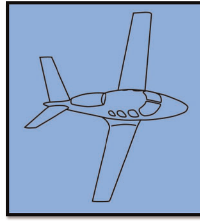
Available online at http://docs.lib.purdue.edu/jate



JATE

Journal of Aviation Technology and Engineering 6:1 (2016) 16–24

Can Flight Data Recorder Memory Be Stored on the Cloud?

Yair Wiseman

(Bar-Ilan University)

Abstract

Flight data recorders (FDRs, or black boxes) generate data that is collected on an embedded memory device. A well-known difficulty with these devices is that the embedded memory device runs out of space. To avoid getting into this problematic situation, the software of the FDR is designed to operate in a watchful mode, constantly working to minimize the use of memory space; otherwise a larger FDR would be needed. However, larger FDRs can be a problem because they have very rigorous requirements; thus, enlargement is costly. Outcomes of this research include the recommendation to send FDR data to a remote cloud storage system, so the data memory device will be unbounded.

Keywords: IaaS, flight data recorder, data compression

Introduction

The information produced by flight data recorders (FDRs, or black boxes) is by and large gathered on an embedded memory device within the FDR itself; however, this embedded memory device faces capacity issues. In order to avoid this problematic circumstance, the FDR has to attentively manage memory. Manufacturing larger memories for FDRs is very difficult because they have challenging physical constraints, making any potential enlargement a high cost. Thus, FDRs are designed to continually minimize memory space utilization, in order to avoid the need for enlargement. Improving upon this method for efficiency, our implementation makes the memory of the FDR effectively unlimited by transmitting the information produced by the FDR to a remote cloud storage system. Essentially, investment in remote FDRs can be classified as a preventive investment. With growing costs related to air disasters, investment in improving their prevention should increase over time to balance the cost vs. investment in the optimal manner. See, for example, Giat, 2013.

1. Motivation

An FDR is a replaceable computer element used in airplanes. Its task is to record pilots' inputs, electronic inputs, sensor positions, and information sent to any electronic systems on the airplane (You, Ye, & Yang, 2014). An FDR is informally referred to as a "black box." FDRs are designed to be quite small and are carefully manufactured to withstand the impact of both high speed and extreme heat (Endre & Winterhalter, 2012).

http://dx.doi.org/10.7771/2159-6670.1119

Today, passengers in airplanes can use both the Internet and cell phones (Ehssan & Jamalipour, 2006); the ability to wirelessly connect the network is mainly based upon two techniques:

1. Cellular-based network that employs many cell phone towers over ground. This method is obviously unsuitable over bodies of water. The towers have been fabricated to direct their signals at the sky rather than along the ground. The airplanes catch the information using a receiver installed on their undersides. When the information arrives at an airplane, it will be distributed throughout the cabin by the use of a conventional Wi-Fi system.
2. Satellite Internet access provided through communications satellites. Similar to the cellular-based network, when the information arrives at an airplane it will be distributed throughout the cabin by the use of a conventional Wi-Fi system.

We suggest using these communication techniques to send FDR information to an IaaS cloud. This method eliminates the need for the FDR's embedded memory device to handle memory storage, and transfers that burden to the cloud. However, the transmitted data might be exceedingly large; therefore, compression might be needed prior to transmission.

Furthermore, as was noted by Kavi (2010) and Purcell and colleagues (2011), after a crash the FDR cannot always be found. For example, Air-France Flight 447 crashed into the Atlantic Ocean on June 1st, 2009. In that accident, 228 people were killed. Since the FDR was not recovered from the ocean floor until May 2011, nearly two years later, it was hard to determine the causes of the crash. Additionally, no immediate action was taken to remedy the failures.

Another case is the disappearance of Antonov An-72 on December 22, 1997. This airplane went missing on a flight from Port Bouet Airport, Côte d'Ivoire to Rundu Airport, Namibia, but never arrived at Rundu Airport. Five crew members and the airplane vanished over the southern Atlantic Ocean, but since the FDR has not been found, the reason for the disappearance remains unknown (Ranter & Lujan, 2010).

One more notable disappearance is Malaysia Airlines Flight 370, which disappeared on March 8, 2014 while flying from Kuala Lumpur International Airport in Malaysia to Beijing Capital International Airport in China, carrying 12 crew members and 227 passengers (McNutt, 2014). It was published that the battery for the FDR of this airplane expired in December 2012 (Hawley, 2015). Regardless, the FDR itself has not been found, so the cause of the incident remains undetermined.

These cases illustrate how FDR memory stored in a cloud system can be a practical answer for this obstruction. Up-to-date high density FLASH memory devices have facilitated the SSFDR (solid-state flight data recorder) to be manufactured with a substantially larger memory size. Many airplanes are now equipped with SSFDRs and no longer make use of disk drives (Wiseman & Barkai, 2013). Additionally, in the past twenty-five years, the density of memory chips has significantly multiplied, and the capability to record thousands of parameters for hundreds of flight hours in FDRs or quick assess recorders is now possible. So the old question "Can flight data recorders hold enough data for an international flight?" is no longer relevant. The new question is "How many parameters, videos, and other data can be added to the memory of the flight data recorder?"

FAA regulations stipulate that FDRs retain just the last two hours of recorded information (FAA, 2011). Thus, many of the recorders have capacity to store only 2 hours of data. Unlike the debate in the personal computers arena, in the embedded computing arena—particularly with regard to FDRs—there is a concensus: the memory space size is too small. Actually, the memory storage space in FDRs is less than one percent of storage space available on a conventional desktop computer. Typically, in an unexceptional embedded computer system there is an electronic card with a plain processor supporting a small solid-state device that has just about 1–4GB of memory space for all of the system files. Usually it is impossible to insert additional memory space such as a hard disk drive or even a Secure Digital (SD) reader because of hardware constraints, system constraints, size constraints, and power consumption constraints (Yaghmour, Masters, Gerum, & Ben-Yossef, 2008).

Therefore, it is understandable why we cannot install a full operation system environment, which includes a compilation chain (tool chain), in such a small memory space. For instance, basic installation of a Gentoo Linux distribution with a command line user interface, a stage-3 compilation tool chain, and its Portage package manager, without any graphical interface or other packages, requires 1.5 GB. Installation of the Windows operating system takes up much more memory space.

The easiest solution is removing features, installing only the essentials, and developing lighter applications for the embedded cards of FDRs. Compression algorithms are employed by the manufacturers of FDRs and may turn out to be even more relevant with the introduction of video FDRs. Although video FDRs are quite old (Armstrong, 1989), the latest video compression techniques have a considerable compression ratio, which is commonly more than some hundreds (i.e. the compressed file will be much less than 1% of the original data; Horowitz et al., 2012). This explains why the compression concern has resurfaced, even though the memory capacity is much larger now (Yang, Dick, Lekatsas, & Chakradhar, 2005; Xu, Clarke, & Jones, 2004).

A common difficulty is when FDRs run out of memory space (Weisberg & Wiseman, 2013). Because the designers

of the FDR are concerned about this lack of memory, they design the FDRs' memory management process to make a constant effort to reduce the used memory space (Wu, Banachowski, & Brandt, 2005). Unlike FDRs, in many other embedded computer systems, more often than not disks are not overloaded; therefore, usually it is better to keep old versions of important files on the disks even though in most cases the old versions are not used (Muniswamy-Reddy, Wright, Himmer, & Zadok, 2004).

Enabling Data Transmission From an FDR to a Cloud Storage System

With the aim of transferring an adequate amount of data from an FDR to the cloud, compression is required. For such purposes, as with the computational codes that run across today's FDRs, compression techniques cannot be used arbitrarily. Their use must be dynamically configured to match current requirements (i.e., desired transmission rates and current platform resources, or network bandwidth and CPU load).

The system presented in this paper enables the FDR to automatically configure the compression technique to fit the current requirements. When enough network bandwidth is available, for instance, no compression will be applied, thereby the computational loads will be reduced; however, when network bandwidth is not enough for the transmitted data, the transmitted data will be compressed. The particular compression technique will be automatically selected, using dynamic data sampling techniques to assess the effectiveness and current rapidity of compression.

The objective of this paper is to guarantee that the rate of compression speed due to available CPU resources and the compression efficiency will create suitable data volumes transmitted over the network at rates that match current available network resources as well as application requirements. Given the datasets, the trade-offs in compression rapidity vs. reductions in required network bandwidth will be calculated. A configurable process to select a suitable compression technique based on the calculation has been developed. The technique selection process takes into account compression rapidity, current machine load, efficiency of the compression techniques, type of data, and available network bandwidth. Because platform resources change, technique selection is performed frequently, throughout the lifetime of data transmitted by the FDR.

3. Compression Methods

Compression techniques reduce data size by applying compression and decompression techniques to data. This section succinctly reviews the techniques employed in this work, in order to show the trade-offs in using these different techniques.

Huffman Compression

Huffman coding (Huffman, 1952) has been the first practical compression method. Huffman coding is usually not used in a stand-alone mode (Dandekar, 2013); rather, it is used within more complex compression techniques like JPEG (Wallace, 1991; Wiseman, 2014). The concept of Huffman coding is assigning a shorter codeword to a common item and a longer codeword to an uncommon item. The following algorithm shown in the recursive pseudo code below describes how Huffman chooses these codewords with minimum average size for items $A1, \ldots, An$ of lengths $L1, \ldots, Ln$, where $P1, \ldots, Pn$ are the items' probabilities.

If ($n = 2$), then
  {0, 1}
Else
  Combine the two smallest probabilities Pn, $Pn - 1$
  Solve for $P1, P2, \ldots, Pn - 2, Pn - 1 + Pn$
If $Pn - 1 + Pn$ is represented by x, then
  $Pn - 1$ will be represented by $x0$
  Pn will be represented by $x1$

"Solve for" means calling the function again with $n - 1$ elements because $Pn - 1$ and Pn become one element, as indicated by $Pn - 1 + Pn$.

The main advantages of Huffman codes are their simplicity and rapidity. These codes work well for binary data when string repetition is rare. Huffman assumes that each character has no relation to the adjacent one; therefore Huffman usually does not perform well on texts. Huffman's complexity is $O(m + n\log n)$ where m is the size of the text and n is the size of the alphabet. It should be noted that this is much better than the Burrows-Wheeler compression technique described below, which has a complexity of $O(m\log m)$.

Lempel-Ziv Methods

The conventional dictionary compression technique is Lempel-Ziv coding (Wiseman, 2007b). WINZIP (WinZip, 1998) and gzip (Deutsch, 1996), among other common practical compression tools, employ versions of Lempel-Ziv coding. Whereas Huffman coding does not consider an item's surroundings, the main advantage of Lempel-Ziv methods is that they consider previous appearances of strings. The concept of the algorithm is described herein:

Let $x1, \ldots, xn$ be a sequence of items.

We want to find a sub-sequence $xk, \ldots, xm$ which holds *Eqn* (3):

$$P(x_k,\ldots,x_m) > \prod_{i=k}^{m} P(x_i) \quad (3)$$

For example, $p(qu) > p(q) \cdot p(u)$.

The Lempel-Ziv scheme puts a pointer into the place of each previewed string. We use a version of Lempel-Ziv that

compresses these pointers by Huffman coding (Peterson, 2013). The pointers of Lempel-Ziv look like (345, 8), which means go backward 345 bytes and copy 8 characters. Most pointers point to close data, and a copy of only a small number of bytes is done, so both of the numbers have a tendency to be small. These pairs of numbers are then replaced by Huffman codewords, which give shorter representation for small numbers.

The Burrows-Wheeler Transformation

The Burrows-Wheeler transformation (Burrows & Wheeler, 1994) is a dictionary compression technique. This technique utilizes repetitions of words' sequences in order to improve compression. The technique is lossless (i.e., no information is lost in the compression procedure). Burrows-Wheeler transformation outperforms Lempel-Ziv coding; therefore, the use of Burrows-Wheeler transformation in a variety of compression utilities is widespread. However, the execution time of Burrows-Wheeler transformation is normally very long.

The technique has several steps: The first step produces pointers to all characters of the data being compressed. The pointers are sorted according to the characters to which they are pointing. The preceding characters of each of the pointers are conveyed to the next step according to the order of the sorted pointers. Essentially, this sequence of characters in the output of this step has the same characters as in the original data, but the order of the characters is different.

The second step performs a "move to front" algorithm. This algorithm keeps all 256 potential characters in a list. When a character is sent to the next step, its position in the list is sent in its place. After a replacement of a character is sent, it is moved from its current position in the list to the front of the list. The next step applies a run-length coding to the output of the previous step. The output of the run-length coding is compressed usually by Arithmetic coding (Howard & Vitter, 1994; Wiseman, 2001); however, Huffman coding can also be applied.

The main disadvantage of the Burrows-Wheeler transformation is its slow execution time, because of the need to sort the data. In order to reduce this execution time, usually the data is split into blocks, at some loss in compression efficiency, because shorter data is less effectively compressed. This paper uses the SGI version of the Burrows-Wheeler Transform (Wiseman, 2007a).

In order to enable us to decompress the data when the order of blocks received does not exactly correspond to the order in which it is sent, we have adapted the Burrows-Wheeler method, as explained here: Each data is split into blocks of a number of bytes. The Burrows-Wheeler Transform compresses each block. Then, blocks are processed by the move to front procedure, followed by run-length coding. The run-length coding has been changed to use a run-length of at most 254 characters, so that the 255th character never appears. Instead, the 255th character is placed at the end of each compressed block. Next, all of the blocks are compressed together using Huffman coding. Huffman can be synchronized easily (Klein & Wiseman, 2000; 2003). This indicates that if a Huffman-encoded data is read from any arbitrary point, it possibly will have a few erroneous bytes in the beginning, but the rest of the characters will be correct. So, the compressed data can be decoded from any arbitrary point, since Huffman will keep track of character positions, and when position 255 is observed, a new block has been detected.

Method Comparison

The attributes of these compression techniques have been evaluated; accordingly, the system will be able to choose the most appropriate compression technique for any given attribute of the data, the available communication bandwidth, and available CPU cycles.

Table 1 qualitatively ranks compression techniques, scaled on four levels:

- Excellent
- Good
- Satisfactory
- Poor

Given these technique evaluations, the following selection algorithm chooses the compression technique most suitable for the current execution environment.

In this algorithm, we use the term "reducing speed" to capture the speed at which a certain technique can compress data, given currently available CPU cycles. This speed is checked repeatedly, as subsequent blocks of data are compressed. In addition, the speed with which compressed

Table 1.
Attributes of compression techniques.

	Burrows-Wheeler	Lempel-Ziv	Huffman
Compress Files With String Repetitions	Excellent	Excellent	Poor
Compress Files With Low Entropy	Excellent	Poor	Excellent
Compression Efficiency	Excellent	Good	Poor
Time of Compression	Poor	Satisfactory	Excellent
Time of Decompression	Satisfactory	Excellent	Excellent
Global Time	Poor	Good	Excellent

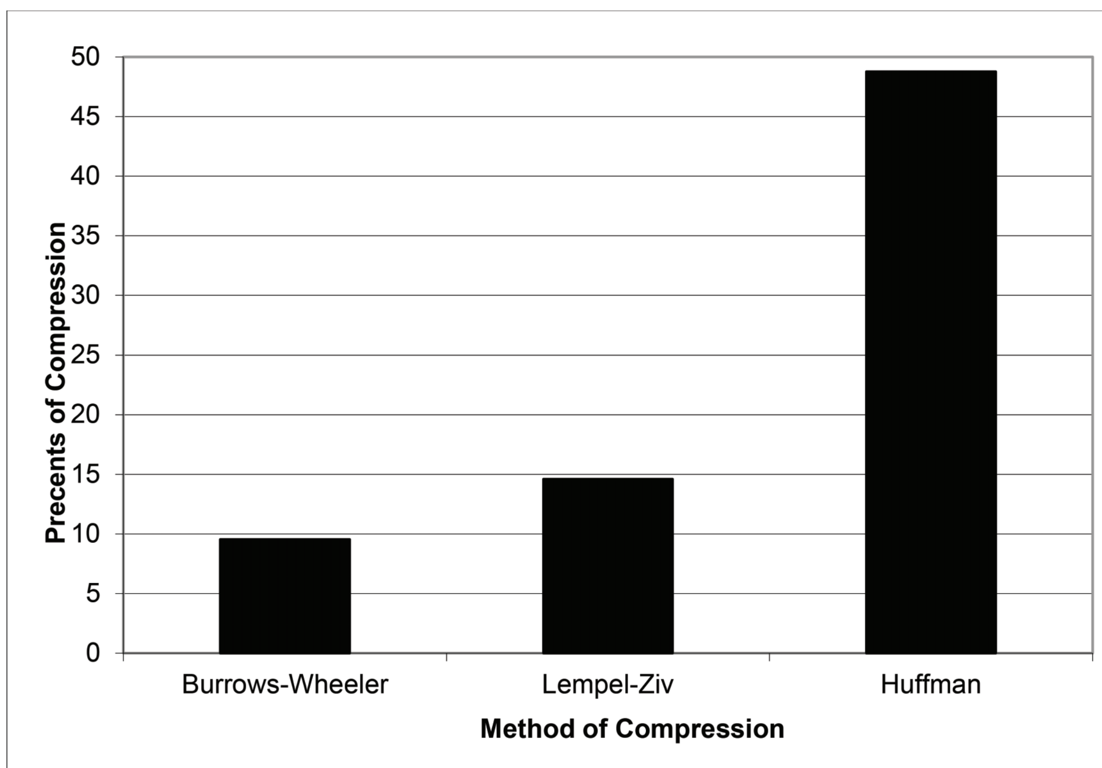


Figure 1. Compression ratios.

blocks are accepted by receivers is repeatedly checked, thereby analyzing both current network bandwidth and receiver speed. These end-to-end numbers are more relevant than knowledge of actual network bandwidth, because decompression requires the use of receivers' CPU cycles.

The sizes of the blocks have been preferred based on the common page size (Weisberg & Wiseman, 2009; Itshak & Wiseman, 2009) and the efficiency of compression techniques derived from Klein & Wiseman (2005). The ratios between the sending time and the reducing speed size have been set according to the statistics detailed in Figure 3. The efficiency of the sampling has been set according to the numbers in Figure 1. Obviously, this information is specific to the particular data; however, these numbers can be easily tuned if needed by sampling even a small piece of data (Wiseman, Schwan, & Widener, 2005) extracted from the original data and sending this piece of data over an unloaded line employing unloaded CPUs. It should be noted that usually the numbers being used are very close to the constants that are detailed here, so we use these constants to give an impression of the scope of numbers.

Assume the reducing size speed of the first block is infinity.

```
While not EOF
Take a block of 128 kB.
  If (sending time) > 0.83*(the reducing size speed of Lempel-Ziv)
    If sampling has been compressed into less than 48.78%
      If (sending time) > 3.48*(the reducing size speed of Lempel-Ziv)
        Use Burrows-Wheeler
      Else
        Use Lempel-Ziv
    Else
      Use Huffman
  Else
    Do not compress
  Fork a sampling process to compress the first page (4 kB) of the next block by Lempel-Ziv and use its output to decide on the reducing speed size and the compression ratio for the next 128 kB block.
  Send the block.
  Wait for the child process.
```

Experimental Results

FDRs generate various information; therefore, the compression techniques used in this paper (i.e., Huffman, Lempel-Ziv, and Burrows-Wheeler) have been tested with multiple datasets, including a text dataset and a binary dataset. The text dataset includes these parameters:

(1) Time;
(2) Altitude;
(3) Airspeed;
(4) Vertical acceleration;
(5) Heading;

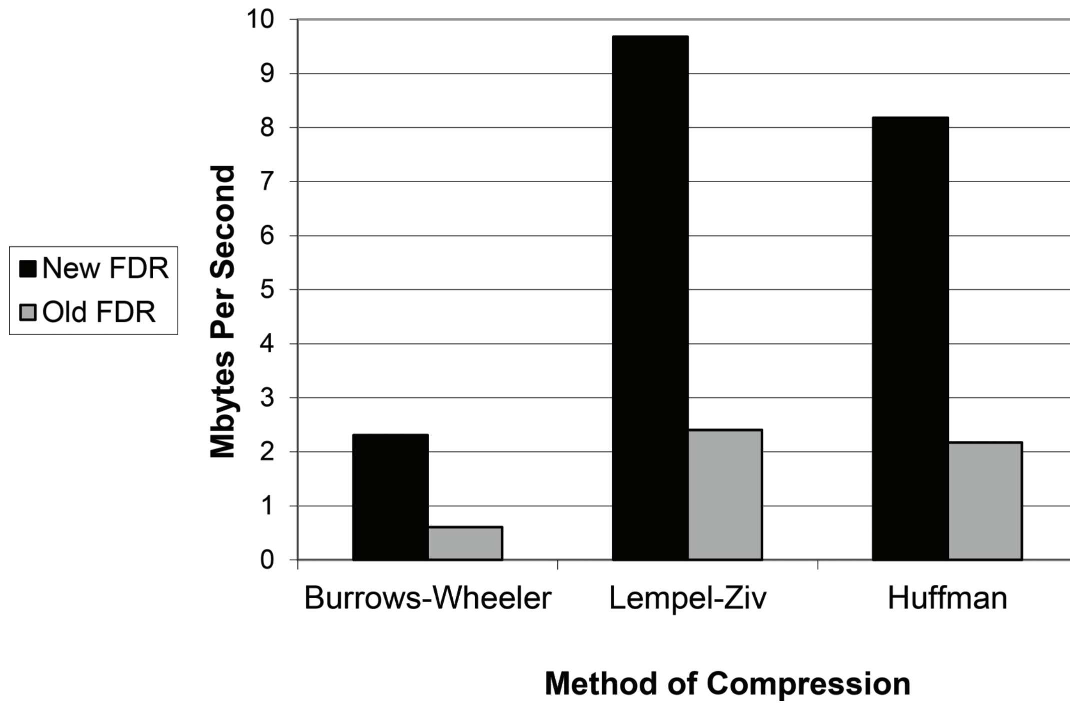


Figure 2. Reducing size speed.

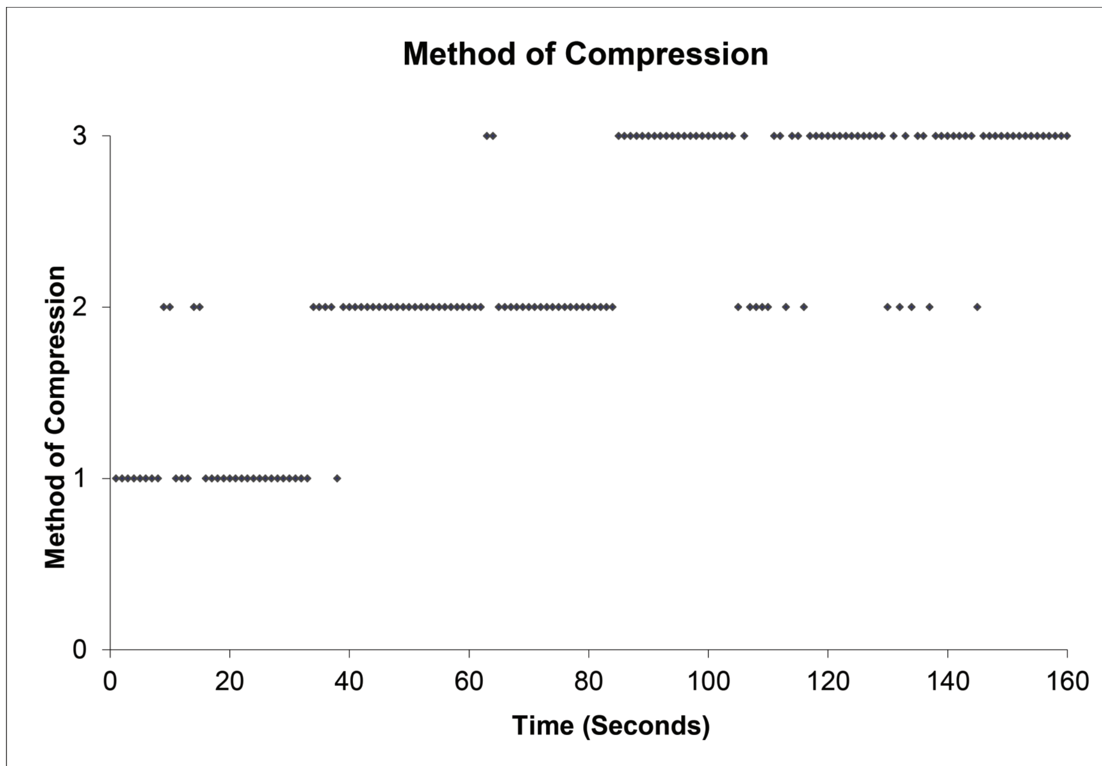


Figure 3. Switching of compression techniques.

(6) Time of each radio transmission either to or from air traffic control;
(7) Pitch attitude;
(8) Roll attitude;
(9) Longitudinal acceleration;
(10) Pitch trim position;
(11) Control column or pitch control surface position;
(12) Control wheel or lateral control surface position;

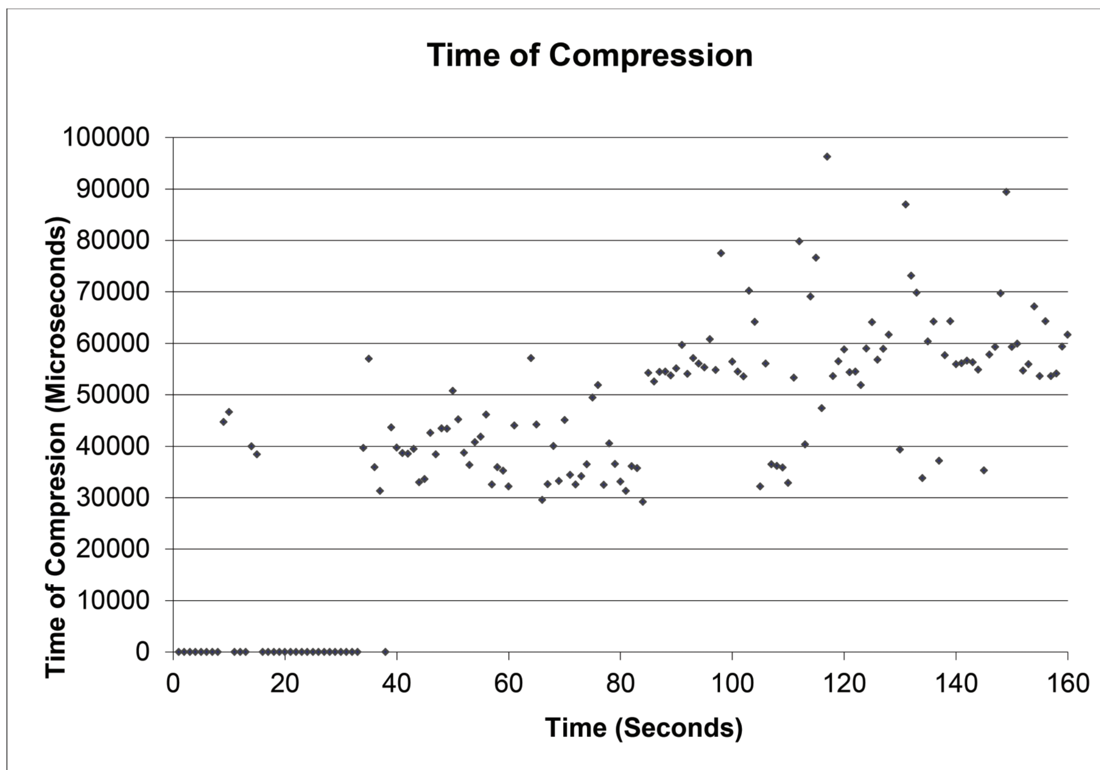


Figure 4. CPU time compression.

(13) Rudder pedal or yaw control surface position;
(14) Thrust of each engine;
(15) Position of each thrust reverser;
(16) Trailing edge flap or cockpit flap control position; and
(17) Leading edge flap or cockpit flap control position.

The binary dataset is the cockpit voice data. Actually, the cockpit voice data is now usually compressed by mp3 (Kawakita, 2001), which is a version of Huffman, so in this attribute we did not contribute further compression.

For the text dataset, the compression ratios are shown in Figure 1.

Decisions about suitable compression techniques should be based not only on data sizes or link speeds, but also on data characteristics. Huffman codes are suitable for low entropy data, while Lempel-Ziv methods are good at handling data with string repetitions. Burrows-Wheeler handles both of these cases.

The consequent approach taken in our work is one that samples data as it is being produced and transported, to detect whether data has low entropy, string repetitions, or both. The results of such sampling are used to choose a suitable compression method.

The crucial statistics from these experiments is the speed with which a CPU compresses some large amount of data. Figure 2 summarizes the test results accomplished with two FDRs. Because of confidential commercial issues, we cannot detail the names of the FDRs and we will refer to them simply as "New FDR" and "Old FDR."

Figure 2 shows what is called in this paper the "reducing size speed" of different FDR processors, which is the ability of an FDR's processor to reduce the amount of bytes per second. If such a memory space reduction can be performed faster than the transfer time for a given amount of data, it is worth it (time-wise) to compress the data.

If an FDR processor is fast but the communication line is slow, even a multipart compression method can be used. Conversely, if the CPU is slow and the communication line is fast, no compression technique will be assigned. Between those two extremes, the use of a fast and uncomplicated compression method will be used.

Usually cell phone and Internet lines have a large standard deviation of the transfer speed. This brings about different compression technique selections at different points of time.

The conclusion from the above figures is that if the CPU is very fast, then Burrows-Wheeler will be the best technique. Burrows-Wheeler has a poor ratio of reducing MBits per second, but if the CPU is fast enough, the CPU will be able to pay back any lost time. If the communication line is very fast, Huffman will be the best technique. Huffman has a poor compression ratio but compresses data quite fast. When an intermediate case occurs, Lempel-Ziv seems to be a good compromise between compression time and transfer time.

Figure 3 illustrates the switching of compression techniques over time. The data being compressed, transported, and decompressed is a set of information captured from a

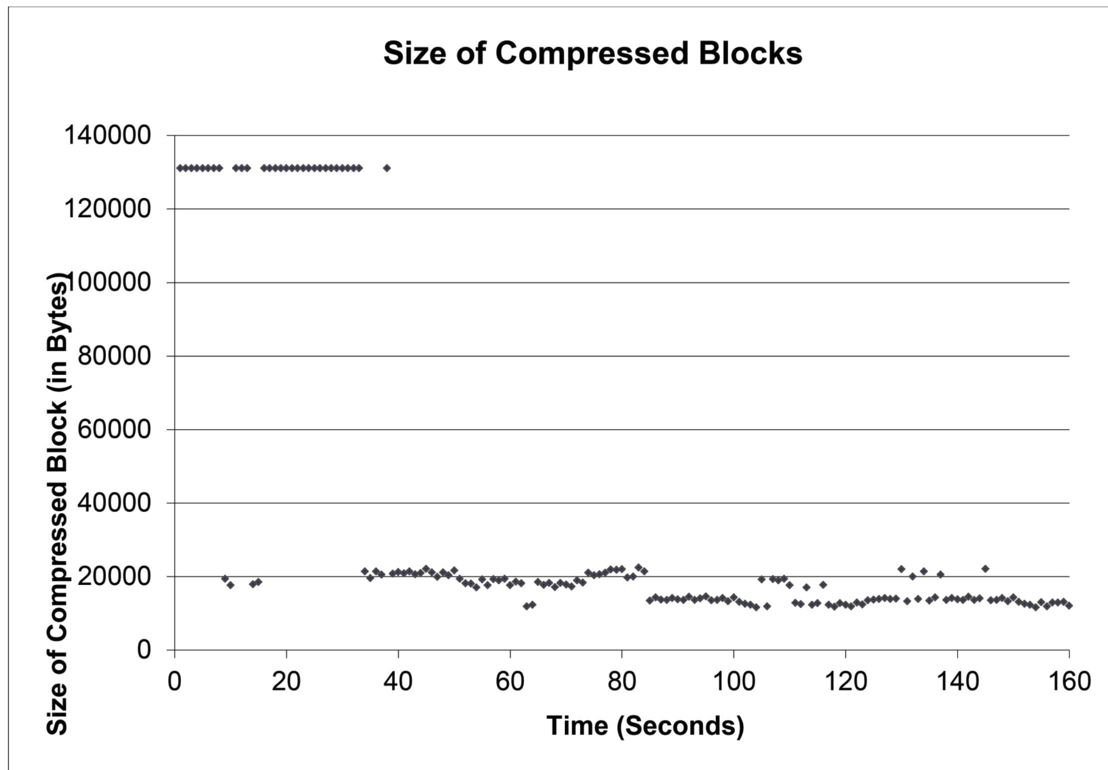


Figure 5. Transmitted compressed blocks' size.

large company. This dataset has a high rate of data repetitions, so the best techniques used were Lempel-Ziv and Burrows-Wheeler. This analysis explains the automatic decision-making depicted in Figure 3. Initially, with no network load, no compression is performed (labeled as "1" in the figure). With increasing network load, the first compression method used is Lempel-Ziv (see "2" in the figure), followed by Burrows-Wheeler (see "3") under high network loads.

Figure 3 depicts the selection of compression technique over time, whereas Figures 4 and 5 show the compression times and the sizes of the compressed blocks, respectively, achieved by these techniques. These figures clearly show that the relatively small improvement in data reduction achieved by the use of Burrows-Wheeler justifies its usage only under very high network loads.

Conclusions

Several benefits have been achieved by placing the FDR in an IaaS cloud:

(1) Safer FDR memory;
(2) Extensive enlargement of the FDR memory; and
(3) Faster transmission of data.

These results are encouraging. The transmission system is able to select the compression algorithm intensity and whether to compress or not at real time based on available CPU cycles and the load on the network line, so the cloud can be a replacement for the internal memory device. Additionally, clouds are crash-proof. The possibility of a traditional FDR being damaged in a crash is low, because FDRs are designed to sustain even very severe crashes; however if an airplane does crash, potential damage to the cloud is absolutely zero.

References

Armstrong, H. B. (1989). Improving aviation accident research through the use of video. *ACM SIGCHI Bulletin*, *21*(2), 54–56.

Burrows, M., & Wheeler, D. (1994). Block sorting lossless data compression algorithm. System research center, research report 124. Digital System Research Center, Palo Alto, CA.

Dandekar, O. (2013). Full-band CQI feedback by Huffman compression in 3GPP LTE systems. *International Journal of Computer Applications*, *77*(9), 37–42.

Deutsch, L. P. (1996). GZIP file format specification version 4.3. Retrieved from http://tools.ietf.org/html/rfc1952

Ehssan, S., & Jamalipour, A. (2006). The global in-flight internet. *IEEE Journal on Selected Areas in Communications*, *24*(9), 1748–1757.

Endre, B., & Winterhalter, M. (2012). Crash survivable memory unit. U.S. Patent 8, 121, 752.

Federal Aviation Administration (FAA; 2011). Cockpit voice recorders, Sec. 121.359, paragraphs (i)(2) and (j)(2).

Giat, Y. (2013). The effects of output growth on preventive investment policy. *American Journal of Operations Research*, *3*, 474–486.

Hawley, S. (2015). Malaysia Airlines MH370: Report finds battery powering locator beacon on black box expired in 2012, no red flags raised over crew or aircraft. ABC, Australia. Retrieved from http://www.abc.net.au/news/2015-03-08/mh370-report-says-black-box-locator-beacon-expired/6289462

Horowitz, M., Kossentini, F., Mahdi, N., Xu, S., Guermazi, H., Tmar, H., Li, B., Sullivan, G. J., & Xu, J. (2012). Informal subjective quality comparison of video compression performance of the HEVC and H. 264/MPEG-4 AVC standards for low-delay applications., In Proceedings of the *Applications of Digital Image Processing XXXV*, San Diego, CA, p. 84990W.

Howard, P. G., & Vitter, J. S. (1994). Arithmetic coding for data compression. Proceedings of the *IEEE*, *82*(6), 857–865.

Huffman, D. (1952). A method for the Construction of Minimum Redundancy Codes. Proceedings of the *IRE*, *40*, 1098–1101.

Itshak, M., & Wiseman, Y. (2009). AMSQM: Adaptive multiple superpage queue management. Special issue of The International Journal of Information and Decision Sciences (IJIDS) on the best papers of IEEE Conference on Information Reuse and Integration (IEEE IRI-2008), 1(3), 323–341.

Kawakita, K. (2001). Crash prevention recorder (CPR)/video-flight data recorder (V-FDR)/cockpit-cabin voice recorder for light aircraft with an add-on option for large commercial jets. U.S. Patent Application 09/999, 589.

Kavi, K. M. (2010). Beyond the black box. *IEEE Spectrum 47*(8), 46–51.

Klein, S. T., & Wiseman, Y. (2000). Parallel Huffman decoding. Proceedings of the *Data Compression Conference* DCC-2000, Snowbird, UT, pp. 383–392.

Klein, S. T., & Wiseman, Y. (2003). Parallel Huffman decoding with applications to JPEG files. *The Computer Journal*, *46*(5), 487–497. Oxford University Press: Swindon, UK.

Klein, S. T., & Wiseman, Y. (2005). Parallel Lempel Ziv coding. *Journal of Discrete Applied Mathematics*, *146*(2), 180–191.

McNutt, M. (2014). The hunt for MH370. *Science*, *344*(6187), 947.

Muniswamy-Reddy, K., Wright, C. P., Himmer, A., & Zadok, E. (2004). A versatile and user-oriented versioning file system. In Proceedings of the *Third USENIX Conference on File and Storage Technologies* (FAST 2004). San Francisco, CA, 115–128.

Peterson, P. A. H. (2013). Datacomp: Locally-independent adaptive compression for real-world systems (Doctoral dissertation). UCLA Electronic Theses and Dissertations.

Purcell, M., Gallo, D., Packard, G., Dennett, M., Rothenbeck, M., Sherrell, A., & Pascaud, S. (2011). Use of REMUS 6000 AUVs in the search for the Air France Flight 447. In Proceedings of *IEEE OCEANS* 2011, pp. 1–7.

Ranter, H., & Lujan, F. I. (2010). ASN Aircraft accident Antonov 72 ER-ACF between Abidjan and Rundu. *Aviation Safety Network*. Retrieved from https://aviation-safety.net/database/record.php?id=19971222-1.

Wallace, G. K. (1991). The JPEG still picture compression standard. *Communication of the ACM*, *34*, 3–44.

Weisberg, P., & Wiseman. Y. (2013). Efficient memory control for avionics and embedded systems. *International Journal of Embedded Systems*, *5*(4), 225–238.

Weisberg, P., & Wiseman, Y. (2009). Using 4KB page size for virtual memory is obsolete. Proceedings of the *IEEE Conference on Information Reuse and Integration* (IEEE IRI-2009). Las Vegas, NV, pp. 262–265.

WinZip. (1998). Nico Mak Computing, Inc. Mansfield, CT, USA.

Wiseman, Y. (2001). A pipeline chip for quasi arithmetic coding. *IEICE Journal—Trans. Fundamentals, E84–A*(4), 1034–1041.

Wiseman, Y. (2007a). Burrows-Wheeler based JPEG. *Data Science Journal*, *6*, 19–27.

Wiseman, Y. (2007b). The relative efficiency of LZW and LZSS. *Data Science Journal*, *6*, 1–6.

Wiseman, Y. (2014). The still image lossy compression standard—JPEG. In, *Encyclopedia of Information and Science Technology* (3rd ed.; chapter 28). HersheyPA: IGI Global.

Wiseman, Y., & Barkai, A. (2013). Smaller flight data recorders. *Journal of Aviation Technology and Engineering*, *2*(2), 45–55.

Wiseman, Y., Schwan, K., & Widener, P. (2005). Efficient end to end data exchange using configurable compression. *Operating Systems Review*, *39*(3), 4–23.

Wu, J. C., Banachowski, S., & Brandt, S. A. (2005). Hierarchical disk sharing for multimedia systems. In Proceedings of the *International Workshop on Network and Operating Systems Support for Digital Audio and Video*. New York, NY, pp. 189–194.

Xu, X. H., Clarke, C. T., & Jones, S. R. (2004). High performance code compression architecture for the embedded ARM/Thumb processor. In Proceedings of the *Conference on Computing* Frontiers. Ischia, Italy, pp. 451–456.

Yaghmour, K., Masters, J., Gerum, P., & Ben-Yossef, G. (2008). *Building embedded linux systems*. O'Reilly Media, Inc.

Yang, L., Dick, R. P., Lekatsas, H., & Chakradhar, S. (2005). CRAMES: Compressed RAM for embedded systems. In Proceedings of the *International Conference on Hardware/Software Codesign and System Synthesis*. Jersey City, NJ, pp. 93–98.

You, L. C., Ye, G. R., & Yang, Q. F. (2014). Design of flight data signal generator system. In, *Applied Mechanics and Materials*, *556*, 5143–5147.

Dr. Yair Wiseman teaches and advises MSc and PhD students at Bar-Ilan University, Israel Aircraft Industries, and Holon Institute of Technology. He has an MSc and PhD from Bar-Ilan University, and Albert Einstein is his academic great-great-grandfather (i.e., the advisor of the advisor of the advisor of Dr. Wiseman's advisor). Dr. Wiseman completed two postdocs: at the School of Computer Engineering at the Hebrew University in Jerusalem, and the Georgia Tech Center for Experimental Research in Computer Systems. He has collaborated with other partners and received research grants to run an active laboratory from inter alia Sun Microsystems, Intel, Polak Foundation, and The Open University; and he has evaluated several large projects of the European Union, Israel Science Foundation, MB Logic, and more. He has been a committee member for dozens of conferences around the world and judged numerous academic articles, and he currently serves on the editorial boards of numerous academic journals, including the World Review of Intermodal Transportation Research, Journal of Traffic and Transportation Engineering, American Journal of Vehicle Design, and more.