PcapWT: An Efficient Packet Extraction Tool for Large Volume Network Traces

Young-Hwan Kim[a,*], Roberto Konow[b,c], Diego Dujovne[b], Thierry Turletti[a], Walid Dabbous[a], Gonzalo Navarro[c]

[a] *INRIA, France*
[b] *Escuela de Informatica y Telecomunicaciones, Universidad Diego Portales, Chile*
[c] *Department of Computer Science, Universidad de Chile*

Abstract

Network packet tracing has been used for many different purposes during the last few decades, such as network software debugging, networking performance analysis, forensic investigation, and so on. Meanwhile, the size of packet traces becomes larger, as the speed of network rapidly increases. Thus, to handle huge amounts of traces, we need not only more hardware resources, but also efficient software tools. However, traditional tools are inefficient at dealing with such big packet traces. In this paper, we propose *pcapWT*, an efficient packet extraction tool for large traces. *PcapWT* provides fast packet lookup by indexing an original trace using a Wavelet Tree structure. In addition, *pcapWT* supports multi-threading for avoiding synchronous I/O and blocking system calls used for file processing, and is particularly efficient on machines with SSD. *PcapWT* shows remarkable performance enhancements in comparison with traditional tools such as *tcpdump* and most recent tools such as *pcapIndex* in terms of index data size and packet extraction time. Our benchmark using large and complex traces shows that *pcapWT* reduces the index data size down below 1% of the volume of the original traces. Moreover, packet extraction performance is 20% better than with *pcapIndex*. Furthermore, when a small amount of packets are retrieved, *pcapWT* is hundreds of times faster than *tcpdump*.

Keywords: Network traces, Packet indexing, Packet extraction, Wavelet

*Corresponding author
Email address: hawnious@daum.net (Young-Hwan Kim)
[1]This work has been done while the author was at INRIA.

Tree, Multi-thread file I/O

1. Introduction

The volume of network packet traces becomes larger and larger, and much more complex than before. The reason is that the speed and size of networks have increasingly expanded. Thus, much more H/W resources (e.g., massive storage space, stronger computing power, and so on) are necessary to deal with big traces including numerous packets and complex traffic patterns. Furthermore, efficient tools are required to analyze the data effectively. However, traditional tools such as *tcpdump* [1], *tcpflow* [2], or *tcpstat* [3] are inefficient to handle very large packet traces.

Among current network traffic analysis tasks such as protocol performance evaluation, network monitoring and forensic investigation, packet lookup is one of the most basic and important functions to investigate errors and to evaluate performance. In particular, packet lookup is a CPU-greedy task, especially when it deals with a huge packet trace including many complex traffic patterns. However, most of the traditional tools use linear search algorithms, which are simple, but take a long processing time proportionally to the packet trace size. Basically, a packet trace is a very long array, so the time complexity of a linear search is comparatively higher than that of other algorithms, such as binary search, balanced tree (B-tree), and hashing. Moreover, most of available packet trace formats (e.g., pcap [1]) do not include the number of matched packets and their locations. Consequently, the search delay can rapidly increase with the length of the trace and the complexity of the filtering query. In addition, traditional tools (e.g., *tcpdump* and *wireshark* [4]) are inefficient for iterative operations and possible future reuse, since they do not maintain historical data (last search result) or reusable information (index data).

In order to enhance packet trace analysis, a number of contributions have been published, such as fast packet filtering [5], packet trace compression [6, 7], and network statistics information extraction [8]. Moreover, a few practical tools have also been recently proposed, such as *PCAP-Index* [9], Compressed Pcap Packet Indexing Program (CPPIP) [10], and *pcapIndex* [11, 12]. Overall, they use extra data sets for fast search, which are extracted from the original trace.

However, these tools result in poor performance, as we show in Section 5.3. *PCAP-Index* running on top of a database requires too much space for

handling the index data, and also its packet extraction procedure is considerably slow. CPPIP can save storage space by compressing original traces, but it can only support a few simple queries. *PcapIndex* is much faster than these tools, and is able to retrieve and extract matched packets from large size trace files (up to 4.2 GB). However, the index data size built by *pcapIndex* is abnormally increased when the number of packets is in the millions. Thus, there is a trade-off between the index data size and the packet extraction performance when processing a large packet trace.

In this paper, we propose *pcapWT*, a fast packet extraction tool for large network traces, and evaluate its performance in terms of index data size and packet extraction time. *pcapWT* adopts an advanced data structure named Wavelet Tree (WT) [13], which enables a fast search and high compression ratio at the same time. Moreover, this tool supports multi-threading, which is able to enhance random file read and write performance over Solid State Drive (SSD).

This paper is structured as follows. Section 2 presents the state of the art. Section 3 provides a background on Wavelet Tree, and Section 4 describes the design of *pcapWT* in detail. Section 5 evaluates the performance of *pcapWT* and compares its performance with other tools. Finally, Section 6 concludes the paper.

2. Related Work

Today most of network trace analysis tools, such as *tcpdump* [1] and *wireshark* [4], run on a single thread, and their complexity increases linearly with the volume of original trace files. Possible solutions to improve their performance include using a higher processor clock speed, replacing a Hard Disk Drive (HDD) by a SSD, or splitting the large packet trace into multiple pieces. For that reason, several tools have been recently published to enhance performance of packet extraction on large traces, such as CPPIP [10], *PCAP-Index* [9], and *pcapIndex* [11].

CPPIP uses bgzip [14] (i.e., a block compression/decompression software) to reduce the volume of original packet trace files. This tool extracts an index data from the compressed trace file, and filters out matched packets directly from the compressed file. Although this tool is able to reduce the storage space for the original trace files, it can only support simple queries, such as packet number and received timestamp. In addition, CPPIP needs a

significant amount of space (about 7% of the original trace file) to store the index data.

PCAP-Index uses a database (SQLite Ver. 3) to build the index data and to perform packet lookup. Thus, this tool is more flexible than other command-line based tools to express queries. However, as discussed in Section 5.3, the performance obtained is poor in terms of time needed to build index data, packet lookup time and index data size.

PcapIndex, having a similar name as *PCAP-Index*, is currently part of a commercial network monitoring solution [12]. In order to reduce both index data size and packet lookup time, this tool adopts an encoding method using a bitmap compression technique based on a pre-defined codebook, named COMPressed Adaptive index [15]. Thus, it obtains better performance than CPPIP and *PCAP-Index*, in terms of index data size and packet extraction time. In comparison with *tcpdump*, this tool can reduce packet extraction time up to 1,100 times. Moreover, its index data size only takes about 7 MB per GB of the original trace file. However, as discussed in Section 5.3, *pcapIndex* is not as fast compared to what is mentioned in the paper [11], when it extracts a large amount of packets from a big trace file. In particular, the index data size rapidly increases when the volume of original trace file is greater than 4.2 GB.

3. Wavelet Tree

In this work, we present a compact index for pcap-traces that is built on top of compact data structures. To understand how these data structures are employed, we first need to define three basic operations: *rank*, *select* and *access*.

Given a sequence $\mathcal{S}$ containing n symbols from an alphabet of size σ (e.g., $\sigma = 2$ for binary sequences), $rank_b(\mathcal{S}, i)$ counts the number of occurrences of symbol b until position i, $select_b(\mathcal{S}, j)$ finds the position of the j-th occurrence of b in the sequence, and $access(\mathcal{S}, k)$ returns the symbol that is located at position k.

In the case of binary sequences, the queries described above can be solved in constant time $O(1)$ by using a bit sequence representation presented by Munro [16]. Gonzales et al [17] show that practical implementations (RG) are possible to achieve constant time for rank and access queries, and logarithmic time for select queries using $5 - 37\%$ extra space. Raman, Raman and Rao

(RRR) [18] presented another solution that is able to compress the bit sequence to $H(\mathcal{S})+o(n)$, which corresponds to the *empirical entropy* plus some low order terms. This representation is able to compress the sequence up to 50% in practice while allowing to perform rank, select and access queries in constant time. Note however that performing these operations using RRR is significantly slower than using the RG representation.

For general sequences (i.e., $\sigma > 2$), the *Wavelet Tree* [13, 19] data structure can be used. Its name is derived from an analogy with the *wavelet transform* for signals, which operates in a similar way. The wavelet tree is a perfect balanced binary tree, that stores a bit sequence in every node except the leaves. Every position of the bit sequences is marked with either a '0' or '1' depending on the value of the most significant bit of the symbol at the position in the sequence; this can be seen as dividing the alphabet into halves. Starting from the root, the symbols that are marked with a '0' go to the left subtree, while the rest go to the right subtree. This segmentation of the values continues recursively at the subtrees with the next highest bit. The tree has σ leaves, and each level of the tree requires n bits, the height of the tree is $\log_2 \sigma$, thus we need $n\lceil \log_2 \sigma \rceil$ bits to represent the tree. Each sequence of bits at every level must be capable to answer access, rank and select queries, thus we can use the RG or RRR representation to handle the bit sequences. The wavelet tree is able to solve rank, select and access operations in $O(\log \sigma)$ time, that is to say, the execution time depends only on the size of the alphabet, not on the length of the sequence. Note that both the time to execute the operations and the size of the data structure heavily depends on the size of the alphabet σ. In the case that we use RRR to represent the bit sequences at each level, the wavelet tree uses $nH(\mathcal{S}) + o(n \log \sigma)$ bits of space. If we want to compress the data structure even more, we can change the shape of the tree to the Huffman shape of the frequencies of symbols appearing in $\mathcal{S}$, and maintaining the $O(\log \sigma)$ complexity for handling the operations. In practice, if we use the Huffman shaped wavelet tree using RRR for the bitmaps, we can reduce the space of the data structure by 12% [20]. Figure 1 shows an example of a regular wavelet tree and a Huffman shaped wavelet tree built over a small sequence.

We will explain how to perform *access* operation by an example based on the regular wavelet tree from Figure 1. Let us assume that we want to perform $access(\mathcal{S}, 6)$, that is, return the symbol located at position 6. We start at position 6 at the root bit sequence (B_{root}) and ask if the corresponding bit is marked as '1' or '0'. If the bit is '0', we go to the left branch, if not, to the

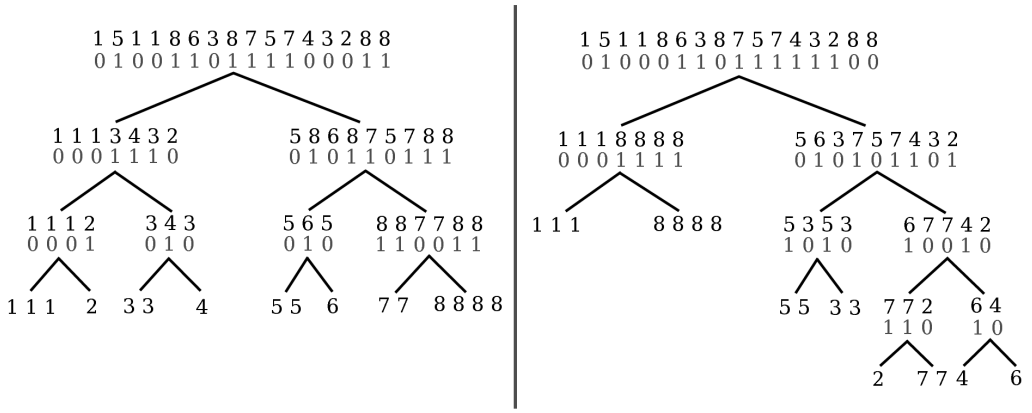

Figure 1: On the left the regular wavelet tree, on the right the Huffman shaped wavelet tree, for the sequence $\mathcal{S} = 1511863875743288$, $\sigma = 8$ and $n = 16$.

right. In this case $B_{root}[6] = 1$, so we go to the right branch of the tree. Now we have to map the original position ($k = 6$) to the corresponding position at the right branch. In other words, we want to know how many $1's$ were at the root bit sequence before position 6. We can easily do this in constant time by performing the operation supported by RG and RRR bit sequences: $rank_1(B_{root}, 6)$, which returns the value 3. Using this value we can now go to the right branch of the root and execute the same procedure by setting $k' = rank_1(B_{root}, k) = 3$. Then, we obtain the value at position k' from the right branch of the root (B_{rr}). Since $B_{rr}[3] = 0$ we go to the left branch of B_{rr} (B_{rrl}) and execute the same procedure, but this time we will count how many 0's were in B_{rr} before position k', and set $k'' = rank_0(B_{rr}, k') = 2$. We repeat the procedure by obtaining the bit value from B_{rrl} at position $k'' = 2$, since the value is 1 we know that we have to go to the right branch. It turns out that the right node of B_{rrl} is a leaf, so we are done with the traversal, and we can return the symbol located at position 6 from the sequence $\mathcal{S}$ which is 6. A very similar procedure is done to perform $rank_b(\mathcal{S}, i)$ and $select_b(\mathcal{S}, j)$. We refer the reader to the previous work from Navarro [19] and Gagie et al [21] for an extensive explanation of how these operations are performed and the virtues of wavelet trees with its wide range of possible applications.

Many researchers have studied data compression algorithms, and developed tools in the past [6, 22, 23, 24], however enhancing the compression performance is still a big challenge. Y. Liu et al. [22] proposed an interesting

information theoretic framework for compressing network packet traces, and developed different models by theoretical bounds based on the entropy of packet traces. They introduce a comprehensive guideline for developing high performance compressors for network packet traces. Our work is also based on one of the guidelines which compress the packet traces by dividing them into multiple sub-sequences containing the individual information fields obtained from the packet header (destination address, destination port, source address and others). In their work, they show that this approach is highly efficient, since individual sub-sequences tend to have low entropy values. As mentioned above, the wavelet tree is able to represent a sequence $\mathcal{S}$ to its empirical entropy $nH(\mathcal{S})$ plus low order terms, and also allows fast *access*, *rank* and *select* operations. These properties make this data structure considerably compelling for representing and indexing individual fields from the packet trace as individual sequences. We explain how we use the wavelet tree data structure and the design of our index in the following sections.

4. The design of pcapWT

In this section, we describe how to build the index data, and explain the process for lookup querying and for extracting packets. The overall processes consist of six steps, as shown in Figures 2 through 7.

4.1. Building index data

In this work, the generated index is a data set that improves the speed of packet lookup operations on network traces. The index data is used to quickly locate packets specified by user query without the need to search for every item from the data set. The index data is created using part of packet headers or additional information possibly available.

The first step is to extract index data from a packet trace file, such as packet offset, Ethernet source (Src) / destination (Dst) addresses, IPv4 Src/Dst addresses, Src/Dst port numbers, and Layer 4 (L4) protocol type. Here, the packet offset stands for the distance in bytes between two consecutive packets, which corresponds to the packet size in bytes. The reason why the offset is replaced by the packet size is that sequentially accumulated values consume a lot of memory. In addition, such large numbers are inefficient for the WT compression, since they increase the required number of bits to represent them. As shown in Figure 2, all index data are stored into each

array, and they include the same number of elements with the number of total packets in the source trace file.

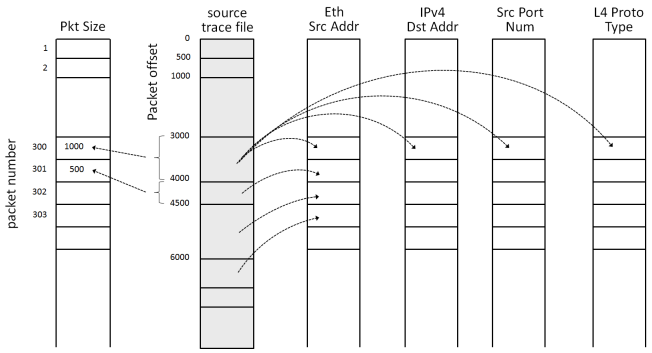

Figure 2: Extracting index data from a source packet trace file, such as packet size, Ethernet Src/Dst address, IPv4 Src/Dst address, Src/Dst port number, and Layer 4 protocol type.

However, as mentioned in Section 3, the size of the WT compressed data is highly affected by the number of bits per element and the number of elements. For instance, when an element set of index data is $S = \{100, 200, 200, 300..., n\}$, the required number of bits per element is $(\lceil log_2 \sigma \rceil)$, where σ is the maximum value among these input elements. Thus, for enhancing the compression performance, the elements have to be converted into sequential positive integer numbers ($\Sigma = \{1, 2, ..., \sigma\}$), and long arrays must be divided into multiple pieces ($S = \{S_1, S_2, ..., S_N\}$). For example, as shown in Figure 3, IPv4 addresses are mapped into positive integer numbers in consecutive order. In addition, to increase efficiency, we use a balanced tree (B-tree) to map the addresses with a positive integer. Consequently, a trace file accompanies a number of mapping tables, and the tables are provided as part of the index data.

In the last step for building index data, as shown in Figure 4, the WT builder creates one chunk of compressed index data every one million elements, and each chunk is stored in an independent file. This value has been empirically set to minimize the compressed index data size.

Meanwhile, the packet offset is not compressed by WT, because the large value of σ causes significant degradation of the compression performance. As

an alternative, we use an efficient array provided by the *libcds* package [25], which removes redundant bits.

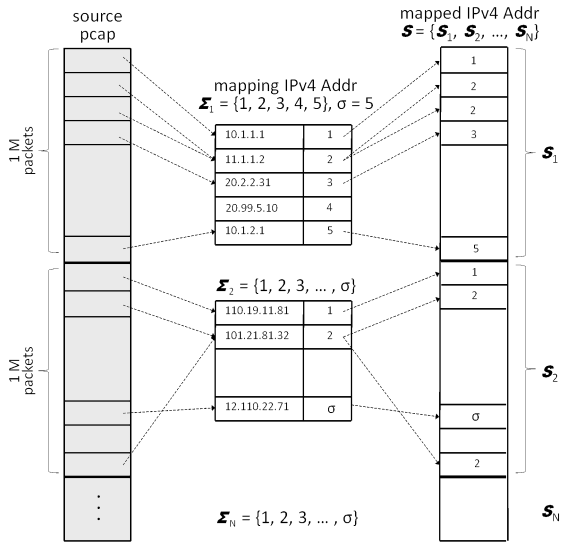

Figure 3: Mapping between the extracted index data and positive integers in consecutive order for reducing the index data size and packet lookup time.

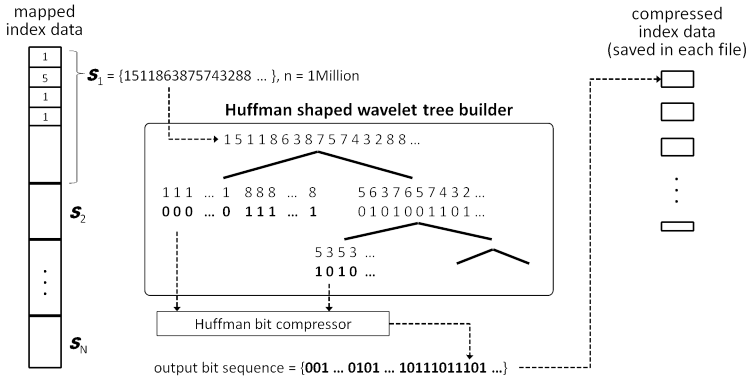

Figure 4: Building WT chunks from the mapped index data once every one million packets. This partitioning can reduce the size of index data.

4.2. Packet lookup and extraction

Figure 5 illustrates the packet lookup procedure. First, this procedure reconstructs the packet offset by accumulating the packet sizes. For example, if the packet sizes are $P = \{100, 200, 200, 300, ..., p_n\}$, their offsets are $O = \{0, 100, 300, 500, 800, ..., o_n\}$. Then, it loads an index data related to the query. At this step, the index data is located in main memory, however it does not need to be decompressed to execute a lookup.

The query syntax is as follows,

$$\begin{aligned} Q &:= Q_{type}\ Q_{operand} \\ &:= Q_1\ and/or\ Q_2 \end{aligned}$$

A basic query consists of a query type (Q_{type}) and a query operand ($Q_{operand}$). The query type defines the kind of index data extracted from the traces, and *pcapWT* supports *pkt_num*, *eth_src*, *eth_dst*, *ipv4_src*, *ipv4_dst*, *port_src*, *port_dst*, and *l4_proto*. The query operand is the value that needs to be found on a data set of index data which is indicated by the query type. For example, the Q_{type} is *ipv4_src* and the valid $Q_{operand}$ is 10.1.1.1. The query syntax also allows to perform intersection (*and*) or union (*or*) of different queries, thus we can construct more complex queries, such as Q_1 *and* Q_2 *and ... or* Q_n.

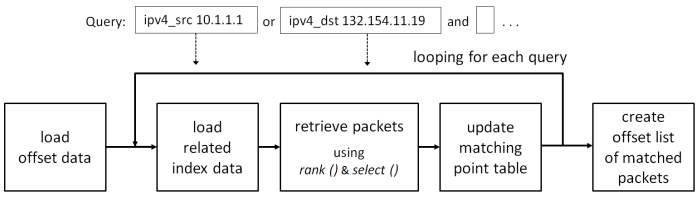

Figure 5: Searching matched packets from the index data.

More precisely, the query operand is converted to a positive integer number, using the mapping table provided from the index data build process. As mentioned above, the reason is that all index data, except offset, is mapped to positive integers. If the mapping table does not have the same entry as the query operand, this indicates that there are no matched packets for the query in the trace.

Regarding basic functions, as described in Section 3, we can get the number of matched packets (m) containing the query operand by performing

rank() of the WT. Next, we can figure out indexing numbers of matched packets by iterative executing of *select()*, for example $Y = \{2, 301, 303, ..., y_m\}$ as shown in Figure 6.

The matched packets are marked on the matching table, which is composed of a boolean array with the same number of entries as the number of packets included in the source pcap file. If there are multiple queries concatenated by *and*/*or* operators, those steps are repeated -from loading related index data- through updating the matching table. As shown in Figure 6, the matching table assigns *'1'* to its entries for the matched packets which are found by the first query or queries located after the *or* operator. On the other hand, if the packets are found by queries located after the *and* operator, the table assigns *'1'* only to entries kept *'1'*, otherwise *'0'*. Once the lookup procedure is completed, two lists are created where offsets and packet sizes for the final matched packets are marked *'1'* on the matching table.

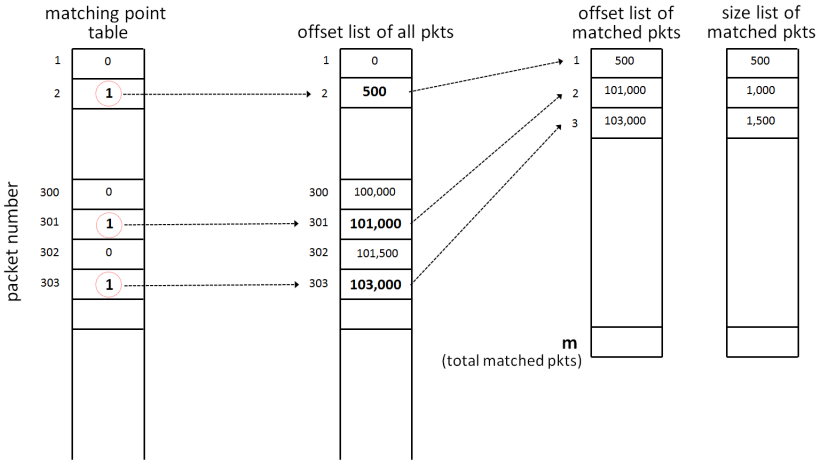

Figure 6: Listing up the final matched packets marked by '1' at the matching point table. Then the offset position and the size of selected packet are written into the offset list and the size list, respectively.

The performance of packet extraction highly depends on the capability of storage devices. In particular, this function performs random read and sequential write from/to files. The random access performance is significantly degraded in comparison with sequential access, in terms of read and write throughput. The main reason is related to the total idle time caused by mul-

tiplication of the average access delay (e.g., ranging from dozen to hundred milliseconds) and the number of random accesses. In addition, performance enhancement using cache for pre-reading consecutive data and a buffer for full padded block writing are not allowed in random I/O operations, unlike the sequential ones. However, it is possible to reduce the access delay by replacing HDD with SSD. On the other hand, file I/O libraries are also deeply related to performance degradation. Especially, in the standard C/C++ library, file seeking functions such as *lseek()* and *fseek()*, take a lot of time at each call, and generate frequent idle operations.

Therefore, we need an efficient way to compensate the performance degradation. SSDs have a very short access delay (e.g., below 1 millisecond) compared to HDDs, thus we can reduce the idle time by distributing the I/O requests into multiple processes, without significant delay overhead. Consequently, the proposed design uses multiple threads to handle multiple requests simultaneously, which are performed by OpenMP provided by gcc [26]. For more details, we provide numerical performance comparisons in Subsection 5.1.

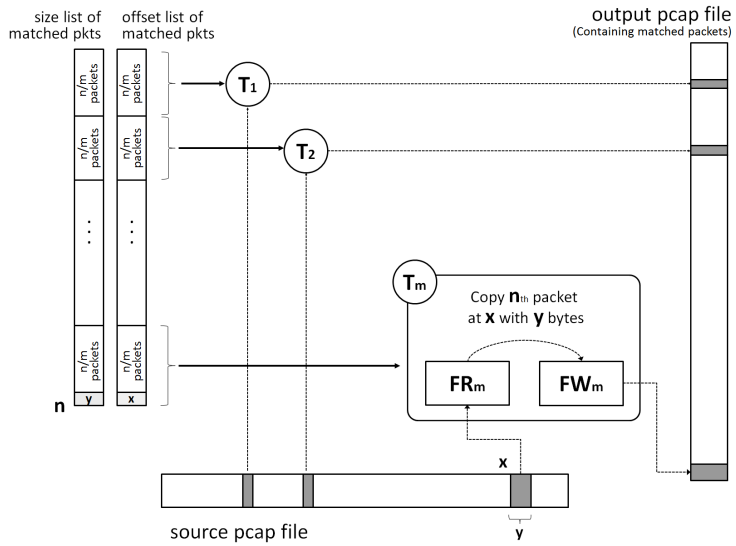

Figure 7: Retrieving the packets using multi-threaded file I/O. Multiple threads (m) evenly divide the amount of the final matched packets (n), and each thread $(T_1, T_2, ..., T_m)$ copies the assigned packets from the source to output pcap file, simultaneously.

As shown in Figure 7, a number of threads are used to balance the overall workload for the packet lookup and extraction tasks. For example, if the number of matched packets is m and the number of threads is w, each thread will retrieve m/w packets to an output file simultaneously. More precisely, each thread already knows the exact writing locations in the output file before the packet retrieval stage, by summing up the packet sizes that are assigned to each thread. Inside of each thread, the target packets are copied into user space memory from the source trace file, and then they are written into the output file.

Note that the current version of pcapWT does not support detailed filtering operations including flag fields and higher level protocols than L4, such as ACK/SYN/FIN of TCP, Type of Service (TOS) of IP, RTP/RTCP/RTSP. We expect to add this feature in a future version of the tool and believe that this will not have a high impact on the size of the indexed data. The reason of this is that these kinds of fields generally have a short range of numbers (σ), for example '0' or '1' in the ACK/SYN/FIN fields and twenty six values in the TOS fields. Consequently, σ is going to be small, and thus its wavelet tree must be very simple in terms of structure and small in terms of size.

5. Performance Evaluation

In this section, we evaluate the performance of *pcapWT*, in terms of index data size, index data building time, and packet extraction time. *PcapWT* has been developed using gcc and *libcds* [25] on the Linux platform. In particular, *pcapWT* and *libcds* have been compiled with g++ (Ver. 4.4.7) with the highest optimization flag (*-O3*). In addition, we set the *-fopenmp* option of g++ for enabling OpenMP. The machine used for benchmarking consists of CentOS 6.4 (kernel Ver. 2.6.32 for 64 bits x86 system), Intel i7-2600k processor including 8 threads (4 cores), 4 GB of main memory, a 60 GB SSD (OCZ Agility 3, low-grade product), and a 500 GB HDD (Toshiba HDD2F22).

5.1. Performance benchmark of SSD and HDD

First we benchmark read and write operations on the storage devices that will be used in the experiments. As shown in Table 1, we measure the average read or write throughput with six types of tests: sequential read/write, 4KB random read/write, and 4KB random read/write using 64 threads.

As shown in Table 1, SSD and HDD exhibit similar capabilities for the sequential write. However SSD is 3.5 times faster than HDD for the sequential read. On the contrary, random read and write performance tested on both devices is degraded, compared to the sequential read/write operations. On the other hand, multi-threaded random write obtains almost the same performance as the sequential write on the SSD. Likewise, random read performance is considerably improved when using multi-threading. However, the performance drops below 1 MBps on the HDD because of the long seeking time of the HDD, which results in inefficient multi-threading.

Basically, the packet extraction process is based on random reads and sequential writes from/to file. As we can observe in Table 1, multi-threaded read is 2.5 times faster than single threaded read on the SSD, even though the drive used in the benchmark is one of the most basic SSDs [2]. Thus, the random read capability substantially impacts performance of packet extraction on SSD. Nevertheless, other tools such as *tcpdump* and *pcapIndex* do not use multi-threading for random read.

Table 1: Sequential (Seq), random (Rand), multi-threaded by 64 threads (64T) random I/O performance measurements of the SSD and the HDD used in the experimentation.

	SSD [MBps]	HDD [MBps]
Seq Read	198.8	56.2
Seq Write	52.7	52.0
4 KB Rand Read	17.7	0.4
4 KB Rand Write	28.0	0.9
4 KB 64T Rand Read	47.0	0.8
4 KB 64T Rand Write	51.0	0.8
	[ms]	[ms]
Read Access Time	0.18	17.39
Write Access Time	0.31	4.27

[2] In the same test made on one level higher product (OCZ-Agility 3, 240 GB) than the SSD used in this evaluation, the gap is greater than 10 times (Seq Read: 211.4 MBps, 4K Rand Read: 13.6 MBps, and 4K 64T Rand Read: 140.9 MBps).

5.2. Sample packet traces

PcapWT supports *pcap* [1], one of the most popular packet trace formats. Table 2 shows four sample pcap files used in the performance evaluation, such as simulated traffic (ST) and real traffic 1/2/3 (RT-1/2/3). ST was generated by the ns-3 network simulator [27], and contains 15 million packets in a 4.2 GB pcap file. RT-1 was generated by capturing packets in our local network, and also contains 15 million packets in a 3.6 GB pcap file. ST consists of 8 endpoints at the transport layer, whereas RT-1 has 2416 endpoints. Meanwhile, those two samples are smaller than 4.2 GB because *pcapIndex* cannot support larger pcap files than the volume.

RT-2 and RT-3 are used to evaluate larger source trace files than 4.2 GB. Those two samples are generated by adding virtual packets to RT-1 in order to compare the results with RT-1 directly. Thus, RT-2 contains 115 million packets in a 13.4 GB pcap file, and the latter contains 200 million packets in a 21.6 GB pcap file.

As mentioned above, the packet extraction performance depends on the random read and write capabilities. Thus, we have manipulated those two samples in order to contain a larger amount of packets than ST and RT-1, instead of reducing their average packet sizes.

Table 2: Description of four sample pcap files: simulated traffic (ST) and real traffic 1/2/3/ (RT-1/2/3).

	ST	RT-1	RT-2	RT-3
Number of packets [million]	15	15	115	200
File size [GB]	4.2	3.6	13.4	21.6
User data size [GB]	3.9	3.4	11.5	18.4
Average packet size [Byte]	261.7	224.1	100.1	92.2
Number of end-points	8	2416	2417	2417

5.3. Other tools using packet indexing

We first analyze the performances of other tools based on packet indexing, such as Compressed Pcap Packet Indexing Program (CPPIP) [10], *PCAP-Index* [9], and *pcapIndex* [12].

These three tools use pre-built index data, but have different architectures. In particular, CPPIP adopts the bgzip data compression utility [14],

PCAP-Index runs the SQLite Ver. 3 database, and pcapIndex uses the COMPAX bitmap compression data structure [15]. We compare their performance on the SSD described in Table 1, in terms of index data size and packet extraction time.

In order to build the index data from RT-1, *pcapIndex* and CPPIP need 27 MB (0.8% [3]) and 289 MB (8.0%) of storage, respectively. However, *PCAP-Index* needs around 2.6 GB (72%), even though *PCAP-index* deals with a few simple fields (e.g., addresses, port numbers, protocol types, and so on). This is due to the database, which is not configured to use data compression. Moreover, *PCAP-Index* needs around 265 seconds to complete, whereas *pcapIndex* and CPPIP take only around 25 and 40 seconds, respectively. The reason is that *PCAP-Index* requires a lot of time to inject the index data into the database.

CPPIP and *PcapIndex* take around 20 and 30 seconds to extract the largest traffic corresponding to 48% of the volume and 36% of the total number of the packets of RT-1, respectively. However, the available operations are not the same ones used as in the prior case, since this tool supports only simple filtering queries using a range of packet timestamps and striding (e.g., selecting every n-th packets). Moreover, *PCAP-Index* takes about 80 seconds (packet lookup time from the database about 30 seconds and file writing time about 50 seconds), which is significantly slower compared to the other tools. Even *tcpdump* takes less than 32 seconds in the same use case.

CPPIP affords the advantage of being able to extract packets directly from a compressed trace file. However, some efforts are needed to support more complex filtering queries and to reduce the index data size as much as that of *pcapIndex*. *PCAP-Index* is not satisfactory from many points of views, even though it uses a database. In contrast, *pcapIndex* is the only reliable one among those tools, since it provides not only comprehensive query operations, as well as remarkable performances in terms of index data size and packet extraction time. Therefore, we select *pcapIndex* to compare performance with *pcapWT* in the following.

5.4. Building index data

PcapWT aims to minimize the index data size, for example to compress it below 1% of the volume of original trace files. As shown in Table 3, *pcap-*

[3] percentage of volume of the original trace file.

wt-index is successful for all sample files.

Table 3: A comparison of the index data size produced by *pcapIndex* and *pcapWT* with RG(50). The percentage indicates the ratio between the index data size and the volume of the original trace file.

	pcapIndex [%]	pcapWT [%]
ST	0.45	0.30
RT-1	0.78	0.86
RT-2	6.86	0.93
RT-3	Error	0.96

More precisely, as mentioned in Section 3, WT supports two bitmap representation methods (i.e., RG and RRR) and its encoding unit size that can be set by the user. Figure 8 shows that RRR is slightly better at reducing index data size than RG. On the other hand, according to Figures 9 and 10, RG is more efficient than RRR, in terms of index data building time and the

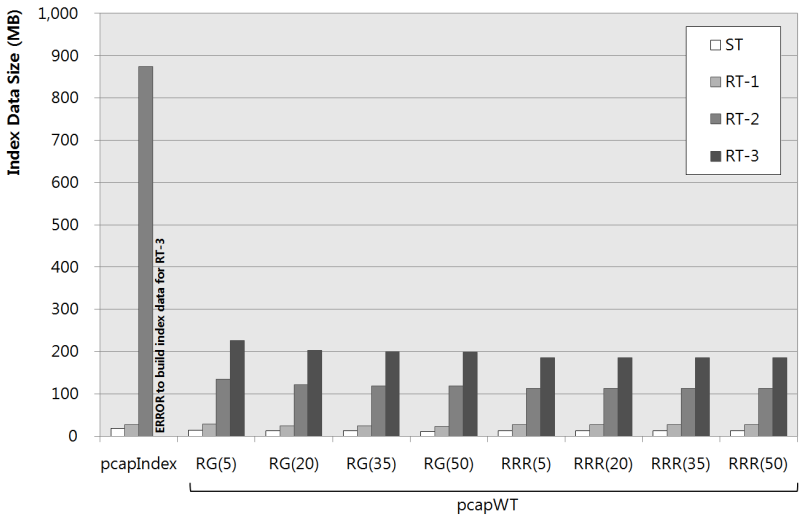

Figure 8: A comparison of the index data size for different sampling methods (RG and RRR) and sampling sizes (5, 20, 35, 50).

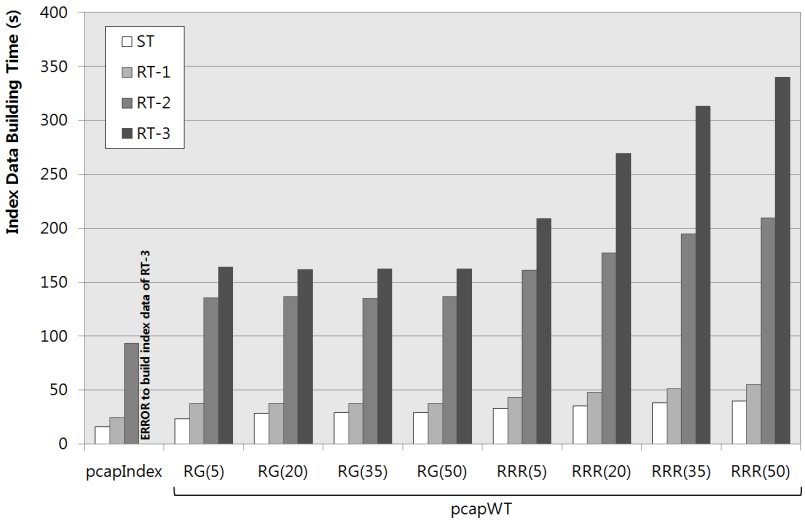

Figure 9: A comparison of the building time of index data for different sampling methods (RG and RRR) and sampling sizes (5, 20, 35, 50).

packet lookup time. Especially, as shown in Figure 8, RG using a wide sampling provides mostly complementary performance between those features. Thus, we decided to use RG(50) as the default configuration.

In addition, for improving compression efficiency and reducing search time at the same time, we use Huffman shaped WT which combines multi-layered wavelet trees and entropy compression of the bitmap sequences. Moreover, we minimize redundant bits (e.g., consecutive *'0'* bits) that can be generated in between multi-layered wavelet trees, using Huffman compression.

In contrast, *pcapIndex* obtains an abnormally large index data size and causes a segmentation fault when building the index data for trace files larger than 4.2 GB. In the case of RT-2, the index data size corresponds to 6.86% of the volume of the original trace file, and around 3.0 GB of main memory is required. Furthermore, this tool fails to build the index data for RT-3 because of lack of memory. Note that RT-2 and RT-3 have been generated by adding virtual packets to RT-1, in order to provide a direct comparison with the index data size and packet extraction performance of RT-1. More precisely, the virtual packets are equal except for the timestamp field. Thus, the index data from those samples has to be minimized regardless of the number of packets. Nevertheless, the index data size produced by *pcapIndex*

exponentially increases with the number of packets.

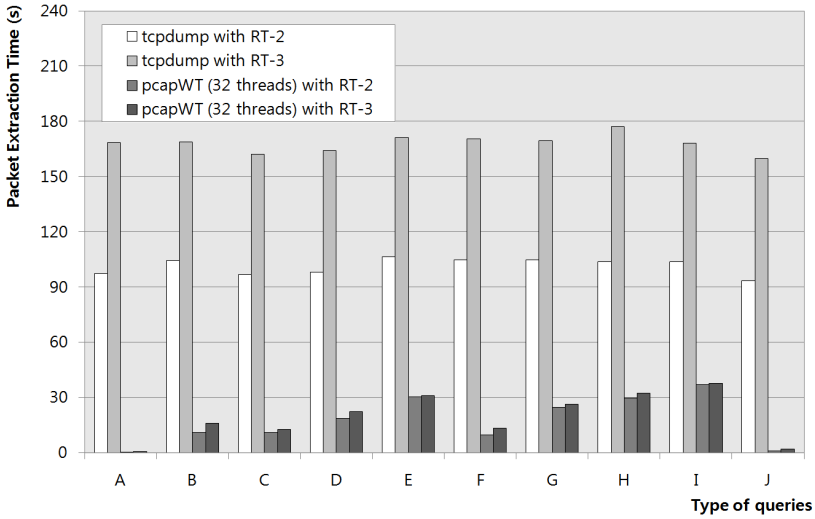

Figure 10: A comparison of the packet lookup time for different sampling sizes.

5.5. Performance evaluation of multi-threaded file I/O

The packet lookup time required to find matched items from index data takes a small portion of the overall packet extraction time. As shown in Figure 10, *pcapWT* takes less than 1 second to retrieve millions of packets. In case of ST, the number of the retrieval targets is 9.6 million packets (i.e., 63.8% in comparison to the total number of packets included in this sample.), and the volume of the target packets corresponds to 1.82 GB (i.e., 43.7% in comparison with the original volume). In case of RT-1/2/3, the amount of target packets are all same at 5.6 million packets and 0.97 GB, whereas the percentages of the number of the target packets are 37.5%, 3.8%, and 2.8%, and the percentages of the volumes of them correspond to 23.3%, 7.3%, and 4.5%, respectively. The rest of time is consumed by file I/O, including file seeking, reading, and writing. Especially, as mentioned in Section 4.2, file seeking is one of the main reasons causing performance degradation of random file read and write operations. Thus, with *pcapWT*, we propose to apply multi-threaded file I/O to increase the throughput of random file read and write.

As we can observe in Figure 11, *pcapIndex* and *pcapWT* have almost the same packet extraction performance on both the SSD and the HDD when using a single thread. In this test, *pcapWT* extracts two different sizes of traffic from ST: one low traffic containing 1.8 million packets (12.2%) corresponding to 146 MB and one high traffic containing 13.2 million packets (88.0%) corresponding to 3.0 GB. Note that the percentages indicate the ratio of the extracted packets to the total number of packets included in the sample. Meanwhile, when *pcapWT* extracts a large portion (88% of the total packets) from ST on the SSD, 32 threads can reduce the time down to 26%, compared to when using a single thread. However, multi-threading on HDD does not provide any gain. Even worse, the time is increased to 24% when using 32 threads. In this case, multi-threading generates frequent random accesses which increase delay and degrade performance on HDD.

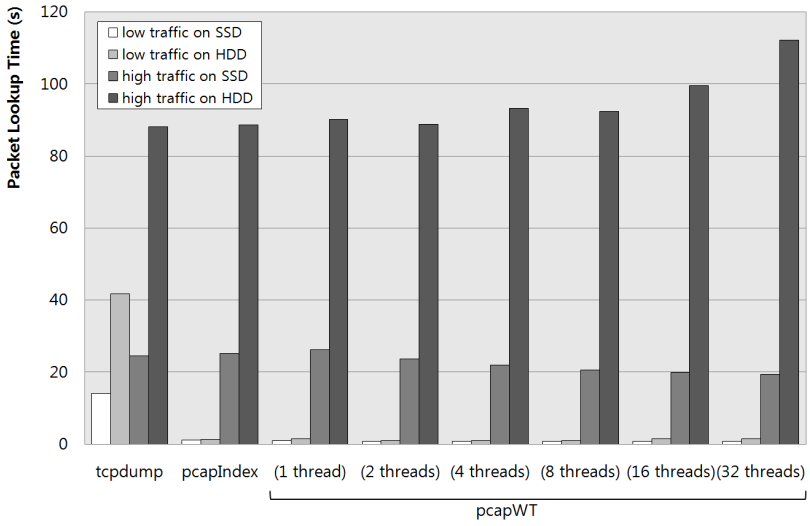

Figure 11: A performance comparison of packet extraction with various number of threads for parallel file I/O.

5.6. Performance evaluation of packet extraction

To analyze the performance of packet retrieval, we use various queries, and extract a large amount of packets. As shown in Table 4, simple queries (from A through E) extract packets specified by a pair of the query type

and its operand, and complex queries (from F through J) indicate combining simple queries by using the union operator. For example, query A extracts 4,030 packets from RT-1/2/3, corresponding to 0.03%, 0.003%, and 0.002% in comparison with the total numbers of packets, respectively. Likewise, query F (combined with A and B) extracts 525K (524,684) packets, and the total amount corresponds to 20.97%, 5.66%, and 3.49% compared to the volume of the original traces, respectively. In the same way, query I includes the former eight queries (from A through H), and contains over 12 million packets. On the other hand, query J searches packets belonging to queries B and E in common, and the extracted packets are same as the result of query B.

Table 4: Packet filtering queries and specifications of RT-1/2/3. The percentage values indicate (1) the ratio of the number of matched packets and (2) the ratio of the size of matched packets compared to the packet number and the volume of the original trace file, respectively. Note that all numbers are rounded up.

Query (IP Addr)	Matched packets $[\times 10^3]$	$[\%]^{(1)}$ (I)	(II)	(III)	Total packet size [MB]	$[\%]^{(2)}$ (I)	(II)	(III)
A	4	0.03	0.00	0.00	0.3	0.01	0.00	0.00
B	521	3.5	0.35	0.26	754	20.94	5.65	3.48
C	1,161	7.7	0.77	0.58	298	8.28	2.23	1.38
D	5,631	37.5	3.75	2.82	971	26.96	7.27	4.49
E	5,428	36.2	3.62	2.71	1731	48.07	12.97	8.00
F = $\{A \cup B\}$	524	3.5	0.35	0.26	755	20.97	5.66	3.49
G = $\{F \cup C\}$	1,686	11.2	1.12	0.84	1053	29.24	7.89	4.87
H = $\{G \cup D\}$	7,317	48.8	4.88	3.66	2024	56.21	15.16	9.35
I = $\{H \cup E\}$	12,225	81.5	8.15	6.11	3000	83.31	22.47	13.86
J = $\{B \cap E\}$	521	3.5	0.35	0.26	754	20.94	5.65	3.48

As shown in Figure 12, in experiments using RT-1, *pcapIndex* and *pcapWT* are faster than *tcpdump* in retrieving small amounts of packets, such as queries A, B, C, F, G, and J. On the contrary, the gap of packet extraction performance becomes smaller between *tcpdump* and those two tools using the packet indexing scheme. The reason is that the performance depends highly on the amount of packets extracted. Nevertheless, *pcapWT* (using 32 threads) is faster than *pcapIndex* in experiments performing complex queries, such as 21.8% at D, 20.3% at E, 20.3% at H, and 9.9% at I.

In experiments using larger samples (i.e., RT-2/3), we evaluate the performance of *pcapWT* and compare it with *tcpdump*, since *pcapIndex* cannot

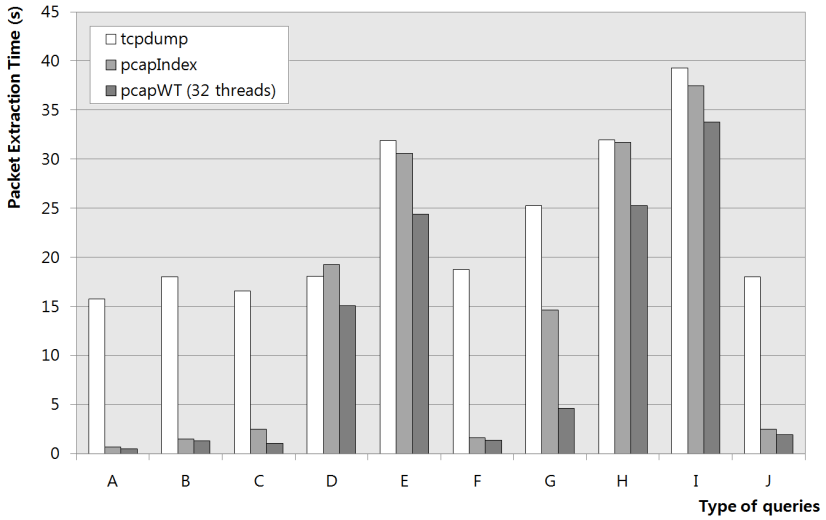

Figure 12: A performance comparison of packet extraction using RT-1.

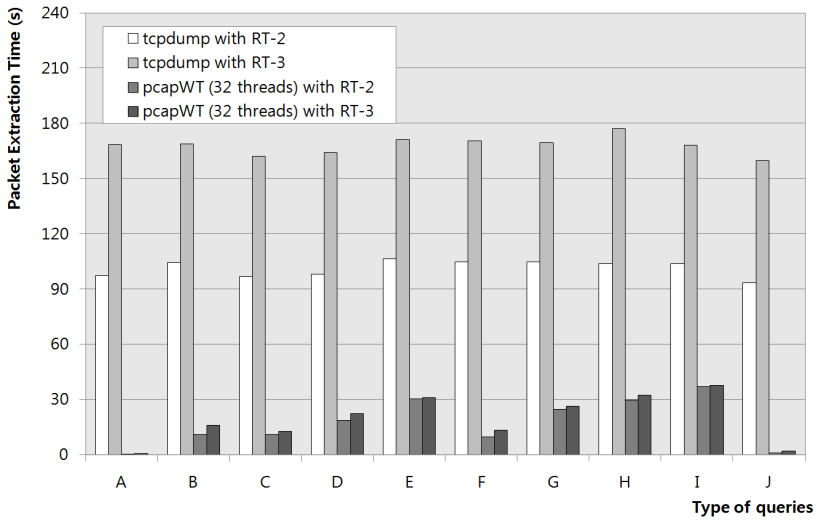

Figure 13: A performance comparison of packet extraction using RT-2/3.

support such large traces. In these experiments, *pcapWT* uses a prompt mode which supports iterative operations once the index data is loaded. Thus, this mode can avoid unnecessary time loading the same index data repetitively.

As shown in Figure 13, *tcpdump* takes much longer (i.e., around 100 seconds in the case of RT-2 and 170 seconds in the case of RT-3 along the all queries) compared to the results of RT-1, even if it extracts the same amount of packets. In other words, the performance based on the sequential approach is significantly impacted by the number of packets, regardless of the amount of packets retrieved. On the other hand, *pcapWT* using 32 threads is much faster than *tcpdump*, and the time increases proportionally to the amount of matched packets, unlike the sequential approach.

6. Conclusion

In network analysis, packet traces have always been important to analyze, since they record the complete information exchanged through networks. However, as network links, traffic complexity, and applications throughput have significantly increased, the analysis of huge amounts of those traces is becoming a challenging task, and traditional analysis tools are not efficient at dealing with such traces. Even though a few tools have been recently published for fast packet extraction, their performance is still not satisfactory in terms of index data size, and packet extraction delay.

In this paper, we propose a new tool designed to process efficiently very large network traces, which allows not only enhancing packet extraction performance on large traces, but also reducing storage requirements. *PcapWT* uses a Wavelet Tree data structure and a multi-threading scheme for parallel file I/O. Our benchmarks including SSD show that *pcapWT* is about 10 to 20% faster than *pcapIndex* in the worst case scenario and about 200% in the best cases. Moreover, this tool allows reducing the storage space required to store the index data by about 10 to 35% compared to *pcapIndex*.

Note that the current version of pcapWT does not support fine-grained filtering operations with flag fields in protocol headers. We plan to add an enhanced filtering feature in a future version of the tool and believe that this will have minor impact on the size of index data. The reason is that the added fields to account for include few bits and the corresponding wavelet tree generated is expected to be of small size and simple in terms of structure.

Acknowledgement

The authors would like to thank Dr. Renato Cerro for his great help to improve the quality of the paper. This study was supported by Anillo Project ACT-53, the Fondecyt project No. 11121475, CIRIC (INRIA-Chile) Project "Network Design", Project Semilla - UDP "ANDES" and "Anàlisis y diseño de algoritmos en redes de bajo consumo aplicado a condiciones extremas de los Andes": Programa de Cooperaciòn Cientìfica Internacional CONICYT/MINCYT 2011.

References

[1] V. Jacobson, C. Leres, S. McCanne, Tcpdump, URL `http://www.tcpdump.org/`, last visited on 18/12/2013 (2003).

[2] J. Elson, Tcpflow, URL `http://www.circlemud.org/jelson/software/tcpflow/`, last visited on 18/12/2013 (2009).

[3] P. Herman, Tcpstat tool, URL `http://www.frenchfries.net/paul/tcpstat/`, last visited on 18/12/2013 (2001).

[4] G. Combs, et al., Wireshark (2007) 12–02Last visited on 18/12/2013.

[5] L. Deri, High-speed Dynamic Packet Filtering, Journal of Network and Systems Management 15 (3) (2007) 401–415.

[6] R. Holanda, J. Verdu, J. Garcıa, M. Valero, Performance Analysis of A New Packet Trace Compressor Based on TCP Flow Clustering, Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2005) 219–225.

[7] P. I. Politopoulos, E. P. Markatos, S. Ioannidis, Evaluation of Compression of Remote Network Monitoring Data Streams, in: Proceedings of the IEEE Network Operations and Management Symposium (NOMS) Workshops, 2008, pp. 109–115.

[8] C. Estan, K. Keys, D. Moore, G. Varghese, Building A Better NetFlow 34 (4) (2004) 245–256.

[9] Pcap-index, URL `https://github.com/taterhead/PCAP-Index/`, last visited on 18/12/2013.

[10] Cppip, URL http://blogs.cisco.com/tag/pcap/, last visited on 18/12/2013.

[11] F. Fusco, X. Dimitropoulos, M. Vlachos, L. Deri, PcapIndex: An Index for Network Packet Traces with Legacy Compatibility, ACM SIGCOMM Computer Communication Review 42 (1) (2012) 47–53.

[12] PcapIndex, URL http://www.ntop.org/products/n2disk/, last visited on 18/12/2013.

[13] R. Grossi, A. Gupta, J. S. Vitter, High-order Entropy-compressed Text Indexes, in: Proceedings of the fourteenth annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.

[14] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, et al., The Sequence Alignment/Map Format and SAMtools, Bioinformatics 25 (16) (2009) 2078–2079.

[15] F. Fusco, M. P. Stoecklin, M. Vlachos, NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic, The Proceedings of the VLDB Endowment 3 (1-2) (2010) 1382–1393.

[16] I. Munro, Tables, in: Proceedings of the sixteenth Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Vol. 1180 of Lecture Notes in Computer Science, 1996, pp. 37–42.

[17] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical Implementation of Rank and Select Queries, in: Proceedings of the 4th Workshop on Efficient and Experimental Algorithms (WEA), 2005, pp. 27–38.

[18] R. Raman, V. Raman, S. R. Satti, Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets, ACM Transactions on Algorithms (TALG) 3 (4) (2007) 1–25.

[19] G. Navarro, Wavelet Trees for All, in: Proceedings of the 23rd Annual conference on Combinatorial Pattern Matching (CPM), 2012, pp. 2–26.

[20] F. Claude, G. Navarro, Practical Rank/Select Queries over Arbitrary Sequences, in: Proceedings of the fifteenth International Symposium on String Processing and Information Retrieval (SPIRE), 2008, pp. 176–187.

[21] T. Gagie, G. Navarro, S. J. Puglisi, New Algorithms on Wavelet Trees and Applications to Information Retrieval, Theoretical Computer Science 426 (2012) 25–41.

[22] Y. Liu, D. Towsley, T. Ye, J. C. Bolot, An information-theoretic approach to network monitoring and measurement, in: Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement, 2005, pp. 159–172.

[23] S. Chen, S. Ranjan, A. Nucci, IPzip: A stream-aware IP compression algorithm, in: Proceedings of the IEEE Data Compression Conference (DCC), 2008, pp. 182–191.

[24] H. Aljifri, M. Smets, A. Pons, Ip traceback using header compression, Computers & Security 22 (2) (2003) 136–151.

[25] Libcds, URL `https://github.com/fclaude/libcds/`, last visited on 18/12/2013.

[26] Openmp, URL `http://gcc.gnu.org/projects/gomp/`, last visited on 18/12/2013.

[27] Ns-3 Network Simulator, URL `http://www.nsnam.org/`, last visited on 18/12/2013.