Clemson University
TigerPrints

Publications School of Computing

5-2017

Random Access in Nondelimited Variable-length Record Collections for Parallel Reading with Hadoop

Jason Anderson
Clemson University

Christopher Gropp
Clemson University

Linh Ngo
Clemson University

Amy W. Apon
Clemson University, aapon@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/computing_pubs

Part of the Computer Sciences Commons

Recommended Citation

Please use the publisher's recommended citation.

This Conference Proceeding is brought to you for free and open access by the School of Computing at TigerPrints. It has been accepted for inclusion in Publications by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

Random Access in Nondelimited Variable-length Record Collections for Parallel Reading with Hadoop

Jason Anderson
Clemson University
jwa2@clemson.edu

Christopher Gropp
Clemson University
cgropp@clemson.edu

Linh Ngo
Clemson University
lngo@clemson.edu

Amy Apon
Clemson University
aapon@clemson.edu

***Abstract*—The industry standard Packet CAPture (PCAP) format for storing network packet traces is normally only readable in serial due to its lack of delimiters, indexing, or blocking. This presents a challenge for parallel analysis of large networks, where packet traces can be many gigabytes in size. In this work we present RAPCAP, a novel method for random access into variable-length record collections like PCAP by identifying a record boundary within a small number of bytes of the access point. Unlike related heuristic methods that can limit scalability with a nonzero probability of error, the new method offers a correctness guarantee with a well formed file and does not rely on prior knowledge of the contents. We include a practical implementation of the algorithm with an extension to the Hadoop framework, and a performance comparison to serial ingestion. Finally, we present a number of similar storage types that could utilize a modified version of RAPCAP for random access.**

I. Introduction

In the fields of computer networking administration, research, and security, the ability to capture a record of traffic at a certain point in the network for later analysis is essential. Common tasks include malware classification[1], intrusion detection[2], and discovering anomalous network activity[3]. In real world systems these *network traces* can quickly become very large; a typical 10Gb/s link can receive up to 4.5TB per hour. As 40, 56, and 100 Gb/s links become more common in commodity machines, the size of comprehensive network traces in arbitrary time windows will grow as well.

The industry standard in network trace files is the Packet CAPture (PCAP) file format. Tools such as TCPdump[4] and Wireshark[5] capture traffic on one or more network interfaces and append each packet to the tail of a PCAP file. Many applications have been written to analyze PCAP traces, including open source solutions like Snort[6] and TCPtrace[7]. More recently, Hadoop-based distributed analysis tools[8][9] have made working with very large trace files more manageable.

Due to the PCAP format's use of record pointers rather than easily distinguishable delimiters or indexing, the standard mechanism for accessing data is a sequential read from the beginning of the file. Being able to arbitrarily choose a point in the file to begin reading enables a number of useful abilities. If the records are ordered by timestamp or some field of the payload data, random access enables a binary search within the file. Tail access becomes a matter of seeking to a point suitably close to the end and reading forward. The primary use case that we envision, however, is to enable parallel ingestion of PCAP files into a distributed analytics package.

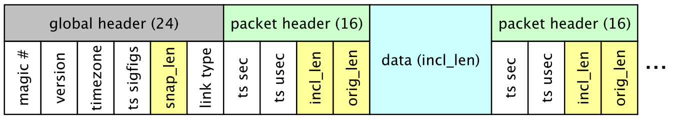

Fig. 1. The record-oriented structure of the PCAP file format. Records and headers are appended sequentially without delimiters or indexing.

The task of random access in a trace file can be challenging to do correctly. Efforts have been made to apply heuristic algorithms to identifying the first packet header within a substring of the binary trace file; however, as we will show in this work, these algorithms can be prone to incorrectly identifying the start point and misinterpreting the following data. Past efforts have also relied heavily on metadata and assumptions about the trace characteristics that are not readily identifiable from the trace file without reading it first.

In this work we present Random Access in PCAPs (RAPCAP), an algorithm that identifies record headers within a PCAP file at any given starting point. RAPCAP provides a correctness guarantee that enables scalable file reading in distributed platforms, and does not require external metadata or inspection of record contents. Aside from the algorithm, the main contributions of this paper are:

- an analysis of real world datasets to demonstrate how inaccurate heuristic identification of record boundaries limits the scalability of distributed processing;
- a comprehensive study of the boundary conditions possible within well-formed input and how RAPCAP guarantees correctness; and
- a description and performance study of a Hadoop implementation of the RAPCAP algorithm.

Finally, we outline a number of other record storage formats similar to PCAP that could employ a slightly modified version of RAPCAP for random access and parallel reading.

II. Background

1) The PCAP file format: A network trace in the PCAP format[4] contains two structural components: the *global header* and zero or more *packet headers*, as shown in Figure 1. The global header spans 24 octets at the beginning of the file, and contains fields that describe the libpcap version, byte order, time zone, timestamp accuracy, network type, and *snap_len*, the maximum number of bytes that are preserved from any

TABLE I. DATASET FEATURE COMPARISON

name	description	MB	packets	B/pkt
datacenter	500-node university datacenter [19]	1,982	19.85M	100
internet	CAIDA Internet backbone trace[13]	1,895	30.25M	63
mptcp	Multipath smartphone TCP traces [18]	280	2.85M	98
mpi	NAS Parallel Benchmark FFT in MPI	294	0.25M	1176
hadoop	Hadoop-MR on Wikipedia dataset	1,274	0.19M	6705
download	Download of a large PCAP file	1,245	0.13M	9577
p2p	Two-way multipeer Bittorrent traffic	1,416	1.39M	1019
netflix	Streaming 4k video from Netflix	1,391	0.75M	1855

captured packet. Each 16-octet packet header precedes the captured snapshot of a packet, and contains three pieces of information: the timestamp, the $incl\_len$ field that specifies how many bytes of the snapshot follow the header, and the $orig\_len$ field that records the length of the original packet.

As a network trace is captured, each packet header and snapshot are appended to the end of the file, and no previous part of the file is modified. It is an efficient method of storing variable-length data; however, since snapshots can contain any sequence of bytes and PCAP files lack indexing, it can be difficult to identify the start of a packet record.

2) Hadoop: Hadoop is a software framework that enables distributed processing of massive amount of data across a cluster of commodity computer systems[10]. The core components of Hadoop include a file management system called Hadoop Distributed File System (HDFS)[11] and the MapReduce application framework (Hadoop-MR)[12]. Large files in HDFS are divided into blocks and replicated across the cluster to provide redundancy. Hadoop-MR enables users to operate on these data blocks in parallel. The simplicity and scalability of MapReduce comes at the cost of having to follow a strict map-reduce workflow with limited knowledge and communication among executing processes. However, because workers operate on individual blocks of the data, there must be a scheme to identify a starting point for each block.

III. MOTIVATING ANALYSIS

The challenge of parsing an arbitrary substring of a PCAP file is that in addition to the real record headers in the trace, every string of 16 consecutive bytes in the file can be interpreted as a possible header. Most of these interpretations do not meet the criteria for classification as a *candidate header*. However, because record payloads can contain any byte string, there is a nonzero probability of random data and injected headers within the packet contents meeting the criteria.

Prior to this work, two methods of locating a packet header after random access have been widely used, both of which employ heuristic algorithms that rely on highly probable assumptions about the data to produce correct output. Lee et al. [15] propose using sensible differences in sequential timestamps to identify header candidates. Lukashin et al. [16] use a method that inspects packet contents for conformity to Ethernet specifications. Our concern with these methods is that misidentification of a packet header within the trace could lead to erroneous output. In this section, we assess the reliability of these heuristics with large packet traces from different sources.

A. Algorithm Characteristics

In both heuristic algorithms, the capture file is parsed byte-by-byte from the point of random access, interpreting the next 16 bytes as a record header (header A) and using the length specified by header A to identify the start of the following record (header B). A set of assertions are then applied to identify or discard the solution. For the Lee algorithm, the conditions are:

- $ts\_min < A.timestamp < ts\_max$ and $ts\_min < B.timestamp < ts\_max$, where ts_min and ts_max are input parameters indicating the extremes of the trace timestamp range,
- $B.timestamp - A.timestamp \leq ts\_delta$, where ts_delta is an input parameter indicating the longest interpacket gap in the trace, and
- $A.orig\_len - A.incl\_len > snap\_len$ and $B.orig\_len - B.incl\_len > snap\_len$, where $snap\_len$ is the maximum record length.

We note that the Lee algorithm assumes absolute packet ordering by timestamp, which is not guaranteed and is dependent on the libpcap implementation. The algorithm also depends on timestamp metadata that is not provided by the PCAP format; specifically, it requires the upper and lower bounds on timestamps in the trace, as well as the maximum gap between two consecutive records. The authors propose a gap value of 1 second, which is likely in many packet traces. However, any default value may result in falsely negative header identifications in traces containing relatively rare events.

In contrast, Lukashin et al. make no assumptions about the timestamps within the trace, and instead employ the following conditions for header identification:

- $A.incl\_len = A.orig\_len$,
- $B.incl\_len = B.orig\_len$,
- $42 \leq A.incl\_len \leq 65535$, indicating the minimum and maximum size of possible captured packets,
- $A.payload[12:13]$ is one of the 1471 valid Ethernet type codes as specified by [17] or the length of the payload l, where $l = A.orig\_len - 14$.

The Lukashin algorithm assumes that packet contents are wholly included in the trace, which is not typical in many anonymized datasets. Furthermore, relying on rigid assumptions about a packet's contents makes the algorithm difficult to adapt to other network packet types, but we assume that to be beyond the scope of their work.

Both algorithms require the $snap\_len$ parameter from the global header. In theory, the tcpdump default value of 65535 bytes could be assumed, since failure to locate a packet header may indicate an insufficient sample window. However, either algorithm could return a false positive within a sample that does not contain a real header. This may only apply to use cases such as reading a fragment of a corrupted PCAP file, which is outside the scope of record detection algorithms.

B. Datasets

To compare the accuracy of the heuristic algorithms, we chose datasets to represent a range of characteristics typical of real world network analytics problems. Table I shows the features of each set. The *mpi*, *p2p*, *hadoop*, and *download* datasets were recorded in our lab, while the other datasets are

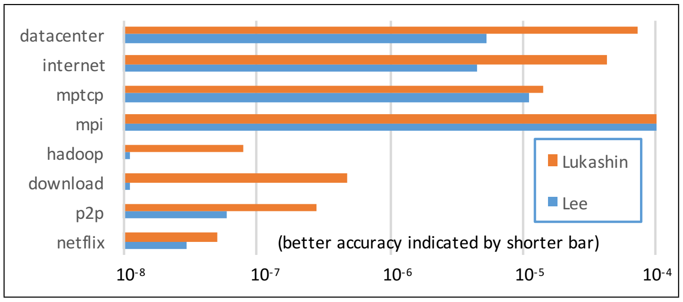

Fig. 2. With heuristic algorithms, the probability of false positive classification of a randomly chosen string of bytes within a dataset as a valid PCAP record is non-zero. Note the log_{10} scale.

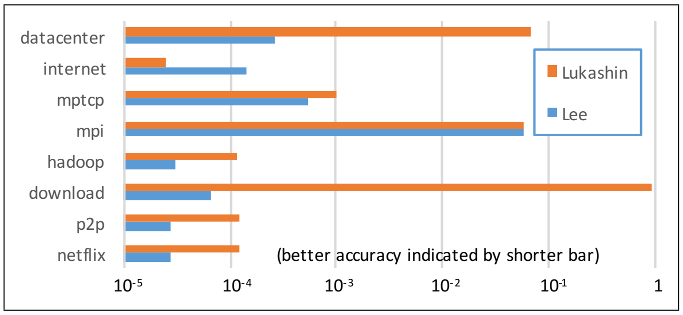

Fig. 3. As consecutive byte strings within a trace file sample are tested, the compounding likelihood of false positive identification of the next record header grows. The probabilities shown use the mean record length of each dataset. Note the log_{10} scale.

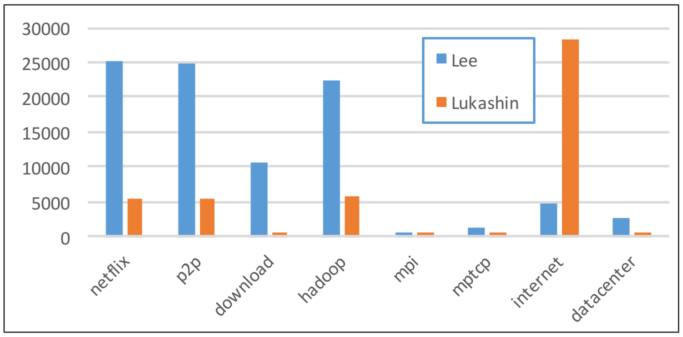

Fig. 4. For parallel file ingestion, the nonzero failure probability of heuristic algorithms limits scalability. Shown is the number of file partitions at which 50% probability of misidentification of a record header is achieved.

from the cited sources. All datasets use the tcpdump default $snap\_len$ of 65535 bytes. However, three of the datasets – *mptcp*, *datacenter*, and *internet* – protect the privacy of the traffic source by truncating the payloads of the recorded packets, while retaining the original length in the $orig\_len$ field of the record. This required us to modify the Lukashin algorithm to allow $incl\_len \leq snap\_len$ for these datasets.

C. Accuracy Evaluation

To assess the quality of results obtained from heuristic random access algorithms, we tested the correctness of header identification on each dataset. For these experiments, we used each algorithm's heuristic criteria to classify every possible 16-octet string within the file as a candidate header. The results formed a confusion matrix, from which we used the number of false positives and size of the dataset to compute a probability for misclassification. Figure 2 shows that the probability of misclassification for both Lukashin and Lee is generally low, but not zero. Notably, the Lukashin algorithm was less accurate with the *download* trace due to the presence of false record headers inside record payloads, which can be seen in Figures 2 and 3.

Using these data along with the mean length of each record within the datasets, we were able to compute a probability for successful identification of the next record header after a random access into the trace. For a mean record length k and false positive probability p, header identification requires an average of $k/2$ trials as sequences of bytes are tested, $(1-p)^{\frac{k}{2}}$. The complement of this, i.e. the probability of false positive header identification, is visualized in Figure 3. While the accuracy is exponentially dependent on the value of k, the order of magnitude differences in values of p between the datasets is also an influential factor.

D. Scalability

Our primary interest is how this failure probability affects parallel reading of a large trace file. As the number of data partitions and associated parallel tasks increases, the probability that one of the tasks falsely identifies the start point increases as well. A false header identification will present the parser with malformed input, and may render the entire job a failure. In Hadoop especially, the mapping nodes lack a way to communicate that the computed file offsets are in error. In the worst case, the job could complete successfully but with erroneous output. We measure this risk as $log_p(0.5)$, i.e. the number of random accesses required to reach 50% probability that one of them is in error, and show the results in Figure 4. For some datasets, such as *mpi* and *mptcp*, the probability that a job completes successfully is less than 50% when data is partitioned into just a few hundred segments.

IV. Reliable Record Detection

The headers within a PCAP can be thought of as nodes in a directed acyclic graph, with the file index as a node's label, and a function of a node's label and the record's *incl_len* forming a single directed edge to another node. In a well formed trace with n records, there exists an unbroken path of length n from the node labeled "0" and the node representing the final record. We refer to this path as the *true solution*, and any correct reading must start from one of its nodes.

In Figure 5, we show how record headers map to a digraph. Nodes can be identified by parsing the byte string inside a read window. This window is sized to guarantee inclusion of at least one member of the true solution, based on the *snap_len* of the file. Because of the possible inclusion of misidentified candidate headers, there are three types of solutions that can arise in addition to the true solution:

- A set of headers that are disjoint from the true solution is a *false solution*, exemplified by nodes 96 and 193 in Figure 5.
- A false solution that cannot be eliminated before the end of the input is a *parallel solution*.
- A set of headers that lead to a node of the true solution is a *tree solution*. In Figure 5, nodes 61 and 218 eventually point to the true solution node 240.

Algorithm 1 Find first header in PCAP split

```
 1: procedure ISVALID($h, snap\_len$)
 2:     if $h.orig\_len > 0$ and
 3:         $h.incl\_len = $ MIN$(h.orig\_len, snap\_len)$ then
 4:         return true
 5:     end if
 6:     return false
 7: end procedure
 8: procedure FINDSTART($file, snap\_len$)
 9:     $solutions \leftarrow$ MINHEAP()
10:     $packet\_len \leftarrow snap\_len + \text{HLEN}$
11:     $chunk \leftarrow file.$READ$(packet\_len + \text{HLEN} - 1)$
12:     for all $i$ in $(0..packet\_len - 1)$ do
13:         $h \leftarrow$ HEADER$(chunk[i, i + \text{HLEN}))$
14:         if ISVALID$(h, snap\_len)$ then
15:             $next\_i \leftarrow i + h.incl\_len + \text{HLEN}$
16:             $s \leftarrow$ SOLUTION$(i, next\_i)$
17:             $solutions.$PUSH$(s)$
18:         end if
19:     end for
20:     if $|solutions| = 0$ then
21:         return no solution
22:     end if
23:     $offset \leftarrow 0$
24:     while $|chunk| \geq \text{HLEN}$ do
25:         $max\_i \leftarrow offset + |chunk| - \text{HLEN}$
26:         while $solutions.$PEEK$().next\_i \leq max\_i$ do
27:             $s \leftarrow solutions.$POP$()$
28:             while $|solutions| > 0$ and
29:                 $solutions.$PEEK$().next\_i = s.next\_i$ do
30:                 $s.last\_i \leftarrow s.next\_i$
31:                 $solutions.$POP$()$
32:             end while
33:             $header\_i \leftarrow s.next\_i - offset$
34:             $h \leftarrow$ HEADER$(chunk[header\_i, \text{HLEN}))$
35:             if ISVALID$(h, snap\_len)$ then
36:                 if $|solutions| = 0$ then
37:                     return $s.last\_i$
38:                 end if
39:                 $s.next\_i \leftarrow s.last\_i + h.incl\_len + \text{HLEN}$
40:                 $solutions.$PUSH$(s)$
41:             end if
42:         end while
43:         $offset \leftarrow offset + packet\_len$
44:         $carry \leftarrow chunk[packet\_len, |chunk|)$
45:         $chunk \leftarrow carry \star file.$READ$(packet\_len)$
46:     end while
47:     return parallel solutions
48: end procedure
```

Tree solutions present an additional challenge; it can be difficult to tell which parent of a candidate is part of the true solution. This can be seen in Figure 5: nodes 164 and 218 both point to node 240, but without information outside the scope of the parser, it cannot be known which is correct. Furthermore, it is also possible that the real parent of a true solution candidate is outside the read window. We refer to these potential parents as *ambiguous*. In a read window that only guarantees inclusion of one node from the true solution, all nodes except the highest indexed true solution candidate are ambiguous.

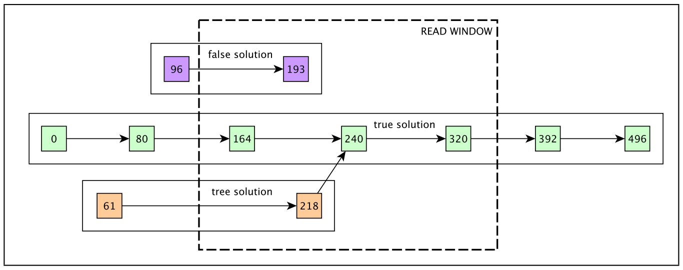

Fig. 5. Candidate record headers within a PCAP string, where the *incl_len* field identifies the starting byte of the next record.

V. The RAPCAP Algorithm

In order to find the correct starting point within a substring of a PCAP file, we must eliminate all false and ambiguous nodes from a comprehensive set of candidates. In this section, we describe the RAPCAP algorithm and the logic used to ensure that the identified node is an unambiguous member of the true solution. The general form is as follows:

1) Sample enough bytes of the input to guarantee the inclusion of a complete header;
2) Parse the sample for valid headers to build a set of candidate solutions;
3) Validate the header at each solution's next index while within the sample window;
4) Read further into the file to validate more candidates;
5) Repeat #3 and #4 until reduced to a single solution.

When only one solution remains, we can be sure that the last validated header index is part of the true solution. If we are unable to narrow the solution set to a single item, then we have *parallel solutions*, an error condition that must be handled differently depending on the implementation. If step 2 finds no valid headers, it is an error condition that indicates a malformed PCAP file. Note that this algorithm is not intended to detect malformation, and assumes well-formed input.

We define a string of 16 bytes as *valid* packet header if it meets two criteria, which are based on the definition of the PCAP file format:

1) $orig\_len > 0$
2) $incl\_len = min(orig\_len, snap\_len)$

Assertion 1 is not strictly necessary, but simply eliminates many false candidate solutions in real world traces. Assertion 2 limits the value of $incl\_len$ to one of two possible scenarios: either the entire original packet was captured, or the packet was truncated to a length equal to $snap\_len$. Algorithm 1, lines 1-7 describe this procedure. Note that we make no assumptions based on the packet contents or about relationships between headers that are not explicitly defined by the format.

In the RAPCAP algorithm, it is unnecessary to store the entire tree, as the only file index that needs to be reported is an unambiguous member of the true solution. Candidate solutions are stored as a tuple containing two fields: $next\_i$, the next index referred to by this solution; and $last\_i$, the last validated header index.

A. Phase 1: Initial Candidates

In a well formed trace, no packet data may be longer than the file's $snap\_len$. We refer to the number of bytes spanned by a maximum-sized packet and its header as $packet\_len$. We assert that a sample of length $packet\_len + \text{HLEN} - 1$ must include at least one complete header from the file's true solution. Therefore, parsing the sample for all valid headers must yield a nonempty set of potential solutions, of which one or more are members of the true solution. In Algorithm 1, lines 9-19, we create the initial set of candidate solutions. These candidates form the leaves of solution trees, which will be pruned and converged in Phase 2.

In a solution tuple, the $next\_i$ field is used as the comparator, so that by always popping the minimum value, we test solution candidates in order of file index.

B. Phase 2: Iterative Solution Pruning

Narrowing the candidate solutions is a straightforward process; we iteratively test each solution until it refers to an index with an invalid header. To accomplish this, we use two nested loops.

The outer loop (lines 23-25 and 43-46) progressively reads samples of size $packet\_len$ from the file into $chunk$, while *offset* tracks the position relative to the starting point. The last header index that can be tested as a whole header is 16 bytes from the end of the sample, so with each iteration, the remaining 15 bytes are prepended to the next sample to continue testing seamlessly (lines 44-45). The loop stops if there is insufficient remaining input to parse as a header.

The inner loop (lines 26-42) continues until the next solution index is beyond the current sample window, at which point another sample is needed. The solution with the minimum $next\_i$ is popped from the solution set, and then all other solutions with the same $next\_i$ are popped to coalesce the children in a tree solution to a single parent node (lines 27-31). The data at that index is interpreted as a header and validated (lines 32-34). If it is valid, one of two cases must be true. If it is the only solution remaining, then the index that was just validated must be unambiguous and part of the true solution, so it is returned as the correct starting point. If more solutions remain, then this solution is updated to point at the next candidate header, and pushed back onto the minheap.

VI. Implementation

In Hadoop, the HDFS file system splits large files across blocks, which have a configurable default of 128 or 256 MiB. These blocks are replicated across compute nodes for redundancy and data locality. In the MapReduce framework, the blocks are abstracted as InputSplits with logical boundaries, which may not align with block boundaries. The RecordReader class adjusts the InputSplit boundaries to suit the data type. An example of this is the Hadoop native LineRecordReader, which supports text processing by adjusting InputSplit boundaries to coincide with the closest newline character.

One method of adjusting the InputSplit boundaries is at the time of instantiating a RecordReader object, where the Hadoop task is responsible for determining its own InputSplit boundaries. For example, LineRecordReader adjusts the InputSplit start index forward to the first character after the first newline in the block, and the end index is advanced to include the first newline character of the next block. In most cases, finding the start index involves reading from the local disk, while the length is found by reading data from another HDFS block that may not be local to the task.

We employ a similar method for local split identification using RAPCAP. The PcapRecordReader object is tasked with identifying the boundary indices, which are the first unambiguous headers of the task's InputSplit and the next InputSplit. Since map tasks do not typically communicate with each other, it is necessary for each boundary to be computed twice, similar to LineRecordReader. However, because the maximum length of PCAP records is typically no more than 64KB, and we can identify the correct header within 1-3 reads with high probability, the amount of data processed twice is typically small. Modern Hadoop clusters use HDFS blocks of 128MB or 256MB, and between 0.05% to 0.30% of data is read to determine split boundaries. One thing to note is that the value of $snap\_len$ must be remotely read by each PcapRecordReader object, or communicated to the map tasks in some other way such as the distributed cache.

An alternative to requiring each PcapRecordReader to determine the boundaries of its own InputSplit is to globally compute all of the boundaries before the job starts by overriding the getSplits() method provided by FileInputFormat. Advantages of this method include only needing to compute each boundary once rather than twice, and reading the input file header one time rather than once for every HDFS block of the input file. However, because of the high overhead of small reads between HDFS datanodes, serializing this process can take significant time to complete before the job can start.

A. Methodology

We assessed the performance of parallel ingestion with Hadoop using a 16-node cluster, with each machine having dual 16-core Intel Xeon E5-2650 CPUs, 256GB RAM, 10Gb Ethernet, and 12x1 TB hard drives. The data processed was a full version of the CAIDA Internet backbone trace[13], which is one 126.5GB file with 2.03 billion packets, which occupies 970 HDFS blocks. We used Hadoop 2.7.1, and data was distributed with a replication factor of 3. Our implementation modifies a fork of the RIPE-NCC packet processing library[8]. The source code is available for use[14] and has been offered to the upstream project for inclusion in future versions.

B. Performance

In our speedup assessment, we performed a simple packet count using both the default non-splitting PcapRecordReader and the RAPCAP-modified version to determine split boundaries. To fairly account for the more distributed reduce operation, the entire run length of the job was measured. On the test cluster, the mean parallel runtime was 78.8 seconds, a speedup of 5.1x over the serial runtime.

Compared to heuristic algorithms, the theoretical performance of detecting a header boundary with RAPCAP can be relatively low. Both the Lee and Lukashin algorithms finish determining a header in $O(n)$ time with a single sample of

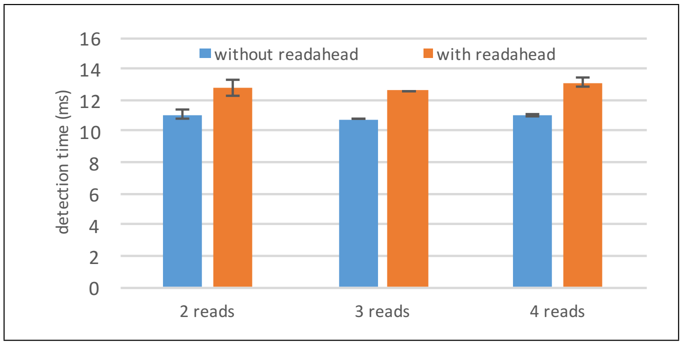

Fig. 6. 128KiB readahead cache strategy having a negative effect on consecutive remote read performance.

snap_len bytes, while RAPCAP is bounded at $O(n^2)$ operations on as many samples as are required to eliminate false solutions. However, in our testing with real world traces, the mean number of samples required to eliminate false solutions was 2.05 with a maximum of 4. In our Hadoop packet counting implementation, header detection was an insignificant portion of job runtime (0.00004% of a 130GB job) with no measurable difference between jobs using RAPCAP and the heuristic algorithms.

From the smaller sample dataset, we observed that many of the boundary detections required multiple $snap\_len$-sized samples to be read from the disk. Since these reads frequently involve an HDFS block stored remotely to the map task, we hypothesized that prefetching the data in anticipation of multiple reads from the input stream would result in lower detection times. In HDFS, a FSDataInputStream allows setting a caching strategy on the underlying DFSInputStream. The default value is 4 KiB, which we increased to 128KiB, or prefetching enough for two reads of the dataset's 64 KiB $snap\_len$. However, the higher readahead value resulted in performance degradation, as shown in Figure 6.

VII. Conclusion

In this work we have described and demonstrated RAPCAP, an algorithm for accurate detection of record boundaries in the industry standard PCAP network trace format. Our work is motivated by observations of existing heuristic algorithms, where nonzero probabilities of inaccurate classification of a record boundary can impact results correctness and limit scalability. Boundary detection enables random access into record collections of this type, enabling binary search, tail seeking, and parallel ingestion with frameworks such as Hadoop.

PCAP is only one example of a nondelimited and nonindexed variable-length record storage format that can benefit from a random access algorithm for parallel reading. Some prominent examples include the Lempel-Ziv-Oberhumer (LZO) compression algorithm[20] that is widely used in Hadoop deployments, the industry standard FASTQ format[21] for storing nucleotide sequence data, and the LAS format[22] used to store point measurements from LIDAR scans. Each of these formats require serial reading, and are prime candidates for minor modifications to RAPCAP to enable random access and parallel reading. It is our hope that future researchers can employ our findings to advance computational efficiency in these and other fields of study.

Acknowledgment

This research was supported by NSF grants #1642542 and #1405767. Clemson University is acknowledged for a generous allotment of compute time and storage on the Cypress Cluster. We thank Dr. Jim Martin for providing network traces of streaming video data. We also thank the anonymous reviewers for their insightful comments, which improved the quality of this paper.

References

[1] S. Nari and A. A. Ghorbani, "Automated malware classification based on network behavior," in *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pp. 642–647, IEEE, 2013.

[2] C.-Y. Ho, Y.-C. Lai, I.-W. Chen, F.-Y. Wang, and W.-H. Tai, "Statistical analysis of false positives and false negatives from real traffic with intrusion detection/prevention systems," *IEEE Communications Magazine*, vol. 50, no. 3, pp. 146–154, 2012.

[3] R. Grossman, M. Sabala, Y. Gu, A. Anand, M. Handley, R. Sulo, and L. Wilkinson, "Discovering emergent behavior from network packet data: Lessons from the angle project," *Next Generation of Data Mining*, pp. 243–260, 2009.

[4] "Tcpdump and libpcap repository." http://www.tcpdump.org.

[5] "Wireshark." http://www.wireshark.org.

[6] "Snort." http://www.snort.org.

[7] "Tcptrace." http://www.tcptrace.org.

[8] "RIPE-NCC Hadoop PCAP library." http://github.com/RIPE-NCC/hadoop-pcap.

[9] Y. Lee and Y. Lee, "Toward scalable internet traffic measurement and analysis with Hadoop," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 5–13, 2013.

[10] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[11] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, IEEE, 2010.

[12] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[13] "The CAIDA anonymized 2016 Internet traces, 01-21-2016." Center for Applied Internet Data Analysis. http://www.caida.org/data/passive/passive_2016dataset.xml, 2016.

[14] "RAPCAP source code and supplemental data." https://www.cs.clemson.edu/dice/pcap/.

[15] Y. Lee, W. Kang, and Y. Lee, "A Hadoop-based packet trace processing tool," in *International Workshop on Traffic Monitoring and Analysis*, pp. 51–63, Springer, 2011.

[16] A. Lukashin, L. Laboshin, V. Zaborovsky, and V. Mulukha, "Distributed packet trace processing method for information security analysis," in *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pp. 535–543, Springer, 2014.

[17] "IEEE 802 numbers." http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml, 2014.

[18] Q. D. Coninck, M. Baerts, B. Hesmans, and O. Bonaventure, "CRAWDAD dataset uclouvain/mptcp_smartphone (v. 2016-03-04)." Downloaded from http://crawdad.org/uclouvain/mptcp_smartphone/20160304/mptcp_smartphone, Mar. 2016. traceset: mptcp_smartphone.

[19] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, ACM, 2010.

[20] "The LZO compression libraray." http://www.oberhumer.com/opensource/lzo/.

[21] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2010.

[22] A. Samberg, "An implementation of the ASPRS LAS standard," in *ISPRS Workshop on Laser Scanning and SilviLaser*, pp. 363–372, 2007.