，このコードでは汎用レジスタ，述語レジスタにともに register rotation の機能を用いている．またリスト 2 に含まれた 2 つのループ脱出の分岐をリスト 3 では 1 つに合併するなどの小さな工夫も新規に行っている[⋆1]．このコードを人手で解読するには多大な労力を要するから，本論文ではこれ以上の解説は行わない．

EMS ソフトウェアパイプライン化コードでは，PE ソフトウェアパイプライン化コードとは異なり，述語レジスタを使用しない[⋆2]．そのため，実際の計算に必要な最少のプロセッ

⋆1 先行研究では入力パケットについて繰り返すループを扱うから，ソフトウェアパイプライン化コードからの途中脱出は検討する必要がなかった．

⋆2 いわゆる述語実行のためには述語レジスタを使用していないが，プロローグ/エピローグ・コードの省略や高速化のための仕掛けとしては register rotation 機能とあわせて使用している．

リスト 2:PE 非ソフトウェアパイプライン化コード

```
Ltop:
    ld8 r31 = [r105],8
    ld8 r30 = [r104],8
    ld1 r29 = [r103],1
    ld8 r28 = [r102],8
    ld8 r27 = [r99],8
    ld1 r24 = [r98],1
    ld2 r23 = [r97],2
;;
    and r31 = r31, r109
    pcmp4.gt r28 = r28,r108
;;
    pcmp4.gt r27 = r108,r27
    cmp.eq p6, p7 = r30,r31       //(1),(2)
;;
    (p6)cmp4.eq.unc p6,p7 = 6,r29     //(3)
;;
    (p7)cmp4.eq.unc p8,p9 = 0,r29     //(9)
    (p6)cmp.eq.unc p6,p7 = r28,r0 //(4),(5)
;;
    and r22 = r24, r106
    (p6)cmp.eq.unc p6,p7 = r27,r0 //(6),(7)
;;
    (p6)cmp4.eq.unc p6,p7 = r24,r22   //(8)
    (p6)br.cond.dptk Laccept
    (p8)br.cond.dptk Laccept
    br.ctop.dptk Ltop
    br Lexit
;;
Laccept:
    st4 [r110] = r23
Lexit:
```

サ・リソースだけを使用して実行可能であり，結果，実行時間が短縮されるという利点がある．しかし，述語レジスタを使用しない代わりに分岐命令が残る．分岐命令の実行では，分岐予測の失敗によって実行時間が増す可能性があり，これが EMS の欠点の 1 つである．また，分岐に依存して実行すべきコード・パターンは多種多様であって，そのパターン数は分岐の数とともに組合せ的に増える．その結果，すべてのコード・パターンを用意するにはコード・サイズが指数的に増大してしまう．これも EMS の欠点がある．EMS ソフトウェアパイプライン化によって生成されたコードの 1 つの例がリスト 4 である．このコードの全体サイズは 700 行を超えるため，リスト 4 では最初と最後の部分のみを示す．

PE/EMS ソフトウェアパイプライン化のより詳しい解説は先行研究[12] を参照してほしい．

リスト 3:PE ソフトウェアパイプライン化コード

```
Ltop:
    (p16)ld8 r32 = [r108],8
    (p21)pcmp4.gt r35 = r110,r35
    (p23)pcmp4.gt r37 = r37,r110
    (p18)ld2 r39 = [r102],2
    (p22)cmp.eq.unc p20,p0 = r36,r0
    (p24)cmp.eq p60,p0 = 0,r33
;;
    (p16)ld8 r33 = [r107],8
    (p16)ld8 r36 = [r105],8
    (p16)cmp.ne p59,p0 = r0,r0
    (p16)and r32 = r32,r111
    (p20)and r43 = r38,r109
;;
    (p16)cmp.eq.unc p22,p0 = r33,r32
    (p20)cmp.eq p62,p0 = r38,r43
    (p16)ld1 r32 = [r106],1
;;
    (p16)ld8 r33 = [r104],8
    (p23)cmp.eq.unc p20,p0 = r37,r0
    (p22)cmp.eq.unc p22,p23 = 6,r32
    (p18)ld1 r37 = [r103],1
    (p62)br.cond.dptk Laccept
    br.ctop.dptk Ltop
    br Lexit
    ;;
Laccept:
    st4 [r110] = r23
Lexit:
```

5. 実行時間モデル

実験の議論に進む前に，この章ではパケットの処理時間に関するモデルを構築し，パケッ

リスト 4:EMS ソフトウェアパイプライン化コード

```
Lxxxx_txxx_ttxx_tttx:
    ld8 r32 = [r108],8
    ld1 r38 = [r103],1
    cmp.eq.unc p6,p7 = 6,r35
    (p7)br.cond.dptk Lxxxx_tfxx_ttxx_tttx
;;
Lxxxx_ttxx_ttxx_tttx:
    cmp.eq.unc p6,p7 = r37,r0
    (p7)br.cond.dptk Lxxxx_ttxx_ttxx_tttf
        ;;
Lxxxx_ttxx_ttxx_tttt:
    ld8 r37 = [r107],8
    pcmp4.gt r33 = r33,r110
    cmp.ne p59,p0 = r0,r0
    and r32 = r32,r111
    cmp.eq.unc p6,p7 = r34,r0
    (p7)br.cond.dptk Lxxxx_ttxx_ttfx_tttt
;;
    .... 途中の 746 行を省略
;;
Lfxxx_xxxx_xxxx_xxxx:
    ld8 r32 = [r105],8
    ld8 r35 = [r104],8
    (p62)br.cond.dptk Laccept
    br.ctop.dptk Lxxxx_xxxx_xxxx_xxxx
;;
    br Lexit
;;
Laccept:
    st4 [r112] = r40
Lexit:
```

トフィルタの性能を評価する土台を作る．

5.1　パケット，ルール，実行時間の関係式

今，ルールが N 個のフィルタパターンを持つと仮定する．そして入力パケットがルールの先頭から k 番目（k 行目）のパターンにマッチするときのパケットフィルタの実行時間 t が以下の式で与えられると仮定する．

$$t = T_O + kT_P$$

ここに定数 T_O はパケットフィルタの動作の前処理，後処理に掛かる時間（いわゆるオーバヘッド）とする．定数 T_P はパケットがある 1 つのフィルタパターンにマッチするか否かを判定するために要する時間とする．なお，パケットがすべてのフィルタパターンにマッチしないときには以下の式とする．

$$t = T_O + NT_P$$

さて，入力パケットが k 番目のパターンにマッチする確率を p_k とする．またすべてのパターンにマッチしない確率を p_0 と表すならば，それは $1 - \sum_{k=1}^{N} p_k$ に等しい．このとき，1 つの入力パケットを処理するために必要な実行時間の期待値は以下のとおりである．

$$\begin{aligned} T &= \sum_{k=1}^{N} p_k(T_O + kT_P) + p_0(T_O + NT_P) \\ &= T_O + \alpha T_P \end{aligned} \tag{1}$$

ここに $\alpha = \sum_{k=1}^{N} kp_k + Np_0$ である．この式 (1) が入力パケットの統計的性質，フィルタルール，実行時間の期待値の 3 者の間の関係式である．

今，実験用の入力パケットの集合とフィルタルールの組が与えられたとしよう．その組について実際のネットワーク実験を行えば，パケット 1 個あたりの平均処理時間 T を実験的に求めることができる．またパケットの集合とルールを別途解析することで確率 p_k を求めることができ，その確率から式 (1) の第 2 項の係数 α（$= \sum_{k=1}^{N} kp_k + Np_0$）を計算できる．よって，1 組の入力パケットの集合とルールから T_O と T_P の一次関係式 1 つを得る．そのような組を複数与えて実験し，最小二乗法を用いて式 (1) を解けば，1 つのフィルタプログラムの実装について定数 T_O と T_P を知ることができる．

5.2　性 能 比 較

フィルタプログラムの 2 種類の実装 A と B について式 (1) の定数の組がそれぞれ (T_O^A, T_P^A)，(T_O^B, T_P^B) と求められたとする．そのとき両者の実行時間の期待値 T^A，T^B の比は以下のとおりである．

$$\frac{T^A}{T^B} = \frac{T_O^A + \alpha T_P^A}{T_O^B + \alpha T_P^B}$$

ここで α の値がフィルタプログラムの実装の種類によらないことを注意する．次に，α の値は確率分布 p_k によって決まるが，一般にはフィルタパターン数 N とともに増加すると考えて妥当であろう．特に多くの入力パケットがいずれのパターンにもマッチせず，デフォルトルールで廃棄されるような場合には $\alpha \approx Np_0$ となる．N が十分に大きな数ならば，結局，

表 2　様々なパケットフィルタの実装とその実行時間
Table 2　Implementations of the packet filter and their execution times.

	実装方式	実行時間（MC）				
		$\alpha = 2.00$	$\alpha = 8.99$	$\alpha = 17.68$	$\alpha = 34.36$	$\alpha = 69.73$
1	素朴な C プログラム（gcc 適用）	418.2	564.0	743.1	995.2	1508.0
2	素朴な C プログラム（icc 適用）	393.3	532.7	721.8	973.4	1540.6
3	アラインメント調整済み C プログラム（gcc 適用）	94.8	239.9	418.7	669.6	1179.4
4	アラインメント調整済み C プログラム（icc 適用）	60.0	199.3	389.2	639.9	1187.9
5	非ソフトウェアパイプライン化コード	52.8	130.0	208.8	350.4	597.2
6	PE ソフトウェアパイプライン化コード	57.8	89.1	123.9	191.0	325.8
7	EMS ソフトウェアパイプライン化コード	71.1	117.8	177.4	288.5	440.4

$T^A/T^B \approx T_P^A/T_P^B$
が成り立ち，定数 T_P^A と T_P^B が性能比較において重要な指標となる．

6. 実　　験

4 章の実装方式に従って開発したフィルタプログラムの実行速度を測定し，その評価を行った．

6.1 実験条件

本研究では，著者らの研究室のネットワークからキャプチャーした 10,000 個のパケットをいったんハードディスクに貯め，その 10,000 個のパケットをハードディスクから繰り返し読み出して実験を行った．実験に使用したマシンはインテル IA-64 Itanium 2 プロセッサ[5]（900 MHz，revision 7），Linux version 2.4.18-1. である．

フィルタルールとして，それぞれ 2 個，9 個，18 個，35 個，71 個のフィルタパターンを含む 5 種類のルールを作成した．よってパケットとルールの組は 5 種類である．式 (1) の係数 α は，10,000 個のパケットとこの 5 種類のルールについてそれぞれ 2.00，8.99，17.68，34.36，69.73 であった．

6.2 実験結果

表 2 は実験結果である．

表の 1，2 行目は 4.1.1 項の C プログラムを gcc と icc でコンパイルしたコードである．表の 3，4 行目は 4.1.2 項で述べたアラインメント調整を加えた C プログラムを gcc と icc でコンパイルしたコードである．5 行目は 4.2.1 項のコード，6，7 行目はそれぞれ 4.2.2 項で述べた PE コード，EMS コードである．実行時間は入力パケット 1 個あたりの平均処理時間であり，単位はマシンサイクル（MC）である．測定誤差はいずれも ±0.1 程度である．

表 3　線形近似係数（T_O，T_P）
Table 3　The coefficients (T_O, T_P) of linear approximation.

	T_0	T_P	$T_P/3.9$
1	425.7	15.8	4.1
2	388.8	16.7	4.3
3	102.5	15.7	4.0
4	59.3	16.4	4.2
5	57.5	7.9	2.0
6	53.0	3.9	1.0
7	74.4	5.4	1.4

表では 5 種類のルールそれぞれについて実行時間を示した．

実行時間から 5.1 節の議論に基づいて定数 T_O と T_P を計算した結果が**表 3** である．定数 T_P がフィルタの性能を表す主な指標である（5.2 節参照）．そこでフィルタ間の性能比を知るために，各 T_P を最も小さな T_P の値 3.9 で割った値 $T_P/3.9$ を表の最右列に示す．この列の値は，PE ソフトウェアパイプライン化コードの実行性能を基準とした各実装法の減速率を表す．

6.3 評　　価

6.3.1 T_P の比較による性能評価

表 3 の「$T_P/3.9$」の列から以下の事実が分かる．

(1)　PE ソフトウェアパイプライン化による実装は，C コンパイラによる実装（表 1〜4 行目）よりも約 4 倍高速である．先行研究の表 1 の場合，約 2〜3 倍であったから，本研究では PE ソフトウェアパイプライン化技法がより効果的である．

(2)　PE ソフトウェアパイプライン化による実装は，非ソフトウェアパイプライン化によ

る実装よりも 2 倍高速である．この結果は表 1 の結果と類似している．

(3)　PE ソフトウェアパイプライン化による実装は，EMS ソフトウェアパイプライン化よりも約 1.4 倍高速である．この結果も表 1 の結果と近い．

これらの結果から本研究の高速化の効果は先行研究のそれと同等のものであるといえよう．

6.3.2　演算性能の理論上限値

T_P の値は図 2 のループのイテレーション立ち上げ間隔[1)]（Iteration Initiation Interveal, 以下 I.I.I.）に対応する（5.1 節参照）．その T_P の値が，PE ソフトウェアパイプライン化コードでは 3.9 であることに注目したい（表 3 の 6 行目）．この数値は，このコードが Itanium 2 プロセッサの演算性能の理論上限値で動作していることを示している．

理由は，以下のとおりである．リスト 3 は実際に実験で用いた Itanium 2 プロセッサ用のコードである．ダブルセミコロン";;"で区切られた命令群が Itanium 2 プロセッサが同時に発行できる命令群であるが，リスト 3 にはそのような命令群は 4 つあることから，このループは理論上は 4 MC で動作するはずである．そして T_P の値が 3.9 であることは，プロセッサがほぼ期待された動作をしていることを示している．ところで Itanium 2 プロセッサは最大 6 命令を同時発行できる．リスト 3 に含まれる命令数は 20 であるから，$\lceil 20/6 \rceil = 4$ となり，どのような命令スケジューリング[⋆1]を行ったとしても，I.I.I（$= T_P$）はけっして 4 MC 以下にはならないのである．なお，実験から得られた T_P 値 3.9 は 4 以下であるが，それは実行時間の測定誤差（±0.1 程度）によるものではなく，5 章のモデルが単純すぎるせいと考えられる．

これに対して先行研究[11),12)]では演算性能の理論上限値は達成できていない．主な理由は，先行研究ではパケットに関して繰り返すループをパイプライン化するが，本研究ではフィルタパターンに関して繰り返すループをパイプライン化する点で異なるためと考えられる（2.1 節参照）．パケットをメモリからレジスタへ読み込むときにはパケットデータをすべて読み込むのではなく，IP アドレスやポート番号などの必要なデータのみを読み込む．そのため，連続アクセスにはならず，キャッシュが効きにくい．対して，フィルタパターンをメモリからレジスタへ読み込むときにはフィルタパターンを格納する配列に対して連続アクセスを行うため，安定したレーテンシでレジスタへ読み込むことができる．

⋆1 ループアンローリングを適用すればさらなる高速化が可能であるが，手動による最適化は複雑すぎて手に余り，断念した．アンローリングを行う最適化器の開発については検討中である．

6.3.3　T_O の比較によるオーバヘッドの評価

表 3 の 1 行目と 3 行目の T_O 値の差 $425.7 - 102.5 = 323.4$，および 2 行目と 4 行目の T_O 値の差 $388.8 - 59.3 = 329.5$ は，データが 4 バイト境界をまたぐことによる性能低下である（4.1.2 項参照）．このことは先行研究では推測はされていたが，本研究によって明らかになった．

1，3 行目の T_O 値よりも 2，4 行目の T_O 値がそれぞれ小さい理由は単に `gcc` と `icc` の能力差と考えられる．なお T_P 値についていえば，1，3 行目の T_P 値よりも 2，4 行目の T_P 値がわずかではあるが逆に大きい．この原因は解明できていない．

7.　まとめと今後の課題

パケットフィルタの高速化を，先行研究である `tcpdump` の高速化技法（多くの条件分岐を含むループに対するソフトウェア・パイプライン化技法）を拡張することで実現し，商用 C コンパイラによって最適化した場合の 4 倍，ソフトウェア・パイプライン化を用いない最適化の 2 倍の性能を達成した．この達成した性能は Itanium 2 プロセッサの性能上限であることも確認した．

今後の研究として 2 つの課題を検討している．

1 つは，本研究の技法（ソフトウェアパイプライン化）を Itanium 2 以外の他のプロセッサに適用することである．述語レジスタを持つ商用プロセッサは現在 Intel IA-64 アーキテクチャのプロセッサに限定されているから，PE ソフトウェアパイプライン化は他のプロセッサには適用できない．しかし，EMS ソフトウェアパイプライン化はそのような特殊なハードウェアを必要とせず，すべての RISC プロセッサに適用可能である．前章の実験結果では EMS の実行速度は PE よりも 1.4 倍遅い．しかしソフトウェアパイプライン化を適用しない場合よりも 1.5 倍，C コンパイラによる実装よりも 3 倍高速である．同等の高速化は Itanium 2 以外のプロセッサでも期待できる．

もう 1 つの課題として，様々なフィルタ・ルールや入力トラヒックに対して，より詳しい性能分析や改良を行い，最終的には実ネットワーク上での性能を評価し，実用性を検証する．また並行して，上位レベルの最適化およびそれらの効果的な組合せも検討する．一方，パターンマッチングに関して，今回は完全一致や範囲一致しか扱わなかったが，最近は正規表現一致などを効率的に行うパケットフィルタ[7)]も研究されている．このような場合に対するソフトウェア・パイプライン化技法の適用も検討していく．

なお，本研究の一部は，日立製作所および日本学術振興会（科研費基盤（C）19500057）

の支援を受けている．

参 考 文 献

1) Appel, A.W.: *Modern Compiler Implementation in C*, Cambridge University Press (1997).
2) Begel, A., McCanne, S. and Graham, S.L.: BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture, *ACM SIGCOMM'99*, pp.123–134 (1999).
3) Cisco Systems, Inc.: *Configuring IP Access Lists*, Document ID: 23602. http://www.cisco.com/warp/public/707/confaccesslists.html
4) Cristea, M.L. and Bos, H.: A Compiler for Packet Filters, *Proc. ASCI'04* (2004).
5) Intel: Intel Itanium Architecture Software Developer's Manual. http://www.intel.com/design/itanium/documentation.htm
6) Jacobson, V., et al.: tcpdump(1), bpf..., Unix Manual Page (1990).
7) Kumar, S., Dharmapurikar, S., Yu, F., Crowly, P. and Turner, J.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection, *ACM SIGCOMM'06*, pp.339–350 (2006).
8) Okumura, T., Mossé, D., Minami, M. and Nakamura, O.: Network QoS Management Framework for Server Clusters An End-Host Retrofitting Event-Handler Approach using Netnice, *3rd Int. Symp. Cluster Computing and the Grid* (*CCGRID*), pp.284–291 (2003).
9) Singh, S., Baboescu, F., Varghese, G. and Wang, J.: Packet Classification Using Multidimensional Cutting, *ACM SIGCOMM'03*, pp.213–224 (2003).
10) ヤマハ：YAMAHA RT シリーズの IP パケット・フィルタ. http://www.rtpro.yamaha.co.jp/RT/FAQ/IP-Filter/index.html
11) Yamashita, Y. and Tsuru, M.: Code Optimization for Packet Filters, *Workshop on Internet Measurement Technology and its Applications to Building Next Generation Internet*, *SAINT2007* (2007).
12) Yamashita, Y. and Tsuru, M.: Software Pipelining for Packet Filters, *3rd Int. Conf. High Performance Computing and Communication* (*HPCC07*), LNCS 4782, pp.446–459 (2007).
13) Yusuf, S. and Luk, W.: Bitwise Optimised CAM for Network Intrusion Detection Systems, *Int. Conf. Field Programmable Logic and Applications*, pp.444–449 (2005).
14) Warter, N.J., Haab, G.E. and Bockhaus, J.W.: Enhanced Modulo Scheduling for Loops with Conditional Branches, *IEEE MICRO-25* (1992).

(平成 19 年 10 月 9 日受付)
(平成 20 年 2 月 25 日採録)

山下　義行（正会員）

昭和 57 年大阪大学理学部物理学科卒業．日立マイクロコンピュータエンジニアリング（株），筑波大学大学院博士課程，東京大学助手，筑波大学助教授等を経て，平成 13 年より佐賀大学理工学部知能情報システム学科教授．言語処理系に関する研究に従事．工学博士．

鶴　　正人（正会員）

昭和 60 年京都大学大学院数理工学専攻修了．沖電気工業，長崎大学，通信・放送機構，九州工業大学情報工学部助教授等を経て，平成 18 年より同学部教授．博士（情報工学）．インターネットの計測，制御，管理に関する研究に従事．