社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2008－HPC－118（20）
2008／12／17

パイプライン化高速パケットフィルタのx86への実装

山下義行† 鶴 正人††

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。しかし汎用CPUを用いるソフトウェアベースの実装を選ぶ場合、柔軟性の反面、高速性に難点がある。著者らはこの問題を多数の条件分岐を含むループのソフトウェアパイプライン化技法の適用として定式化し、Itanium 2プロセッサを対象としてパケットフィルタの高速化を行ってきた。本研究では、この手法がx86プロセッサにおいても同等に有効であることを示す。商用Cコンパイラによって最適化した場合の2倍、ソフトウェアパイプライン化を適用しない場合の1.7倍の高速化を達成した。

Implementations of Pipelined Fast Packet Filters on x86 Processors

Yoshiyuki Yamashita † and Masato Tsuru††

Packet filters are essential for most areas of recent network equipment. The filters implemented by software on general-purpose CPUs are flexible but suffer from poor performance. To solve this problem, the authors have studied the software pipelining techniques for a loop with many conditional branches as the key techniques of fast packet filters and reported their high effects on Intel Itanium 2 processor. In this paper we show that the techniques are also effective on Intel x86 processors; software pipelined programs are two times faster than commercial C compiler based optimal programs and 1.7 times faster than non-software pipelined optimal ones.

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し、パケットが条件を満たせば、対応する処理（通過、廃棄、印付け、記録等）を行う。その重要性ゆえに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いる。ハードウェアベースの実装では高速な処理が可能であるが、高コストであり、柔軟性・拡張性に問題がある。一方、安価かつ柔軟な実装は汎用CPUを用いてソフトウェア的になされるが、高速性に難点がある。本研究では、後者にコード最適化手法、特にフィルタ・プログラム中のループ部へ命令スケジューリング技法であるソフトウェア・パイプライン化[1]を適用し、高速化を目指す。

コード最適化から見た本研究の技術的な課題は、フィルタ・プログラムが多数の条件分岐を含むことである。ソフトウェアパイプライン化を条件分岐を含まないループへ適用する技法は既に確立しているが、条件分岐を含む場合については未だ確立した手法はない。

この課題に取り組むために、著者らは以下の高速化を試み，成果を得ている。

（1）tcpdump[4]のパケットフィルタ処理を高速化することを試み、predicated execution[3]（述語付き実行。以下、PE）やenhanced modulo scheduling[5]（以下、EMS）を多数の条件分岐を含むループに適用できるように拡張した[7),8)]。そして、複数の入力パケットに対する処理のソフトウェアパイプライン化によって高速性を実現した。

（2）(1)を応用し、CISCO社製ルータなどに用

† 佐賀大学理工学部知能情報システム学科
Department of Information Science, Saga University
†† 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

```
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)

```
for(int i = 0; i < フィルタパターン数; i++){
  SIP = [入力パケットの sip] & [i 番目のパターンの sip の bit マスク];
  if(SIP == [i 番目のパターンの sip]){                                    //(1)
    DIP = [入力パケットの dip] & [i 番目のパターンの dip の bit マスク];
    if(DIP == [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 spt] >= [i 番目のパターンの spt の下限値]){      //(4)
          if([入力パケットの spt] <= [i 番目のパターンの spt の上限値]){    //(5)
           if([入力パケットの dpt] >= [i 番目のパターンの dpt の下限値]){   //(6)
              if([入力パケットの dpt] <= [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 action]; //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 action]; //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 C プログラム

いられる一般的な静的 IP パケットフィルタ[2),6)]を高速化できることを確認した[9)]。

しかし、これらは全て Intel IA-64 Itanium 2 プロセッサを対象にしたものであった。このプロセッサはソフトウェアパイプライン化を支援する様々なハードウェア機構 (predication、register rotation など) を有するため、このプロセッサに関する上記成果をそのまま他のプロセッサに敷衍できるか、明らかではなかった。

本研究では、x86 (IA-32) プロセッサにおいても、上記の (2) と同等の高速化が達成できたことを報告する。世の中に普及しているプロセッサのほとんど全ては x86 系であるから、本研究の与えるインパクトは小さくない。

なお、本研究は上記 (2) とは以下の点で異なる。

(3) x86 プロセッサは predication の機能を持たないから PE は実行できない。そこで、適用技法として EMS のみを用いる。

(4) x86 プロセッサは out-of-order 型であり、プロセッサの挙動をプログラマが正確に予測することは困難である。そこで本研究では、大量のソフトウェアパイプライン化コードを組織的に自動生成し、その中から最速なコードを求める。

2. フィルタルール

図 1 が本論文で検討するフィルタルールの例である。これは一般的な静的 IP フィルタルール[2),6)]に準じた記述力を持つ。図の各行がひとつのルールパターンを表しており、パケットに関する以下の情報を記述する。

- 始点 IP アドレス (以下、sip)
- 終点 IP アドレス (以下、dip)
- プロトコル番号 (以下、proto)

```
Lxxx: movq   %rsi,%r8            //  0
      andq   8(%rbp),%r8         //  1
      cmpq   0(%rbp),%r8         //  2
      je     Ltxx
      LOOPBACK(Lxxx)
Ltxx: cmpb   $6,36(%rbp)         //  3
      jne    Ltfx
Lttx: movq   16(%rbp),%mm1       //  4
      movq   %mm7,%mm0           //  5
      pcmpgtd %mm0,%mm1          //  6
      pcmpgtd 24(%rbp),%mm0      //  7
      por    %mm1,%mm0           //  8
      movd   %mm0,%r9            //  9
      testq  %r9,%r9             // 10
      je     Lttt
      LOOPBACK(Lxxx)
Lttt: movzbl 38(%rbp),%r10d      // 11
      movl   %r14d,%r11d         // 12
      andl   %r10d,%r11d         // 13
      cmpl   %r11d,%r10d         // 14
      je     Laccept
      LOOPBACK(Lxxx)
Ltfx: cmpb   $0,36(%rbp)         // 15
      je     Laccept
      LOOPBACK(Lxxx)
```

図 3　64bit 拡張モードの素朴なコード

- 始点ポート番号（以下、spt）
- 終点ポート番号（以下、dpt）
- tcp フラグ・フィールド
- フィルタ・アクション（以下、action）

パケットフィルタは、各入力パケットとルールパターンを上の行から順に照合し、全ての項目で適合したパターンの action を戻り値とする。どのパターンにも適合しないパケットはデフォルトのアクションに従う。

図 2 はこの一連の動作を C プログラム風に記述したものである。

3. コード生成の方法

3.1 概　　要

本研究で対象とする 3 種類のプロセッサ（x86 64bit 拡張モード、x86 32bit モード、Itanium 2）について以下の共通の手順を用いる。

(1) まず、人手によって図 2 を素朴なアセンブリコードに変換する。

(2) 次に、上記 (1) のコードに含まれる命令を意図的に並べ変える。並び変えた命令列はそのままでは正しいコードではないが、それをソフトウェアパイプライン化コードのカーネル部のひな形と解釈すると、多くの場合にそのひな形からソフトウェアパイプライン化コードを構築できる。

(3) 上記 (2) の並び変えを組織的に行い、様々なソフトウェアパイプライン化コードを大量に生成し、テストデータを用いて実行時間を測定し、最速のコードを求める。

```
Lxxx_ttt: movzbl -2(%rbp),%r10d  //11
          movl   %r14d,%r11d     //12
          andl   %r10d,%r11d     //13
          cmpl   %r11d,%r10d     //14
          je     Laccept
          movq   %rsi,%r8        // 0
          andq   8(%rbp),%r8     // 1
          cmpq   0(%rbp),%r8     // 2
          je     Ltxx_ttt
          LOOPBACK(Lxxx_fxx)
Ltxx_ttt: cmpb   $6,36(%rbp)     // 3
          jne    Ltfx_ttt
          ...
          中略
          ...
Lttx_ttf: movq   16(%rbp),%mm1   // 4
          movq   %mm7,%mm0       // 5
          pcmpgtd %mm0,%mm1      // 6
          pcmpgtd 24(%rbp),%mm0  // 7
          por    %mm1,%mm0       // 8
          movd   %mm0,%r9        // 9
          testq  %r9,%r9         //10
          je     Lttt_ttf
Lttf_ttf: LOOPBACK(Lxxx_ttf)
Lttt_ttf: LOOPBACK(Lxxx_ttt)
```

図 4　ソフトウェアパイプライン化コードの例
（全体のコードサイズは約 200 行）

3.2 x86 64bit 拡張モード

x86 プロセッサの 64bit 拡張モード（いわゆる Intel 64）の素朴なコードを図 3 のように与える。ここに LOOPBACK(*label*) は、ループ・カウンタ等を更新し、ループの先頭 *label* へ戻るマクロとする。

このコードを作る際に以下のことを考慮した。

64bit 演算命令、マルチメディア命令（MMX 命令）を積極的に用い、条件分岐の数を減らす。条件分岐を含むループのソフトウェアパイプライン化ではコードサイズが条件分岐の数について指数的に増加し、命令キャッシュ溢れの恐れがあるからである。図 2 は 9 個の if 文を含むが、

図3ではそれを5個の条件分岐命令に削減した。

図3のコード中の相異なる基本ブロック間でレジスタを共用しない。ソフトウェアパイプライン化では複数の基本ブロックが並列実行される可能性があるからである。

次に、図3のコードの主要な命令に0から15を付番する（図3の右端）。そして、これら15個の数字の並びをソフトウェアパイプライン化されたループ中で各番号の命令を実行する順序と解釈する。たとえば数列：

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$$

を上記の素朴なコードの数列表現と見なす。これに対して、たとえば数列：

$$11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

は、iterationの後半で0から10の命令を実行し、次のiterationの前半で11から15の命令を実行するソフトウェアパイプライン化されたコードの数列表現と見なす。そうすると、この数列が生成するソフトウェアパイプライン化コード（カーネル部）は図4のようなものになる。

なお、任意の数列からソフトウェアパイプライン化コードが生成できる訳ではない。しかし、ある一定条件を満たす数列からソフトウェアパイプライン化コードを生成できる。そこで、そのような数列を組織的に生成し、それを逐一コードへ変換し、そのコードの実行時間を測定することで、最速コードを見つける。

3.3 x86 32bitモード

上と同様の手法をx86 32bitモードにも用いる。ただし、32bitモードでは64bit演算命令を使用できないため、素朴なコードに18命令が必要であった（コードの掲載は省略する）。

3.4 Itanium 2

文献9)と比較するために、Itanium 2プロセッサについても同等の方法で大量のコード生成を行う。素朴なコードの必要命令数は17であった。

4. 実　　験

4.1 x86 64bit拡張モード

実験に使用した3種類の計算機（著者らが研究室に所有するもの）の概略および実験結果は表1の上段の通りである。ここに実験パラメータ α $(= 2.00, ..., 69.73)$ は、簡単に言えば、実験で使用する入力パケットがフィルタルールの n 番目のパターンにマッチするときの、n の平均値である。実験では5種類の異なるフィルタルールを用いた。計算機以外の実験環境は全て文献9)と同じである。

各計算機について以下の4種類のコードの実行時間を採取した。これらは10回の実行の平均値であり、精度は $\pm$0.2MCの範囲である。

Cプログラム　図2のCプログラムを gcc -03 でコンパイルしたコード

素朴なコード　3節の素朴なコード（図3）

非ソフトウェアパイプライン化コード　大量生成するコードの中でソフトウェアパイプライン・ステージ数が1であるようなコードの中の最速なコード

ソフトウェアパイプライン化コード　大量生成するソフトウェアパイプライン化コードの中の最速なコード。

3番目の「非ソフトウェア... コード」は、iterationを超えない範囲で命令スケジューリングを行った最速なコードとも言えよう。

生成可能なソフトウェアパイプライン化コードの総数は優に数億を超える。それら全てを調べるには数ヶ月を要するから、ここでは数日以内で実験が終了するように間引いて実験を行った。

実行時間を α の一次式と仮定[9]して最小二乗法を用いて係数を求めたものが、表1の A と B である。特に A はひとつのルールパターンとの照合に掛かる時間を表しており、パターン数が十分大きいときの実行時間を決定するパラメータである。

ちなみに最速なソフトウェアパイプライン化コードを生成する数列は、表1のXeon (2.66GHz) の場合

$$0, 1, 2, 14, 15, 9, 10, 3, 11, 12, 13, 4, 5, 6, 7, 8$$

であった。上記数列は他機種では最速なコードを生成しないが、最速に近い比較的高速なコードを生成する。

4.2 x86 32bitモード

表1中段は32bitモードの実験結果である。64bit拡張モードの実験で使用した2台を含め、5種類の計算機で実験を行った。

表 1　X86 64bit 拡張モードの実行時間

	マシン		実行時間 (MC) $\alpha = 2.00, ...$					$t = A\alpha + B$	
			2.00	8.99	17.68	34.36	69.73	A	B
x86	1	C プログラム	47.4	124.5	168.0	411.8	699.2	9.8	30.0
64 bit	Xeon,	素朴なコード	31.8	69.2	178.6	318.6	609.8	8.7	11.7
	2.66 GHz,	非ソフトウェ...	33.8	73.2	176.0	303.4	577.8	8.1	17.5
	OSX 10.5.5	ソフトウェア...	32.6	68.5	108.6	180.9	354.8	4.7	23.7
	2	C プログラム	28.4	81.4	115.8	339.8	771.4	11.3	−31.4
	Core 2 Duo,	素朴なコード	29.7	65.2	169.0	296.4	542.3	7.7	16.9
	1.83 GHz,	非ソフトウェ...	29.9	71.2	176.3	304.6	543.7	7.7	22.0
	Fedora Core 9	ソフトウェア...	30.0	65.7	106.3	181.8	358.2	4.8	20.4
	3	C プログラム	48.1	123.9	167.2	409.5	696.1	9.8	30.2
	Core 2 Duo,	素朴なコード	31.3	69.3	178.5	318.2	607.4	8.6	12.0
	2.00 GHz,	非ソフトウェ...	31.5	68.6	178.5	317.6	605.9	8.6	12.0
	OSX 10.5.5	ソフトウェア...	32.2	67.9	108.0	180.8	355.0	4.7	23.1
x86	1	C プログラム	63.2	190.1	289.5	681.4	1203.0	17.1	31.8
32 bit	Xeon,	素朴なコード	42.0	92.7	206.1	321.4	685.3	9.5	18.0
	2.66 GHz,	非ソフトウェ...	42.0	92.7	206.1	321.4	685.3	9.5	18.0
	OSX 10.5.5	ソフトウェア...	45.1	97.8	158.3	267.0	542.3	7.3	28.4
	2	C プログラム	38.0	113.1	228.7	598.0	1033.2	15.2	−1.6
	Core 2 Duo,	素朴なコード	38.8	91.3	205.8	339.1	651.1	9.1	25.0
	1.83 GHz,	非ソフトウェ...	39.2	96.7	212.3	356.5	645.8	9.0	32.2
	Fedora Core 9	ソフトウェ...	39.2	90.4	149.1	262.2	527.5	7.2	22.9
	3	C プログラム	53.0	195.7	347.0	647.2	1502.3	21.3	−20.1
	Xeon,	素朴なコード	38.8	89.5	201.0	334.6	629.8	8.8	26.5
	2.13 GHz,	非ソフトウェ...	38.8	89.5	201.0	334.6	629.8	8.8	26.5
	Linux v.2.6	ソフトウェア...	40.4	90.0	150.2	263.7	523.3	7.1	24.5
	4	C プログラム	82.8	193.7	300.6	643.0	1204.6	16.8	39.3
	Core Duo,	素朴なコード	56.1	122.6	251.8	405.0	813.0	11.2	33.7
	1.66 GHz,	非ソフトウェ...	56.1	122.6	251.8	405.0	813.0	11.2	33.7
	OSX 10.5.5	ソフトウェア...	59.7	125.3	203.9	345.5	666.5	9.0	43.4
	5	C プログラム	78.3	222.2	377.6	661.9	1099.2	14.9	92.1
	Pentium 3,	素朴なコード	84.0	206.9	366.2	561.6	972.7	12.8	97.3
	850 MHz,	非ソフトウェ...	84.0	206.9	366.2	561.6	972.7	12.8	97.3
	Fedora Core 5	ソフトウェア...	86.5	164.6	289.6	464.6	819.5	10.8	78.7
IA-64	1	C プログラム	95.8	241.4	424.9	672.3	1186.3	15.8	104.2
	Itanium 2,	素朴なコード	45.6	147.6	251.9	441.4	793.3	10.9	46.8
	900 MHz,	非ソフトウェ...	46.7	118.7	204.1	327.7	594.5	8.0	47.2
	Linux v.2.4	ソフトウェア...	54.0	95.5	172.6	281.4	492.3	6.5	46.8

4.3 Itanium 2

文献9) との比較のために、Itanium 2 プロセッサについても実験を行った。表 1 の下段がその結果である。

5. 評　　価

表 2 は、実験で求めた係数 A の比較である。なお、表の最下の 2 行は文献 9) の結果である。この表から以下のことが分かる。

まず、A の値は様々であるが、高速化率 A_1/A_4、A_3/A_4 の値は、機種に依らず、64bit ／ 32bit モードそれぞれについておおむね同じ値になっている。高速化率が、クロック周波数などに依らず、主にアーキテクチャの基本構造によって決まるためであろう。

高速化率 A_1/A_4 はコード最適化の効果を知る

表 2　定数 A（ルールパターン数による実行時間増分値）の比較

種別	プロセッサ	C プ...	素朴...	非ソ...	ソフ...	高速化率		A_4 の実時間 (nsec)
		A_1	A_2	A_3	A_4	A_1/A_4	A_3/A_4	
x86 64 bit	1. Xeon (2.66G)	9.8	8.7	8.1	4.7	2.1	1.7	1.8
	2. Core 2 (1.83G)	11.2	7.7	7.7	4.8	2.3	1.6	2.6
	3. Core 2 (2.00G)	9.8	8.6	8.6	4.7	2.1	1.8	2.4
x86 32 bit	1. Xeon (2.66G)	17.1	9.5	9.5	7.3	2.3	1.3	2.7
	2. Core 2 (1.83G)	15.2	9.1	9.0	7.2	2.1	1.3	3.9
	3. Xeon (2.13G)	21.3	8.8	8.8	7.1	3.0	1.2	3.3
	4. Core (1.66G)	16.8	11.2	11.2	8.9	1.9	1.3	5.4
	5. Pentium (0.85G)	14.9	12.8	12.8	10.8	1.4	1.2	12.7
IA-64	1. Itanium 2 (0.90G)	15.8	10.9	8.0	6.5	2.4	1.2	7.2
IA-64	文献 9) (PE)	15.8	—	7.9	3.9	4.1	2.0	4.3
	文献 9) (EMS)	15.8	—	7.9	5.4	2.9	1.5	6.0

指標である。x86 64bit モードでは 3 種類全てで 2 以上である。32bit モードも 2 前後であるが、旧型のプロセッサ（たとえば Pentium 3）ではやや劣っている。

高速化率 A_3/A_4 はソフトウェアパイプライン化の効果を知る指標である。64bit モードでは約 1.7 になり、32bit モードでは約 1.3 である。64bit モードの高速化率は、著者らの前論文[9](EMS) の高速化率（$A_3/A_4 = 1.5$）を超えている。なお、本論文の手法を Itanium 2 プロセッサにそのまま適用しても良質なコードが得られていない（$A_3/A_4 = 1.2$）。

A_2 と A_3 は 64bit/32bit モード共にほとんど同じ値である。x86 プロセッサが out-of-order 型であり、狭い範囲の命令の並べ替えはプロセッサ内部で動的に最適化されるためと思われる。しかし iteration を超える広い範囲の並び替えはできないから、A_2 と A_4 の差は大きい。対照的に Itanium 2 プロセッサは in-order 型であり、プログラマ自身が最適な命令配置を行わなければ性能が出ない。結果、A_2 (= 10.9) と A_3 (= 8.0) に差がある。

6. おわりに

本論文では、著者らが Itanium 2 プロセッサ上で研究を続けてきたパケットフィルタ・プログラムのソフトウェアパイプライン化技法が x86 プロセッサについても有効であることを示した。

今後は、高速パケットフィルタを OS に組み込み、実際のネットワーク上で実用性を確かめる予定である。

また、本研究は 1 節の (2) に対応するが、(1) のニーズも依然として大きい。これも x86 プロセッサを対象として研究を進める。

参考文献

1) A. W. Appel : *Modern Compiler Implementation in C*, Cambridge University Press (1997)
2) Cisco : *Configuring IP Access Lists*, Document ID: 23602, http://www.cisco.com/warp/public/707/confaccesslists.html.
3) Intel : *Intel Itanium Architecture Software Developer's Manual*, http://www.intel.com/ design/ itanium2/ documentation.htm.
4) V. Jacobson, et. al. : *tcpdump(1), bpf...*, Unix Manual Page (1990)
5) N. J. Warter, G. E. Haab, and J. W. Bockhaus : Enhanced Modulo Scheduling for Loops with Conditional Branches, IEEE MICRO-25 (1992).
6) 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.
7) Y. Yamashita and M. Tsuru : Code Optimization for Packet Filters, SAINT2007 CD-ROM (2007).
8) Y. Yamashita and M. Tsuru : Software Pipelining for Packet Filters, LNCS 4782 (2007) pp. 446 - 459.
9) 山下義行、鶴正人：高速パケットフィルタの実装と評価、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 (2008) pp.1-11.