

# Software Pipelining for Packet Filters\*

Yoshiyuki Yamashita<sup>1</sup> and Masato Tsuru<sup>2</sup>

<sup>1</sup> Saga University, Honjyo 1, Saga, 840-8502 Japan

yaman@is.saga-u.ac.jp

<sup>2</sup> Kyushu Institute of Technology, Kawazu 680-4, Iizuka, 820-8502 Japan

tsuru@ndrc.kyutech.ac.jp

**Abstract.** Packet filters play an essential role in traffic management and security management on the Internet. In order to create software-based packet filters that are fast enough to work even under a DOS attack, it is vital to effectively combine both the higher-level optimization related to algorithmic structure and the lower-level optimization related to acceleration techniques in compiler study. In the present paper, we focus on the lower-level (machine code) optimization using software-pipelining, and report experimental results that indicate the potential of our approach for accelerating packet filter performance. The technical difficulty is that the packet filter is a lump of conditional branches, so that standard optimization techniques usually applied to basic blocks is not directly applicable to this problem. Using predicated execution and enhanced modulo scheduling, we solve this problem and achieve 20 times higher performance compared with a conventional interpreter-based packet filter. We also compare the proposed filters and compiler-based packet filters, and obtain a better than two-fold increase in performance.

## 1 Introduction

Packet filters, which basically inspect the header and/or payload of each incoming packet and perform appropriate actions on each packet, are essential for traffic management and security management on the Internet and so are implemented in a variety of systems/devices, such as IP routers and firewalls. Software-based packet filters are cost-effective and flexible but are general relatively slow, whereas hardware-based packet filters (e.g., those using ASIC or FPGA[8,11,6] for pattern matching) are fast but expensive and less flexible.

Recently, the rapid growth of network bandwidth has led to the requirement for high-speed packet filters. On the other hand, emerging applications of packet filters require increases scalability and flexibility of filter rules. In particular, in the case of DOS attacks, it is important to drop an enormous number of useless packets efficiently, while valid packets should be handled properly. In addition, these filters should be easily modified in response to new circumstances or new

---

\* This work was supported in part by Hitachi, Ltd, the Ministry of Internal Affairs Communications, Japan, and JSPS, Grant-in-Aid for Scientific Research (C) (19500057).

requirements. In order to realize a packet filter that manages both flexibility and high-speed in a cost-effective manner, it is of practical importance to establish an approach by which to create software-based packet filters that are fast enough to work even when under DOS attack. This requires the effective combination of both higher-level optimization related to algorithmic structure adapted to the input packet sequence and lower-level (machine code) optimization related to acceleration techniques in a compiler study.

We will focus on the lower-level optimization. Several studies have attempted to produce a native machine code from a packet filter rule and to make the filter faster compared with the conventional interpreter-based packet filter[2,3,7]. However, to the best of our knowledge, none of these studies attempted to apply the state-of-the-arts optimization techniques based on software-pipelining. Although software-pipelining is a common technique in compiler construction, it is technically difficult to apply software-pipelining to a packet filter because the filter consists of several (more than 20 in some cases) conditional branches, but the standard software-pipelining algorithm is not applicable to programs with conditional branches. We solve this program with predicated execution[4] and enhanced modulo scheduling[9].

The present paper is an extension of our previous research [10], which showed that we achieved from 10 to 20 times higher performance. In the present paper, we will show that further improvement of the algorithms and more appropriate choices of scheduling parameters result in 12 to 28 times higher performance and an average of 20 times higher performance. In addition, in order to explain our optimization techniques in detail, we present concrete code examples in the C language and the assembly language using an Intel IA-64 Itanium 2 processor.

## 2 Framework

For the purpose of simplification, we consider a packet filtering procedure to consist of two serial procedures. The first procedure classifies each received packet into one of three categories: usually *accept*, *drop*, or *forward*, according to pre-defined filter rules and the context of the packet. The second procedure is to process the packet according to the above classification. These two procedures are usually invoked serially within a single loop, such as

```

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

where we assume that when a packet stream is incoming at a very high rate (e.g., due to a DOS attack), two or more consecutive packets are received by each `receive()` and are stored in the receiving buffer, after which they are continuously processed in the internal loop.

Simply speaking, making a packet filter faster is approximately equivalent to executing this loop in a shorter time. Hence, in the present paper, we apply code optimization techniques to this loop. The function `filter()` in the above loop usually consists of only logical and arithmetic operations. Therefore, we expect that it is rather easy to optimize this function with various code optimization techniques. On the other hand, since the function `action()` includes complex tasks, the function must perform tasks such as hoist a packet to an upper-layer and send a packet to other network ports, it is almost impossible to apply optimization techniques to this function. Then, in order to obtain the maximum effects of optimization, in other words, to enable the application of loop optimization techniques, we divide the above loop into two loops as follows:

```

for(;;) {
    n_packets = receive();
    for(i = 0; i < n_packets; i++){    // Loop1
        result[i] = filter(i);
    }
    for(i = 0; i < n_packets; i++){    // Loop2
        action(result[i]);
    }
}

```

We therefore concentrate on how to optimize `Loop1` efficiently.

The main idea is to process multiple consecutive packets together using a software-pipelined procedure. This approach is especially effective in suffering from DOS attacks, where a major problem is how to prevent the unprocessed packets from being incessantly lost due to buffer overflow, that is, how to increase the maximum acceptable input packet rate. Consider a very simple analytical model. Let  $T_1$  and  $T_2$  be the average times of executing (non-optimized) `filter()` for one packet and `action()` in cases of *accept* and *forward* for one packet, respectively. We also assume the execution time of `action()` in case of *drop* is negligible. Then, let  $\alpha$  denote the ratio of non-dropped (valid) packets to all of the incoming packets. In addition, let  $A$  denote the acceleration ratio of the execution time in optimizing `Loop1`, when many consecutive packets are processed together. The average processing time per packet is then  $T_1/A + \alpha T_2$ . In a DOS attack situation, we can assume that  $\alpha \ll 1$  because the number of received packets is huge but the number of valid packets does not vary from a non-DOS attack situation. Therefore, the ratio of the optimized processing time to the non-optimized processing time is approximately given as  $(T_1/A + \alpha T_2)/(T_1 + \alpha T_2) \approx 1/A$ . It shows that the code optimization in `Loop1` directly affects the total performance of the packet filter.

### 3 Loop Optimization

In this section, we briefly explain the three steps of the optimizations compared herein. Our proof-of-concept prototype of the packet filter is based on `tcpdump`[5] (or `bpf` as its basis). Actually, the compilers for packet filters in our experiments are implemented by rewriting the source program of `tcpdump`.

List 1: `bpf`'s filter machine code for rule, `ip and tcp`

```

(000)  ldh      [12]
(001)  jeq     #0x800   jt 2   jf 5
(002)  ldb     [23]
(003)  jeq     #0x6     jt 4   jf 5
(004)  ret     #96
(005)  ret     #0

```

### 3.1 Interpreter

`tcpdump` translates a filter rule into a code for the pre-defined virtual machine, called a *filter machine*, which consists of an accumulator, an index register, and an array of scratch memory. `tcpdump` then executes the code on the interpreter that simulates the filter machine. The code is a sequence of `bpf` instructions. `bpf` instructions can load a part of a packet contents into a virtual register, perform arithmetic on virtual registers, and return an integer value indicating whether a packet is accepted.

For example, let us consider the `tcpdump` rule, `ip and tcp`, which accepts only IPv4 tcp packets. List 1 shows the corresponding filter machine code of `bpf` instructions. Here, the instructions `ldh`, `ldb`, `jeq`, and `ret` denote load a half word, load a byte, jump if equal, and return a constant value, respectively. The tags `jt m` and `jf m` denote that the execution jumps to location `m` when the comparison yields *true* and *false*, respectively. This code corresponds to the function `filter(i)` in the previous section.

### 3.2 Compiling into a C Program

Since `tcpdump` translates a filter rule into a filter machine code and executes the code on an interpreter, the filtering process is very slow. The process generally becomes faster when we compile the rule into a native code. Several studies [3,7] have already reported that the filter becomes approximately five times faster. As the first step of our research, we compile the rule into a native code. Specifically, we translate a filter rule into a C program and compile the C program into a native code using an existing C compiler.

In the present paper, we examine two types of C programs because we expect that compilers treat them differently. The first type is a C program in which each `bpf` instruction in a filter machine code is straightforwardly replaced with equivalent C statements, including labels and `goto` statements. For example, List 2 is the C program translated from List 1. The other type of C program is a structured program without `goto` statements, like List 3. Structured programs are sometimes more redundant than the unstructured programs but are expected to be compiled into more efficient codes by highly-optimizing C compilers.

Rewriting the source program of the `bpf` interpreter, we can easily build a translator that translates a given filter machine code into both unstructured and structured C programs. The C compilers the authors used in this research are `gcc`, the Gnu C compiler, and `icc`, Intel's C compiler.

List 2: unstructured C program for List 1

```

for(i = 0; i < n_packets; i++) {
    byte* p = packets[i];
    L0: A = EXTRACT_SHORT(&p[12]);
    L1: if(A == 0x800) goto L2; else goto L5;
    L2: A = p[23];
    L3: if(A == 6) goto L4; else goto L5;
    L4: A = 96; goto L6;
    L5: A = 0; goto L6;
    L6: result[i] = A;
}

```

List 3: structured C program for List 1

```

for(i = 0; i < n_packets; i++) {
    byte* p = packets[i];
    short tmps = EXTRACT_SHORT(&p[12]);
    if(tmps == 0x800) {
        byte tmpb = p[23];
        if(tmpb == 6) A = 96;
        else A = 0;
    } else A = 0;
    result[i] = A;
}

```

### 3.3 Simple List Scheduling

A C compiler sometimes generates slow code because the compiler has little information specific to a given C program so that it is likely to select a conservative but inefficient instruction pattern. As the second step of our research, we build a translator that directly translates a filter machine code into a native assembly code using elaborately selected instruction patterns. A simple instruction scheduling, so-called *list scheduling*, is applied to individual basic blocks.

List 4 is an example of such simply list-scheduled code translated from List 3, based on the Intel IA-64 architecture. Here, we assume that the registers `r100` and `r110` hold the addresses of arrays `packets` and `results` (see List 2), respectively, and that `r10` and `r20` hold the constant values `0x800` and `96`, respectively. For ease of reading, in the present paper, every register used in the codes is singly assigned, although register allocation algorithms are applied to actual codes. Double semicolons `;;` indicate a boundary of adjacent instruction groups, in each group of which all of the instructions can be executed simultaneously.

### 3.4 Software Pipelining

As the third step of our optimization, we apply software-pipelining techniques[1], which are highly sophisticated aggressive instruction scheduling techniques for

List 4: naive code for List 3

```

L0:  ld8 r32 = [r100],8      ;; // r100 == &packets[i]
     adds r40 = 12,r32      ;;
     ld2 r50 = [r40]        ;;
     cmp.eq p6,p7 = r10,r50 // r10 == 0x800
     (p7)br L2              ;;
     adds r60 = 23,r32      ;;
     ld1 r70 = [r60]        ;;
     cmp.eq p6,p7 = 6,r70
     (p7)br L1              ;;
     mov r80 = r20          // r20 == 96
     br L3                  ;;
L1:  mov r80 = r0
     br L3                  ;;
L2:  mov r80 = r0           ;;
L3:  st4 [r110] = r80,4     // r110 == &results[i]
     br.ctop.dptk L0

```

loops. Although software pipelining techniques are popular and have been implemented for several existing compilers, including `gcc`, most of the compilers can only apply the techniques to loops whose bodies are basic blocks, but cannot apply the techniques to loops with conditional branches like List 2 or List 3.

In order to obtain software-pipelined codes for loops with conditional branches, we adopt two special techniques. The first is called *predicated execution*[4] (PE for short), which can be performed by processors with the facility of predicate registers, such as the Intel Itanium 2 [4]. The second is called *enhanced modulo scheduling* (EMS for short) [9], which can be performed by any RISC processor, but generates code of larger size. Since these techniques are the primary theme in the present paper, we will explain them in detail in the following two sections.

## 4 Predicated Execution

In this section, we briefly explain the basic concepts of PE and the software-pipelining technique based on PE.

### 4.1 Predication

Predicate registers are used to eliminate branch instructions, which may seriously slow program execution. List 5, translated from List 4, is an example of code that uses predicate registers. For example, the instruction `cmp.eq p10,p20 = r10,r50` assigns true and false values to predicate registers `p10` and `p20`, respectively, if the values of `r10` and `r50` are equal. Otherwise, this instruction assigns false and true values to predicate registers `p10` and `p20`, respectively. The instruction `(p10)adds r60 = 23,r32` performs addition if `p10` is true. Otherwise, this instruction works as a `nop` instruction, and the instruction is said to be *nullified*. The compare instruction `(p10)cmp.eq.unc p30,p40 = 6,r70`, which is qualified with the predicate register `p10`, assigns the appropriate truth values

List 5: naive PE code for List 3

```

L0:  ld8 r32 = [r100],8          ;;
      adds r40 = 12,r32         ;;
      ld2 r50 = [r40]           ;;
      cmp.eq p10,p20 = r10,r50 ;;
      (p10)adds r60 = 23,r32    ;;
      (p20)mov r80 = r0         ;;
      (p10)ld1 r70 = [r60]     ;;
      (p10)cmp.eq.unc p30,p40 = 6,r70 ;;
      (p30)mov r80 = r20       ;;
      (p40)mov r80 = r0         ;;
      st4 [r110] = r80,4
      br.ctop.dptk L0

```

List 6: software-pipelined PE code (kernel) for List 3

```

L0:  ld8 r32 = [r100],8          //stage //1
      (p12)cmp.eq.unc p30,p40 = 6,r71 //4
      (p11)adds r60 = 23,r34       //3
      ld2 r50 = [r41]             //2
      (p41)mov r82 = r0           //5
      (p21)mov r80 = r0           ;; //3
      (p11)ld1 r70 = [r60]       //3
      cmp.eq p10,p20 = r10,r50   //2
      adds r40 = 12,r32         //1
      st4 [r110] = r82,4        //5
      (p30)mov r81 = r20        //4
      br.ctop.dptk L0

```

to p30 and p40 according to the results of comparison if p10 is true. Otherwise, the instruction assigns false values to both p30 and p40.

## 4.2 Software pipelining for Predicated Code

Since a loop body without branches is a straight-line code, we can optimize the loop by applying a standard method of software-pipelining[1] (or modulo scheduling as a concrete algorithms). List 6 is such a software-pipelined predicated code (hereinafter PE code) optimized from List 5 under the assumption that every memory access latency is one machine cycle (MC). In order to minimize the initiation interval of the loop iterations (hereinafter I.I.), we use register rotation[4] for both predicate and integer registers. In List 6, for example, just after the loop-back branch instruction `br.ctop` is executed, the predicate registers p10, ..., p40 are renamed p11, ..., p41, respectively, and the integer registers r32, ..., r81 are renamed r33, ..., r82, respectively. The code in List 6 consists of five software pipeline stages, denoted 1 through 5 following the corresponding instruction at each line of List 6.

The idealistic I.I. of the loop in List 6 is 2 MC because the first six instructions of the loop can be executed simultaneously and so the last six instructions can also be executed, although it is almost impossible to attain I.I. because the memory access latency is longer than 1 MC and often varies greatly.

A major benefit of using PE is that PE frees us from branches (and branch penalties) and can facilitate the application of software-pipelining. Conversely, a major disadvantage is that the instructions that would be nullified do not contribute any effective computation and waste processor resources. In general, a PE code becomes slower when the code size (precisely speaking, the number of nullified instructions) increases.

## 5 Enhanced Modulo Scheduling

Since it is difficult to explain the EMS algorithm[9] in detail in the limited space available here, we only present the concept of how the processor runs effectively on the code generated by the EMS algorithm. The EMS algorithm generates

List 7: EMS code (kernel) for List 3

```

Lxx_tx:                //stage      Lfx_tt:                //stage
  ld8 r32 = [r100],8    //1          ld2 r50 = [r40]        //1
  st4 [r110] = r82,4    //4          mov r81 = r20         //3
  cmp.eq p6,p7 = r10,r51 //2          br.ctop.dptk Lxx_xx
  (p7)br Lfx_tx        ;;          br Lexit             ;;

Ltx_tx:                Lfx_tf:
  adds r40 = 12,r32     //1          ld2 r50 = [r40]        //1
  adds r60 = 23,r33     //2          mov r81 = r0         //3
  cmp.eq p6,p7 = 6,r71 //3          br.ctop.dptk Lxx_xx
  (p7)br Ltx_tf        ;;          br Lexit             ;;

Ltx_tt:                Lxx_xx:
  ld2 r50 = [r40]       //1          ld8 r32 = [r100],8    //1
  mov r81 = r20         //3          st4 [r110] = r82,4    //4
  ld1 r70 = [r60]       //2          cmp.eq p6,p7 = r10,r51 //2
  br.ctop.dptk Lxx_tx   //2          (p7)br Lfx_xx        ;;
  br Lexit              ;;          Ltx_xx:
Ltx_tf:                adds r40 = 12,r32       //1
  ld2 r50 = [r40]       //1          adds r60 = 23,r33     ;; //2
  mov r81 = r0         //3          ld2 r50 = [r40]       //1
  ld1 r70 = [r60]       //2          ld1 r70 = [r60]       //2
  br.ctop.dptk Lxx_tx   //2          br.ctop.dptk Lxx_tx
  br Lexit              ;;          br Lexit             ;;

Lfx_tx:                Lfx_xx:
  adds r40 = 12,r32     //1          adds r40 = 12,r32     //1
  mov r80 = r0         //2          mov r80 = r0         ;; //2
  cmp.eq p6,p7 = 6,r71 //3          ld2 r50 = [r40]       //1
  (p7)br Lfx_tf        ;;          br.ctop.dptk Lxx_xx ;;

Lexit:

```

continues to the right column

such a code. Strictly speaking, the EMS algorithm originally given in [9] treats a loop having a few unnested conditional branches and is not sufficient to optimize the loops with *many* (more than 20 in the present cases, as shown in the next section) deeply-nested (maximum depth: 10) conditional branches. The authors are now preparing another paper that explains how to treat such complex loops.

### 5.1 Difference from Predicated Execution

A software-pipelined PE code uses a set of predicate registers to memorize trace information of the evaluation results of compare instructions. Review the code in List 6. The rotating predicate register `p10` is defined at the second stage and is referred at the third and fourth stages as `p11` and `p12`, respectively. `p20` is defined as the second stage and referred at the third stage as `p21`. `p30` is defined and referred at the fourth stage. `p40` is defined at the fourth stage and referred at the fifth stage as `p41`. Thus, at the beginning of every new iteration, the predicate information that must live over iterations is held in the set of the registers, (`p11`, `p12`, `p21`, `p41`), so that there are at least 16 ( $=2^4$ ) patterns of code executions for the one code in List 6.

The EMS algorithm does not use predicate register, but instead generates two or more code patterns corresponding to the set of the truth values the predicate registers possess in the PE code.

### 5.2 Code Example

List 7 is the kernel part of the software-pipelined code generated with the EMS algorithm (hereinafter EMS code). Each basic block in the code has a label of form  $L_{a_1b_1}a_2b_2$ , where the letters  $a_i$  and  $b_i$  represent the truth values of the results of the comparisons `cmp.eq p6,p7 = r10,r51` and `cmp.eq p6,p7 = 6,r71`, respectively, for the  $i$ -th past iteration. The letters ‘t’, ‘f’, and ‘x’ denote the true, false, and undefined (or useless) values, respectively. For example, the label `Lxx_tx` at the first line in List 7 represents the situation in which no comparison has yet been evaluated for the most recent iteration but the comparison `cmp.eq p6,p7 = r10,r51` has been evaluated affirmatively for the second most recent iteration. If a series of adjacent packets are all IPv4 tcp packets, the program traverses only the three basic blocks labeled `Lxx_tx`, `Ltx_tx`, and `Ltx_tt`. If none of the packets is a IPv4 tcp packet, the program traverses the basic blocks labeled `Lxx_xx` and `Lfx_xx`. Otherwise, the trace becomes more complicated. Under the assumption that there is no branch penalty, the ideal I.I. of the code in List 7 is 3 MC, even if the processor runs on any path.

In contrast to the PE code, the EMS code is not free from branch penalty in general but rather contains no nullified instructions because the EMS code only executes the instructions required for the computation. Therefore, the EMS code is faster than the PE code if the penalty of the nullified instructions is greater than the branch penalty.

**Table 1.** Filter Rule in Our Experiments

No	Rule	depth	#ifs	#insts	matching ratio
1	ip	1	1	7	100.0%
2	ip and tcp	2	2	11	99.8%
3	tcp	3	3	17	99.8%
4	ip and udp	2	2	11	0.2%
5	udp	3	3	17	0.2%
6	ip and dst net <ip_address>	2	2	15	4.9%
7	dst net <ip_address>	4	6	36	4.9%
8	ip and net <ip_address>	3	3	22	10.9%
9	net <ip_address>	5	9	59	10.9%
10	ip and dst port <number>	7	19	66	9.3%
11	dst port <number>	8	29	88	9.3%
12	ip and port <number>	9	19	90	21.41%
13	port <number>	10	29	125	21.41%

## 6 Experiments

This section reports the experimental results obtained using the code optimization methods explained in Sections 3.1, 3.2, 3.3, 4, and 5.

### 6.1 Environment

Table 1 shows 13 examples of the filter rules of `tcpdump` examined herein. Each rule is simple, short, and uni-functional, although, in reality, a longer rule that combines these simple rules is likely to be employed. This is because examining each simple rule separately is suitable for analyzing the effect of our optimizations in depth. In order to clarify how the rule is complex to optimize, the columns *depth*, *#ifs*, and *#insts* in the table show three characteristics of the structured C program in Section 3.2 and the native assembly code in Section 3.3 corresponding to each rule. The depth is the maximum depth of nests of `if` constructs that the program contains, *#ifs* is the number of `if` constructs that the program contains, and *#insts* is the number of assembly instructions not including branch instructions.

The sample packets used in the experiments were captured from the network of the authors' lab using `tcpdump -w`. The total number of packets was 10,000. The column *matching ratio* in Table 1 is the ratio of the packets accepted by each rule. In our experiments, the 10,000 packets are loaded on a large buffer (virtual network). Then, every `n_packets` packets are copied from the virtual network buffer to the receiving buffer and are processed by `Loop1`  $10,000/n\_packets$  times (recall Section 2). The total execution time is then divided by 10,000 to obtain the execution time per packet for the given `n_packets`. For performance evaluation purposes, we use the execution time for the optimal `n_packets`, which varies depending on the filter rule. Experiments were performed on an Intel IA-64 Itanium 2 processor (900 MHz, revision 7), Linux version 2.4.18-1.

**Table 2.** Execution Time per Packet (MC)

No	$T_{ip}$	$T_{gcc}^u$	$T_{gcc}^s$	$T_{icc}^u$	$T_{icc}^s$	$T_{sls}$	$T_{pe}$	$T_{ems}$	$T_{hyb}$
1	80.6	17.5	17.6	13.4	7.6	11.2	4.3	4.3	4.3
2	126.7	19.4	19.5	15.5	8.6	14.1	4.5	5.8	4.5
3	143.6	20.6	21.7	16.2	9.6	15.3	5.5	7.5	5.5
4	126.5	19.5	19.6	14.6	8.6	14.1	4.5	7.7	4.5
5	143.6	20.6	21.7	16.2	9.6	15.3	5.5	7.6	5.5
6	132.5	188.1	188.8	184.4	189.5	18.7	6.9	10.4	6.9
7	132.7	190.6	189.8	187.1	184.9	19.0	10.7	9.9	10.7
8	181.8	332.0	332.0	330.1	329.2	24.9	9.1	13.5	9.1
9	181.6	333.7	333.5	331.4	330.5	25.2	18.2	14.4	14.4
10	266.2	54.3	53.6	32.5	34.5	30.0	18.6	15.1	15.1
11	283.2	55.1	55.4	30.7	35.4	31.1	21.2	16.9	16.9
12	309.5	64.6	67.3	37.1	40.6	37.2	22.2	20.5	20.5
13	326.5	66.7	69.4	37.6	39.5	38.0	28.8	22.2	22.2

## 6.2 Results

Table 2 shows the average execution time per packet for Loop1 (in units of machine cycle time). Here,  $T_{ip}$  is the execution time of the original filter function of `tcpdump`,  $T_{gcc}^u$  and  $T_{gcc}^s$  are the execution times of unstructured and structured C programs, such as List 2 and List 3, respectively, compiled by `gcc` 2.96 with a `-O3` option,  $T_{icc}^u$  and  $T_{icc}^s$  are the execution times of unstructured and structured C programs compiled by Intel `icc` 9.1 with a `-O3` option,  $T_{sls}$  is the execution time of the simply list-scheduled code, such as List 4,  $T_{pe}$  is the execution time of the software-pipelined PE code, such as List 6, and  $T_{ems}$  is the execution time of the EMS code, such as List 7. Since a software-pipelined code and its execution time can vary greatly depending on the memory access latency, we profiled the possible execution times in terms of the memory access latency and selected the optimal execution time as  $T_{pe}$  (or  $T_{ems}$ ). Table 3 shows the acceleration ratios  $A_x = T_{ip}/T_x$ , and Figure 1 shows this information in the form of a bar graph. From the tables, we observe the following:

The ratio  $A_{gcc}^u$  is approximately 5, except for cases No. 5 through No. 8, which agrees with the results reported in [7]. The ratio  $A_{gcc}^s$  is approximately  $A_{gcc}^u$  for all cases, because `gcc` applies no special optimization to loops with conditional branches, regardless of their program structures.  $A_{icc}^s$  is approximately twice as large as  $A_{icc}^u$  for cases No. 1 through No. 5, because `icc` optimizes the loops of which the bodies are small and well-structured by applying software pipelining with predicated execution but sacrifices optimization when the program size is sufficiently large.

The ratio  $A_{pe}$  shows that the PE code is at least 10 times faster and approximately 18 times faster on average than the interpreter-based execution. This ratio decreases as the number `#insts` increases. The ratio  $A_{ems}$  shows that

**Table 3.** Acceleration Ratio ( $A_x = T_{ip}/T_x$ )

No	$A_{ip}$	$A_{gcc}^u$	$A_{gcc}^s$	$A_{icc}^u$	$A_{icc}^s$	$A_{sls}$	$A_{pe}$	$A_{ems}$	$A_{hyb}$
1	1.0	4.6	4.6	6.0	10.6	7.2	18.7	18.7	18.7
2	1.0	6.5	6.5	8.2	14.7	8.9	28.2	21.8	28.2
3	1.0	7.0	6.6	8.9	15.0	9.4	26.1	19.1	26.1
4	1.0	6.5	6.5	8.7	14.7	9.0	28.1	16.4	28.1
5	1.0	7.0	6.6	8.9	15.0	9.4	26.1	18.9	26.1
6	1.0	0.7	0.7	0.7	0.7	7.1	19.2	12.7	19.2
7	1.0	0.7	0.7	0.7	0.7	7.0	12.4	13.4	12.4
8	1.0	0.5	0.5	0.6	0.6	7.3	20.0	13.5	20.0
9	1.0	0.5	0.5	0.5	0.5	7.2	10.0	12.6	12.6
10	1.0	4.9	5.0	8.2	7.7	8.9	14.3	17.6	17.6
11	1.0	5.1	5.1	9.2	8	9.1	13.4	16.8	16.8
12	1.0	4.8	4.9	8.3	7.6	8.3	13.9	15.1	15.1
13	1.0	4.9	4.7	8.7	8.3	8.6	11.3	14.7	14.7
ave.	1.0	5.7*	5.6*	8.3*	11.3*	8.3	18.6	16.3	19.7

\* Calculated without No. 6 to No. 9.

the EMS code is at least 12 times faster and 16 times faster on average than the interpreter-based execution. Since the declining speed of  $A_{ems}$  is lower than that of  $A_{pe}$ ,  $A_{ems}$  is larger than  $A_{pe}$  when #insts is greater than 30. This will be discussed further in the following subsection.

$T_{sls}$  can be regarded as the minimum execution time (or, conversely, as the maximum execution speed) of a program to which no software-pipelining method has been applied, because in all cases except cases No. 6 through No. 9,  $T_{sls}$  is approximately  $T_{icc}^u$ , which is the execution time of the program compiled by the Intel compiler optimized for the Itanium 2 processor. Then,  $A_{ems}$  and  $A_{pe}$  are approximately twice as large as  $A_{sls}$  on average. Roughly speaking, this ratio with respect to  $A_{sls}$  indicates the effect of software-pipelining.

The values in entries {No. 6, ..., No. 9}  $\times$   $\{T_{gcc}^u, T_{gcc}^s, T_{icc}^u, T_{icc}^s\}$  are extraordinarily large compared to the corresponding values for  $T_{ip}$ . This is because memory access conflicts occur on the Itanium 2 machine used in this study, and the compilers gcc and icc can never recognize this symptom. The algorithms associated with  $T_{sls}$ ,  $T_{pe}$ , and  $T_{ems}$  manage to avoid this conflict by using a special load/store pattern.

### 6.3 Comparison of PE and EMS Algorithms

In general,  $T_{pe}$  is smaller than  $T_{ems}$  in simple rules, (cases No. 1 through No. 5) and conversely  $T_{ems}$  is smaller than  $T_{pe}$  in complex rules. (cases No. 10 through No. 13). This is because the PE code is affected by the number of nullified instructions, which is usually proportional to the code size (or the complexity of the rule). On the other hand, the EMS code contains no nullified instruction, but rather branch instructions that cause branch penalties. In a simple rule, the

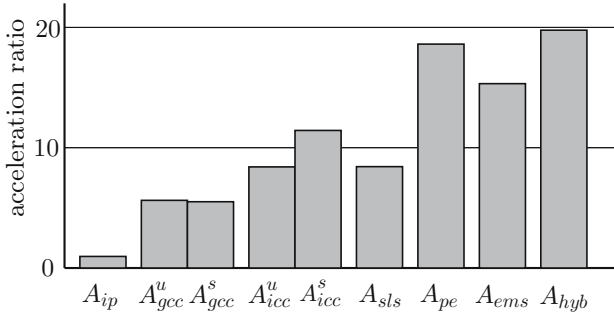


Fig. 1. Average Acceleration Ratios

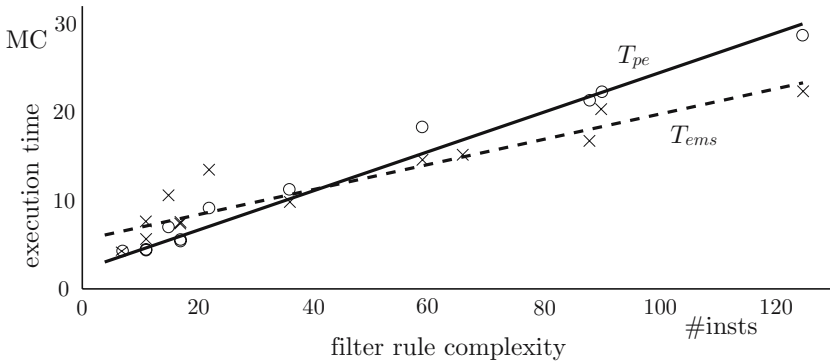


Fig. 2. Linear Approximation of Execution Times

branch penalties dominate the slowdown of the execution but, in a complex rule, the nullified instructions dominate the slowdown.

Figure 2 shows the plots of all  $T_{pe}$  (shown as circles in the graph) and  $T_{ems}$  (shown as cross-hairs in the graph) in terms of #insts. Using the least mean square method, we can approximate  $T_{pe}$  and  $T_{ems}$  as the following liner equations:

$$T_{pe} = 0.22\#insts + 2.93, \quad T_{ems} = 0.14\#insts + 6.01$$

as in Figure 2 with considerably small errors. The root mean square errors of these equations are 1.13 and 1.86, respectively. The intersection of these two lines is located at #insts = 39.3. Therefore, we can alternatively select the better software-pipelining algorithm according to the size of #insts. The execution time  $T_{hyb}$  in Table 2 is a value obtained alternatively from  $T_{pe}$  and  $T_{ems}$  and so is the acceleration ratio  $A_{hyb}$  in Table 3.

Consequently, this hybrid algorithm is approximately 20 (=19.7) times faster than the interpreter-based execution on average, 3.5 times faster than gcc-based execution, and 1.7 times faster than icc-based execution.

## References

1. Appel, A.W.: *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge (1997)
2. Begel, A., McCanne, S., Graham, S.: BPF+: Exploiting Global Data-Flow Optimization in a Generalized Packet Filter Architecture. In: *ACM SIGCOMM 1999* (1999)
3. Cristea, M.L., Bos, H.: A Compiler for Packet Filters. In: *Proceedings of ASCI 2004* (2004)
4. Intel Itanium Architecture Software Developer's Manual, <http://www.intel.com/design/itanium2/documentation.htm>
5. Jacobson, V., et al.: `tcpdump(1)`, `bpf.`, Unix Manual Page (1990)
6. Kumar, S., Dharmapurikar, S., Yu, F., Crowly, P., Turner, J.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection In: *ACM SIGCOMM 2006* (2006)
7. Okumura, T., Mossé, D., et al.: Network QoS Management Framework for Server Clusters An End-Host Retrofitting Event-Handler Approach using Netnice. In: *3rd Int. Symp. on Cluster Computing and the Grid* (2003)
8. Singh, S., Baboescu, F., Varghese, G., Wang, J.: Packet Classification Using Multidimensional Cutting. In: *ACM SIGCOMM 2003* (2003)
9. Warter, N.J., Haab, G.E., Bockhaus, J.W.: Enhanced Modulo Scheduling for Loops with Conditional Branches. In: *IEEE MICRO-25* (1992)
10. Yamashita, Y., Tsuru, M.: Code Optimization for Packet Filters. In: *SAINT2007, Workshop on Internet Measurement Technology and its Applications to Building Next Generation Internet* (2007)
11. Yusuf, S., Luk, W.: Bitwise Optimised CAM for Network Intrusion Detection Systems. In: *Int. Conf. Field Programmable Logic Appl.* (2005)