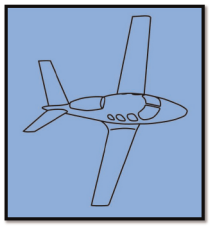
Available online at http://docs.lib.purdue.edu/jate



JATE

Journal of Aviation Technology and Engineering 2:2 (2013) 45–55

Smaller Flight Data Recorders†

Yair Wiseman and Alon Barkai

Bar-Ilan University

Abstract

Data captured by flight data recorders are generally stored on the system's embedded hard disk. A common problem is the lack of storage space on the disk. This lack of space for data storage leads to either a constant effort to reduce the space used by data, or to increased costs due to acquisition of additional space, which is not always possible. File compression can solve the problem, but carries with it the potential drawback of the increased overhead required when writing the data to the disk, putting an excessive load on the system and degrading system performance. The author suggests the use of an efficient compressed file system that both compresses data in real time and ensures that there will be minimal impact on the performance of other tasks.

Keywords: flight data recorder, data compression, file system

Introduction

A flight data recorder is a small line-replaceable computer unit employed in aircraft. Its function is recording pilots' inputs, electronic inputs, sensor positions and instructions sent to any electronic systems on the aircraft. It is unofficially referred to as a "black box". Flight data recorders are designed to be small and thoroughly fabricated to withstand the influence of high speed impacts and extreme temperatures. A flight data recorder from a commercial aircraft can be seen in Figure 1.

State-of-the-art high density flash memory devices have permitted the solid state flight data recorder (SSFDR) to be implemented with much larger memory capacity. A large number of aircraft are now equipped with solid-state recorders and no longer use disk drives. In the past ten to fifteen years, the density of memory chips has greatly increased and the ability to

About the Author

Dr. Yair Wiseman was a postdoctoral scholar at the Georgia Institute of Technology in conjunction with Delta Air Lines, Inc. He is now with Bar-Ilan University and Israel Aircraft Industries, Ltd. Correspondence concerning this article should be sent to wiseman@cs.biu.ac.il.

Mr. Alon Barkai completed an MSc of Computer Science at Bar-Ilan University. Mr. Barkai is the Founder and CEO of Ziroon Ltd.

†The authors would like to thank Israel Aircraft Industries, Ltd. for its support.

http://dx.doi.org/10.7771/2159-6670.1075

Figure 1. Flight data recorder.

record thousands of parameters for hundreds of flight hours in flight data recorders or quick access recorders is now possible. Compression algorithms are used by the manufacturers and may become even more prevalent with the introduction of video recorders. New video compression schemes have a significant compression factor which is usually some hundreds of times; that is, the compressed file will be less than 1% of the size of original file (Horowitz et al., 2012). This means that the compression is still useful, even though the memory capacity is much larger. This work has been done relative to hard disks of flight data recorders, but flash memory developers can utilize the results, as well.

An ordinary difficulty is that flight data recorders run out of space on hard disks. The concern of encountering this difficulty leads one to act cautiously, constantly attempting to reduce the used data space (Wu, Banachowski, & Brandt, 2005). In addition, working with nearly full disks causes the allocation of new file blocks to be distributed across multiple platters. Working with files scattered around the hard disk drive is slow and very demanding on the read/write head, with unnecessary overhead (Ng, 1998). However, unlike flight data recorders, in regular desktops the vast majority of disks are not overloaded and so it is better to keep old versions of important files on the disk even though, in most cases, one will not use the old versions (Muniswamy-Reddy, Wright, Himmer, & Zadok, 2004).

Data are often processed by embedded systems. In the embedded computing world and especially in flight data recorders, it is clear that the storage problem is significant, as the storage area is hundreds of times less than the storage space available on desktop computers. In a common embedded computer system there is an electronic card with a simple processor that supports a small solid state device which provides barely 1 to 4 GB of space for the system files. Usually it is not possible to add additional storage space such as a hard disk drive or even SD reader because of hardware constraints, system constraints, size constraints, and power consumption constraints (Yaghmour, Masters, Gerum, & Ben-Yossef, 2008). A photograph of a flight data recorder's storage device can be seen in Figure 2.

It is difficult to install a full operating system environment which includes a compilation chain (Tool Chain) and

Figure 2. Flight data recorder's storage device.

GUI (X Server) in such a small storage space. For the purpose of illustration, a 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, occupies 1.5 GB.

The easiest solution to this problem is removing features, installing only the essentials, and developing lighter applications for the embedded cards of flight data recorders.

More profitable solutions include the use of disk data compression (Benini, Bruni, Macii, & Macii., 2002; Roy, Kumar, & Prvulovic, 2001). Other devices can use compression of rarely-used data, or compression of all data, and expansion only of data needed in run time; but flight data recorders can assume all the data is rarely-used. Compressing the data will directly yield more storage space without losing any information. However, this has a serious impact on system performance, especially when a relatively small process is located on the same electronic card that needs to simultaneously compress the file being written to the disk while continuing running the other applications without compromising them. For this reason, embedded developers usually do not use file system compression in order to not harm valuable system performance.

With the aim of solving this problem and get the best of both worlds, the researchers offer a decision algorithm which decides, at runtime and according to current available system resources, whether a file should be compressed and, if so, which compression algorithm and strength to use. In the worst case, in which the system is very loaded, none of the new files will be compressed. However, in most cases, that is not the situation and on average most files will be compressed using either weak or strong compression algorithms. Accordingly, use of the new file system can only improve today's flight data recorders.

Related Work

This section describes the research and development related to compression in embedded systems for memory and file systems. Both memory and file systems have a similar problem of minimum size because of attempts to reduce the product's cost and size. Several aspects were investigated where real-time compression can provide a significant improvement:

1) Hardware-based memory compression and software based memory compression. These improve system performance by reducing the use of I/O means of storage and increasing the amount of memory available to applications, and
2) Compression of the file system itself, read-only or read-write, in which the main goal is to reduce the consumption of storage media capacity and reduce the consumption of I/O transfer of compressed data.

Hardware Based Memory Compression

(Benini et al., 2002) proposed to introduce a compression/decompression element between the RAM and the cache, so that any information in the RAM would be saved in a compressed format and all data in the cache would be uncompressed.

Kjelso, Gooch and Jones (1996; 1999) proposed a hardware-based compression scheme for memory. Their algorithm, X-match, using a dictionary of recently-used words, is designed for hardware implementation.

Software Based Memory Compression

(Yang, Dick, Lekatsas and Chakradhar, 2010) showed that using online compression to compress memory pages that were moved to the storage device (to the swap) in embedded systems significantly improves the size of usable memory (about 200%) almost without compromising performance or power consumption (about 10%).

Swap compression (Tuduce & Gross, 2005; Rizzo, 1997) compresses pages that were evacuated from the memory and keeps them in compact form in the software cache which is also located in RAM. Shared memory issues, however, have not been addressed in this system (Geva & Wiseman, 2007). Cortes, Becerra, & Cervera (2000) also investigated the implementation of the Swap compression mechanism in the Linux kernel to improve performance and reduce memory requirements. It seems natural to assume that if the compression of the swap pages which are saved in a storage device produces a significant improvement then, for similar considerations, so would the compression of the rest of the files in the storage device.

Read-only File Systems

In embedded Linux environments, there are several options for a compressed file system that offers a solution to the problem of the small storage space that exists in these small systems (Hammel, 2007). Most of the compressed file systems are read-only. It is a consequence of the ease of implementation and the high performance cost of run-time data compression that performance of the applications in low-resource cases might be hurt. Typically, two file systems are used, one for read-only files which are not going to be changed, and a second uncompressed read-write file system for the files that do change. The user should create beforehand a compressed image of the file system.

CramFS (2010) is a read-only compressed Linux file system. It uses Zlib compression for each separate page of each file and so it allows random access to data. The meta-data is not compressed, but effectively kept smaller to reduce the space consumed.

SquashFS (2008) is a famous compressed file system in the Linux environment. It uses the GZIP or LZMA algorithms for compression. But the drawback is that it is read-only and so it is not intended for routine work, but rather for archival purposes.

Cloop (Kitagawa et al., 2006) is a Linux module that allows a compressed file system to be supported by a Loopback Device (Lekatsas, Henkel and Wolf, 2000). This module allows transparent decompression at run-time when an application is accessing the data without the knowledge of how files are saved in practice.

CBD (Kitagawa et al., 2006) is a Linux kernel patch that adds support for a compressed block device designed to reduce volumes of file systems. CBD is also read-only and works with a block device, as does Cloop. Data written to the device is saved in memory and never sent to the physical device. It uses the Zlib compression algorithm. It should be noted that in some set of circumstances, CBD may cause a kernel crash because of kernel stack overflow (Wiseman, Isaacson & Lubovsky, 2008).

Compressed Read-write File Systems

Implementation of a compressed file system with the ability for random-access write is much more complicated and difficult. We provide some examples of such file systems here.

ZFS is a file system created by Sun Microsystems. ZFS is used under the Solaris operating system, and is also supported in other operating systems such as Linux, Mac OS X Server, and FreeBSD. ZFS is known for its ability to support high capacity, integrating concepts from file management and partitioning management, innovative disk structures, and a simple storage management system. ZFS is an open source project (Rodeh & Teperman, 2003).

One of the features of ZFS is support for transparent compression. The compression algorithm is configurable by the user; available algorithms are LZJB, GZIP, or none (Oracle, 2010). Both LZJB and GZIP are fixed and deterministic. They do not depend on the characteristics of system resources available only during the compression-only file content. The choice of which algorithm to use or the option to not use compression at all is decided by the system administrator in advance and this choice is used in all cases.

FuseCompress (2009) is a Linux file system environment which has transparent compression to compress the file's content when written to the storage device and decompress the data when it is being read from the device. This is done in a transparent manner so the application does not know how the files were really saved; it can therefore work with any application transparently. Compression is executed on the fly, and currently supports four compression algorithms: LZO, ZLIB, BZIP2, and LZMA. The missing feature is the choice of which algorithm is the best one to use at the moment of compression need. The algorithm is selected by the user in advance when mounting the file system.

In the Microsoft NTFS environment there is an option to compress selected files so that application will still be able to access and use them while their data is transparently decompressed when needed. This option is not automatic and the user must give a specific command and select the files that he wants to keep in a compressed format. There is only one algorithm, LZ77, in use for all compressed files. There are only two options for a file: with or without compression (Makatos et al., 2010).

DriveSpace (initially known as DoubleSpace) is a disk compression utility supplied with MS-DOS starting with version 6.0. The purpose of DriveSpace is to increase the amount of data the user can store on disks by transparently compressing and decompressing data on the fly. It is primarily intended for use with hard drives, but use with floppy disks is also supported. However, DriveSpace belongs in the past since FAT32 is not supported by DriveSpace tools and NTFS has its own compression technology ("compact") native to Windows NT-based operating systems instead of DriveSpace (Qualls, 1997).

Finally, Sun Microsystems has a patent related to file system compression using a concept of "holes". A mapping table in a file system maps the logical blocks of a file to actual physical blocks on disk where the data is stored. Blocks may be arranged in units of a cluster, and the file may be compressed cluster-by-cluster. Holes are used within a cluster to indicate not only that a cluster has been compressed, but also the compression algorithm used. Different clusters within a file may be compressed with different compression algorithms (Madany, Nelson, & Wong, 1998).

Adaptive Compressed File System

We propose to improve space utilization by adding adaptive compression features to ZFS, FuseCompress, or others. If good results are obtained for memory pages that are saved in the storage device in compressed format, then one would expect similar results when other files are also saved in a compressed format.

The file system which is being proposed, ACFS (Adaptive Compressed File System), will make use of some features in the authors' previous work (Wiseman, Schwan & Widener, 2003) and will demonstrate better performance in low resource conditions or in loaded systems than other file systems. Its superior performance is attributed to features that existing file systems do not take into account; in particular, the ability to dynamically decide at runtime whether to compress the file data and which algorithm is the best one to use considering the available resources of the system at that particular moment. To our knowledge, no currently existing file system takes into account current system characteristics while saving a file.

The ACFS algorithm can be described as follows: Let us denote a compression type as C. For example, known algorithms which have been used by Yaghmour et al. (2008) include Czip-fastest, Czip-best, Crar-fast, Crar-good, Clzw, and Cnone. We refer to different compression levels as different compressions, and will use only lossless compression algorithms (Zhang, 2012; Arnold & Bell, 1997).

Let us denote a group of compression algorithms as X; for example, X = {Czip-best, Czip-fast, Cnone}. The number of compression algorithms in a group is |X|. For example, we can select a group X, where |X| = 3, which contains:

1. A strong compression algorithm which can highly compress the data; however, it takes a lot of CPU power and memory while compressing, such as BWT (Burrows & Wheeler, 1994).
2. A weak compression algorithm which uses fewer system resources while compressing, but is also less effective in the compression rate of the data, such as (Huffman, 1952).
3. An identity algorithm which does not compress at all; it will produce the same output as input, as such it will not take any resources while compressing.

When there are no resources available while compressing, we will use the third algorithm. When the system is idle, we will want to use the first algorithm (the strongest one). And when the system is doing some other things but there are still available resources, we will want to use the second, lighter algorithm.

Let us denote by R the total available system resources, as percentages, where R=[0, 100]. From R, we will choose which compression algorithm to use. The value of R will be

calculated based on resources available at runtime, at the moment that the compression algorithm has to be chosen. Note that there are many different properties which can affect R values; for example, available CPU, available RAM, available disk space, and available DMA.

These properties do not have to be only available resources; they can also be more subtle properties such as the number of I/O requests at a recent time, or estimation of compression of a certain file type which is to be compressed (we do not wish to use many system resources for trying to heavily compress a file which is already compressed). Sometimes I/O operations can be concurrently executed with CPU tasks (Wiseman & Feitelson, 2003).

We start by identifying the properties we want to consider. Each of these properties will be presented on a scale of percentages between 0, which means not available, to 100, which means it is completely available.

There are two options to consider: 1) each time we need to make a choice, we recalculate each of these properties' values, and 2) recalculate only when a certain amount of time has passed since the last calculation, to minimize the burden on system resources. Property information is received in two dimensions (each property has its value) and we need to convert this to a single dimension value. This can be done by several different means; the simplest is choosing the worst property. That means we select our compression algorithm in relation to the least available resource, as apparently it is the bottleneck.

Other methods could include calculating the average of all values using weights of the importance of each property, or, more appropriately, calculating the desired compression level by taking into account the results of past decisions. At the end of the process we obtain a single value R indicating how much resources are available, and this value will be used in selecting the strength of the compression algorithm that we will use to compress the data.

We define a function F that for each instance of a group X will give a value Y from the range 0 to 100. This Y value is the minimum value of free resources that is needed so that we can use this compression method. This function does not need to be linear, but it must be monotonically increasing. To do this, we will sort the group X by its Y values from smallest to the largest, when Cnone (no compression) will always result in a value of 0. This function is depicted in Figure 3.

We define the Select function as C = Select(X,F,R), which will get the compressions group that we would like to use (X), the available resources required for every compression function (F), the currently available resources (R), and will return the selected compression algorithm to use (C). This is done by simply choosing the best match respecting minimal resources required for each given compression algorithm.

In theory, the S function has to perform a binary search of R in F because F is a monotonic increasing function. However, in practice, |X| is very small, so a simple run on F searching for the first value Y which is equal or greater than R could be much faster than a binary search, resulting in complexity of O(1).

Calculating the R values of the different properties will take O(n), but because of the small number of properties and the fact that it is not dependent on the size of the data being compressed, we may say this time is constant. Only the time of running the selected compression algorithm and actually writing the compressed information to the storage device is substantial.

We will show later that the decompression time is not really an issue, and most of the time it is better than the time required to read the uncompressed data from the storage device.

There exist some special lossless compressions designed for better compression of certain file types that perform

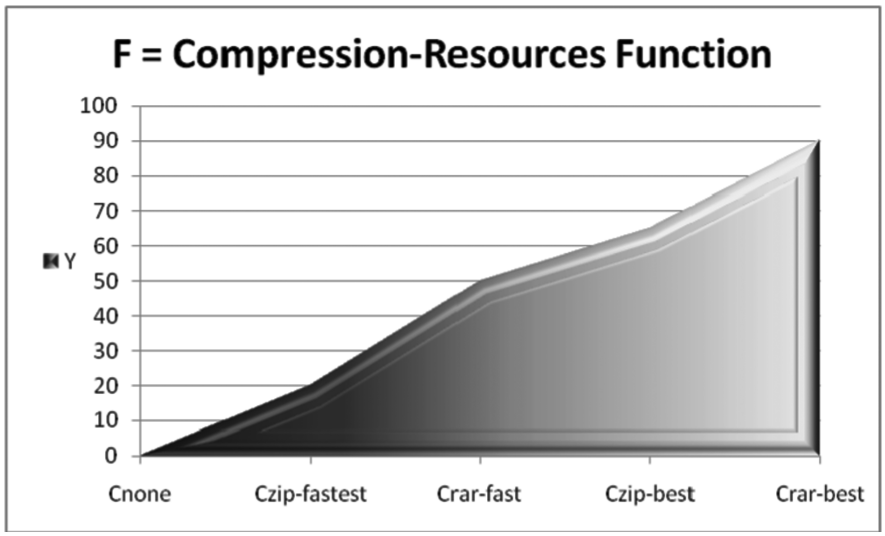


Figure 3. The compression-available resource function.

better than general compression algorithms. PNG compression for BMP files is an example. Moreover, one can tweak some general purpose algorithms to be more effective using a change of parameters; for example, by specifying that the file is a video or a text file. In addition to selecting the compressing algorithm by available resources, one could extend the ACFS algorithm and select the compression by file type, also. We could define some general file types such as text, executable, graphics, or other. For every file type, we will define a group of compression algorithms X and a function F; for example, Xtext, Ftext. For each file, we will analyze the file for file type using methods suggested by FileType (2008) and McDaniel and Heydari (2003). Using this information, the algorithm will call the Select function with the X and F that are related to this file type. If the file type is unknown, it will fall into the other group. Figure 4 demonstrates the selection flow of a compression algorithm according to both file type and available resources.

In most systems the resources are not always fully used; sometimes they are less available and sometimes they are more available. Because of the ability to measure availability of resources, one could go back and compress files using a stronger and better compression algorithm those files or file parts which had been compressed using a lighter compression because at the time they were written to the storage device there were no available resources.

In this manner, we can develop ACFS as a file system with delayed compression which will take advantage of the idle times of the system for compressing the data that is saved quickly at busy system times. This dynamic reassessment will maximize the space usage of the storage device, since once all data is compressed using the strongest compression available then there is no way to achieve a higher data-per-space rate.

An additional feature of ACFS is that it can increase efficiency by adding frequency of file usage to decision parameters. We can easily monitor the number of recent accesses to a file. When we detect a file that is more frequently accessed than other files, we can lower its compression complexity to a lower compression algorithm in the same group X.

Putting together both expansions above, delayed compression and lowering the compression complexity for commonly used files, we achieve a file system that will dynamically increase or lower the compression complexity of files in idle times by the history of their usage.

A third extension, different compression groups for different file types, can be added. This allows increasing or lowering the selected compression algorithm within the

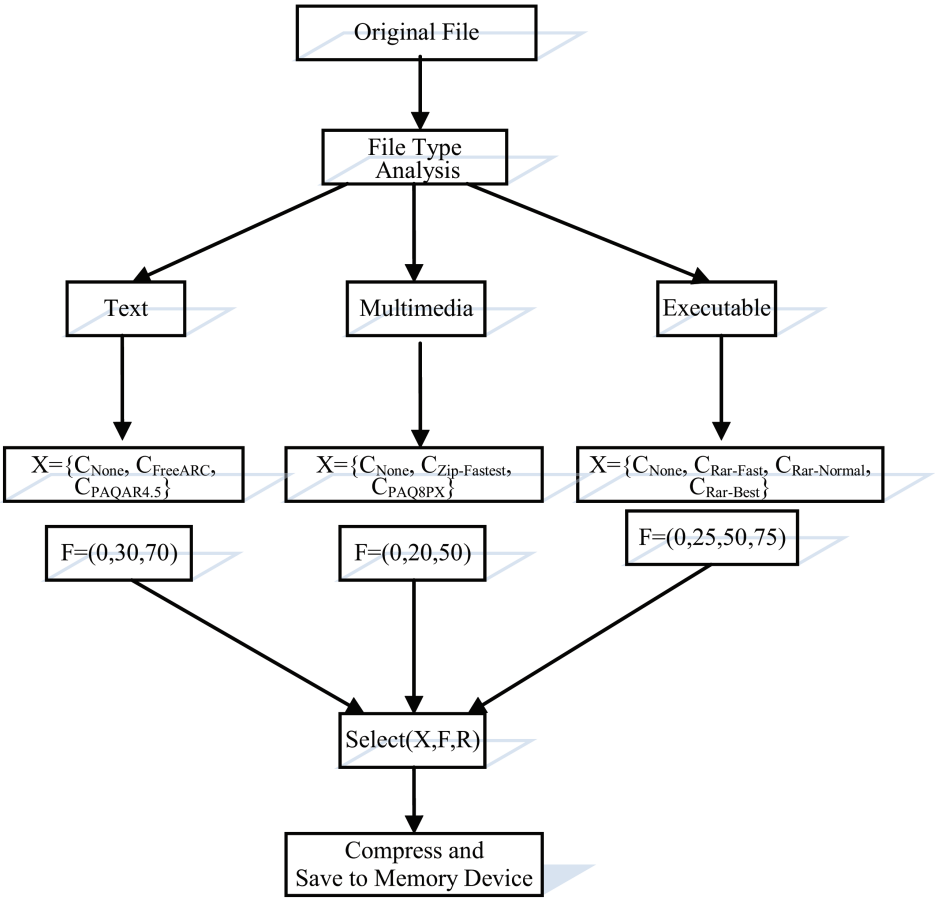


Figure 4. Selecting the compression algorithm.

compression algorithms group relevant to this specific file type.

A file system with these properties will be very efficient because it adapt based on the usage of the system, providing close to maximum compression and maximum performance simultaneously.

Implementation

We do not need to implement a completely new file system. Instead, we can take an existing open source compressed file system and improve it by changing it to dynamic selection of compression algorithms as previously described.

For this work we have chosen the FuseCompress file system that was described in the related work section. This file system compresses all files (except file types that are already in compressed format) using a previously defined compression algorithm, and is a good starting point for the proposed development.

All of the files′ operating system APIs will be redirected to our user-space application which will actually perform the requested operations. That application will return the information from the real files when browsing the directory, and if some application reads data from a file in this folder, the uncompressed version of the data will be returned to the read API. Copying files into the virtual mount point or creating or changing file there will cause the data to be compressed on-the-fly and saved in the real directory as compressed files, while being shown at the virtual directory as uncompressed. Reading files from the virtual directory will cause on-the-fly decompression and return the uncompressed original data to the reader application. Thus, any application can work with the compressed file system exactly as it would a normal file system without awareness of the compression.

Evaluation

The test-set for this evaluation was the/lib directory of a standard Ubuntu 10.10 installation, with a size of 233 MB. The computer was an Intel Core i7 950 CPU @ 3.07 GHz (8 Cores, so host processes would not interfere with the tests), 6 GB of RAM, a Windows 7 64 bit operating system with an Ubuntu 10.10 32 bit virtual machine, a 60 GB hard drive with Flash SSD for the operating system and + 1 TB WDC SATA III for data. The virtual machine can allow the availability of only one CPU when needed. A parallel approach can be also applied, as suggested by Klein and Wiseman (2003; 2005).

Firstly, we executed a normal folder copy command "cp-r" to copy the test-set from its original location to a different location that was not inside a compressed file system mount point, taking 17 seconds. Then we executed the same command with a destination location in the compressed file system virtual folder, taking 53 seconds, or 312% of the original time, due to heavy compression calculations occurring while writing the files to the destination directory.

To evaluate the decompression time we copied the files from the compressed location to uncompressed location, thereby reading the files to cause the decompression. The copying time decreased from 17 seconds to 14 seconds, or 82% of the original time, since less data had to be read from the slow hard disk. The fact that less I/O is involved indicates another benefit of using a compressed file system.

Next we evaluated the effectiveness of an on-the-fly file system compression. We used the same collection of files that were used in the previous test. The size was reduced from 233 MB to 105 MB; that is, 45% of the original size. Figure 5 shows the results in logarithmic scale.

Then we tested the effects on system resources while compressing. When a compression was invoked, the CPU

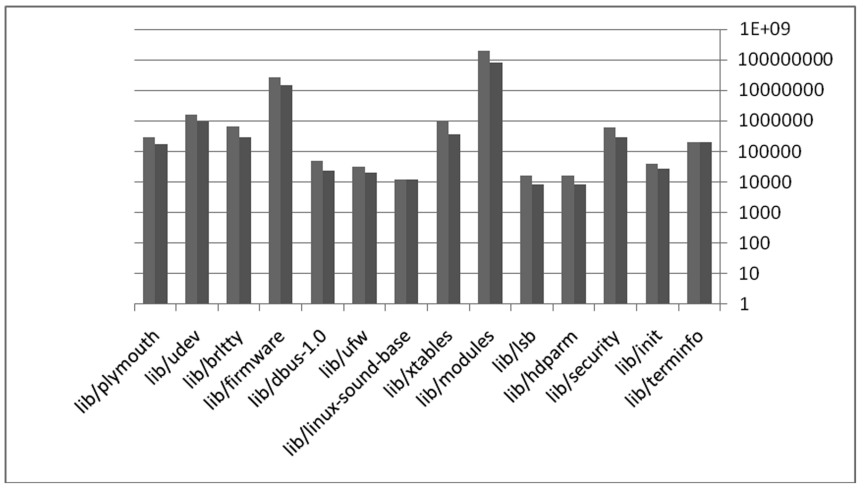


Figure 5. Compression ratio in logarithmic scale. The bright bars indicate the original folders; the dark bars indicate the compressed folders.

was loaded to 96.0% in user mode, 4.0% in kernel mode and 0.0% idle. The memory usage was 502196 kB. When no compression was invoked, the CPU was loaded 2.0% in user mode, 1.0% in kernel mode and 97.0% idle. The memory usage was 498720 kB.

Also, while compressing, we can see that a new user-mode process appears to handle the compression, which takes 94.9% CPU and 2.2% memory. Here, the compression process is taking nearly all of the CPU power, while it has almost no effect on the memory usage.

If the CPU is needed for other assignments, these assignments will have poorer performance. With the ACFS, we overcome this problem by selecting and switching compression algorithms on the fly according to available CPU power in order to avoid a performance decline. A possible performance decline is the main drawback of using compressed file systems as a default in all computer systems. If we can eliminate this potential performance decline by actively monitoring and avoiding it, dynamically selecting different levels of compression algorithms, or even choosing not to compress at all, then we will have no problem using a compressed file system in every computer system. By doing so, we will attain the benefits of a compressed file system without the drawbacks.

Another test we made was conducted by implementing a CPU-Eater process. This process had a loop that executed a very small constant set of instructions called the "Frame". The process counted the number of frames that had been executed and each second printed how many frames it could execute in that second. Obviously, if any other program takes the CPU, it will harm the performance by decreasing the number of loops.

We compared three scenarios. In the first, Test #1, we invoked the CPU-Eater test process alone, when no other operation takes place in the system. In the second, Test #2, we checked the performance of our innocent application (the CPU-Eater process) while an on-the-fly compression takes place for the copied files. In Test #3 we copy files without performing any compression. In this last test since there is an application running in the background that takes high amounts of CPU ($> 90\%$), ACFS will decide not to use compression at all, just like a normal file copying operation.

We measured the performance by the average FPS (Frames Per Second) rate the innocent application (the CPU-Eater process) generates and by average %CPU metric. The results are an average of 50 executions and are shown in Figures 6 and 7.

Test #1 is a reference case; since no file operation occurred in the time of the test, these values are a maximum. In Test #2 the FPS rate decreased to 33.3% and the %CPU metric was reduced by half while the compression algorithm takes the other half. Test #3 shows that a normal file copy operation barely harms the performance of processes, whereas the CPU consumption of the compression operation heavily harms the performance of other running processes.

We can see that the %CPU metric of the innocent application was noticeably reduced when compressing was used to compress the files using the on-the-fly compressed file system, whereas it returned to normal when copying files with no compression. This indicates that by using ACFS applications overall performance will not suffer due to less available CPU because of file system compression operations. If the CPU is busy, the compression strength will be automatically reduced or even completely turned off, so it will not consume more CPU cycles than the available idle CPU cycles.

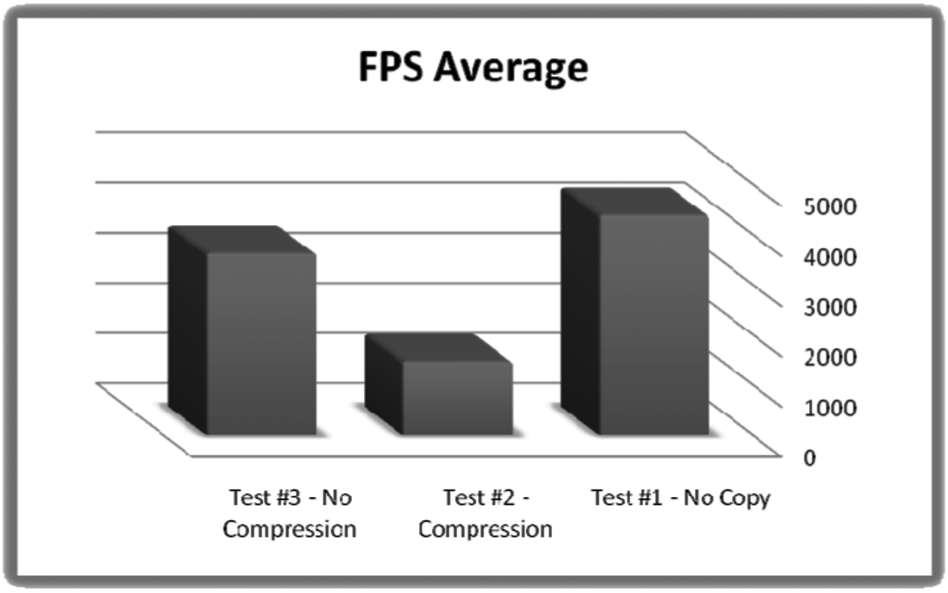


Figure 6. Average FPS rate.

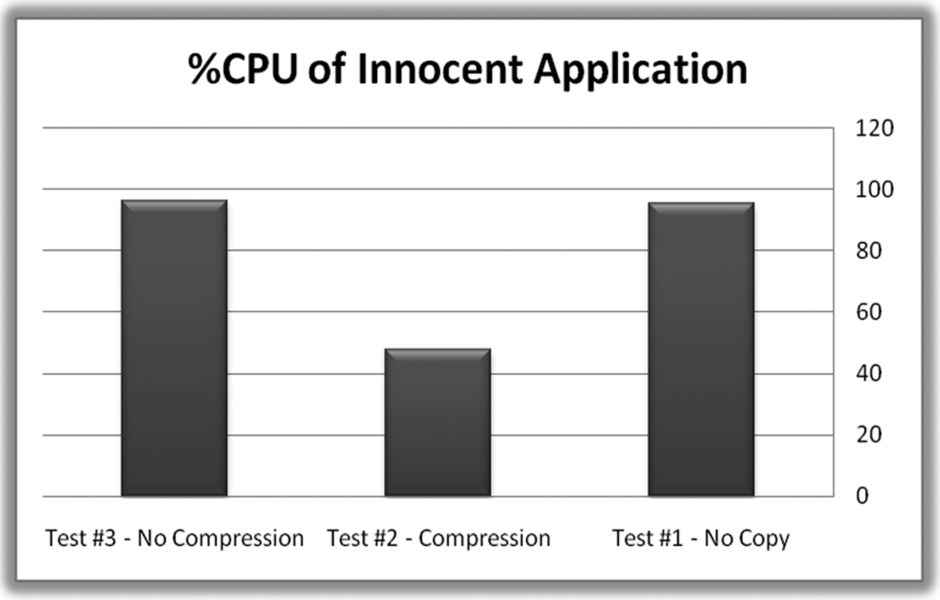


Figure 7. CPU-Eater process %CPU metric.

Optimally, if a process uses a certain number of CPU cycles (e.g., 50%), while a compression does not take place we will attempt to continue using the same number of CPU cycles at the same level of performance and the compression process will only use the idle resources. Then, the compression algorithm itself can be chosen according to available idle CPU resources.

We also tested different priorities of both the user's process and the compression process. The user task for this test was the CPU-Eater program from the previous tests. This program takes as much CPU resources as are available (~100%) and shows its actual performance as FPS value.

In this test we invoked long copy operations into a compressed file system while the user process was running. We executed this test multiple times with different nominal task level values and took the average of each value. The results are shown in Table 1.

One can clearly see from the results that the division of the CPU cycles between the user's process and the compression process is strongly affected by the level setting. This means that the priority setting does have an effect on how much impact the user task will have when a compression is taking place. Actually, the operating system scheduler increases or decreases the time slice of the processes according to the user task level value, explaining why the %CPU metric is nearly linear in that value.

Table 1
Priority Effect on FPS and %CPU

FPS	%CPU Compression	%CPU User Task	User Task's Level
1000	70%	25%	0
1800	55%	40%	-5
2850	30%	65%	-10
3600	11%	85%	-15
4150	3%	95%	-20

It is also apparent from these results that there exists a setting (-20) at which a user's task will result in almost no reduction in performance. In this situation a compression will be very long; however, the file system should select a lighter compression method or no compression when writing files.

Changing the priority of the compression process, however, has no effect on the results. Setting the compression process level to any priority with each of the different user process levels gives almost the same results. Unlike the user's process, the compression process adapts itself to employ only free CPU cycles.

In the previous tests we employed a 100% CPU intensive process because this is the worst case scenario, but this is not how an embedded system normally operates. Our objective is that a user's task will have minimal reduced performance and our compression algorithm will only use the remaining idle CPU cycles. Consequently, another test examined different levels of CPU consumption. The CPU-Eater process from the previous tests was slightly modified so it would set manually to use only a certain amount of %CPU. The change is actually that the process sleeps during a certain part of each second. All the tests here were run with a -20 task level setting so the user task will have a minimal impact. The results are shown in Table 2. In this Table, "Alone" indicates that only the user process is executed with no compression. The values are average values.

We can clearly see that none of the tests shows a noteworthy reduction of the performance of the innocent user's process while copying files to a compressed file

Table 2
Different Levels of CPU Consumption

FPS while compressing	%CPU of the compression itself	%CPU of User Task while compressing	FPS Alone	%CPU Alone	Sleep Setting (In mSec)
4250	2%	96%	4300	98%	0
3100	20%	73%	3200	75%	220
2000	47%	47%	2100	50%	500
1000	70%	25%	1000	25%	750

system, and that the compression algorithm used only unclaimed CPU cycles.

Conclusions

Data that generated by embedded systems are commonly kept on a hard disk, but the hard disk of the typical embedded computer system is small (Lekatsas, Dick, Chakradhar, & Yang, 2005; Xu, Clarke, & Jones, 2004). In particular, flight data recorders have small disk drives and generally only write the disk drives and almost never read them; therefore, compression can be beneficial for such systems. The ACFS suggests a method of using a compressed file system while ensuring that the innocent other tasks will not experience reduced performance. The compression algorithm (whichever algorithm is chosen by the file system) will only use the available CPU cycles.

The file system should select the compression algorithm strength and whether to compress or not in real time based on available CPU resource, so that the application that waits for the file operation to be completed will not have to wait too long.

References

Arnold, R., & Bell, T. (1997). A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the Data Compression Conference* (pp. 201–210). http://dx.doi.org/10.1109/DCC.1997.582019

Benini, L., Bruni, D., Macii, A., & Macii, E. (2002). Hardware-assisted data compression for energy minimization in systems with embedded processors. *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition* (p. 0449). http://dx.doi.org/10.1109/DATE.2002.998312

Burrows, M., & Wheeler, D. (1994). *A block-sorting lossless data compression algorithm* (SRC Research Report 124). Palo Alto, CA: Digital System Research Center.

Cortes, T., Becerra, Y., & Cervera, R. (2000). Swap compression: Resurrecting old ideas. *Journal of Software: Practice and Experience*, *30*(5), 567–587. http://dx.doi.org/10.1002/(SICI)1097-024X(20000425)30:5<567::AID-SPE312>3.0.CO;2-Z

Cramfs. (2010). *Cramfs: Cram a filesystem onto a small ROM*. Retrieved from http://sourceforge.net/projects/cramfs

FileType. (2008). *FileType detection system, open source project*. Retrieved from http://pldaniels.com/filetype

FuseCompress. (2009). *FuseCompress file system*. Retrieved from http://code.google.com/p/fusecompress/

Geva, M., & Wiseman, Y. (2007). Distributed shared memory integration. *Proceedings of the IEEE Conference on Information Reuse and Integration*, (pp. 146–151). Washington, DC: IEEE Conference Publications. http://dx.doi.org/10.1109/IRI.2007.4296612

Hammel, M. J. (2007). Embedded Linux: Using compressed file systems. *LWN.net*. Retrieved from http://lwn.net/Articles/219827

Horowitz, M. J., Kossentini, F., Mahdi, N., Xu, S., Guermazi, H., Tmar, H., … 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, A. G. Tescher, Ed., *Proceedings of SPIE, Volume 8499* (p. 84990W). http://dx.doi.org/10.1117/12.953235

Huffman, D. A. (1952). A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, *40*(9), 1098–1101. http://dx.doi.org/10.1109/JRPROC.1952.273898

Kitagawa, K., Tan, H., Abe, D., Chiba, D., Suzaki, K., Iijima, K., & Yagi, T. (2006). File system (Ext2) optimization for compressed loopback device. In *Proceedings of the 13th International Linux System Technology Conference*. http://unit.aist.go.jp/itri/knoppix/ext2optimizer/tmpfiles/LinuxKongress_Ext2optimizer_kitagawa.pdf

Kjelso, M., Gooch, M., & Jones, S. (1996). Design and performance of a main memory hardware data compressor. In *Proceedings of the 22nd EUROMICRO Conference* (pp. 423–430). Washington, DC: IEEE Conference Publications. http://dx.doi.org/10.1109/EURMIC.1996.546466

Kjelso, M., Gooch, M., & Jones, S. (1999). Performance evaluation of computer architectures with main memory data compression. *Journal of Systems Architecture*. *45*(8), 571–590. http://dx.doi.org/10.1016.S1383-7621(98)00006-X

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

Klein, S. T., & Wiseman, Y. (2005). Parallel Lempel Ziv coding. *Journal of Discrete Applied Mathematics*, *146*(2), 180–191. http://dx.doi.org/10.1016/j.dam.2004.04.013

Lekatsas, H., Dick, R. P., Chakradhar, S., & Yang, L., (2005). CRAMES: Compressed RAM for embedded systems. In *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (pp. 93–98). Washington, DC: IEEE Conference Publications. http://dx.doi.org/10.1145/1084834.1084861.

Lekatsas, H., Henkel, J., & Wolf, W. (2000). Code compression for low power embedded system design. In *Proceedings of the 37th Design Automation Conference* (pp. 294–299). New York: Association for Computing Machinery. http://dx.doi.org/10.1109/DAC.2000.855323

Madany, P. W., Nelson, M. N., & Wong, T. K. (1998). *U.S. Patent No. 5,774,715*. Washington, DC: U.S. Patent and Trademark Office.

Makatos, T., Klonatos, Y., Marazakis, M., Flouris, M. D., & Bilas, A. (2010). Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the 5th European Conference on Computer Systems* (pp. 1–14). New York: Association for Computing Machinery. http://dx.doi.org/10.1145/1755913.1755915

McDaniel, M., & Heydari, M. H. (2003). Content based file type detection algorithms. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*. Washington, DC: IEEE Conference Publications. http://dx.doi.org/10.1109/HICSS.2003.1174905

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* (pp. 115–128). Berkeley, CA: USENIX Association.

Ng, S.W. (1998). Advances in disk technology: Performance issues. Computer, *31*(5), 75–81. http://dx.doi.org/10.1109/2.675643

Oracle (2010). *Oracle Solaris ZFS administration guide*. Retrieved from http://docs.oracle.com/cd/E19253-01/819-5461/index.html

Qualls, J. H. (1997). The PC corner. Business Economics, *32*(3), 70–72.

Rizzo, L. (1997). A very fast algorithm for RAM compression. *ACM SIGOPS Operating Systems Review*, *31*(2), 36–45. http://dx.doi.org/10.1145/250007.250012

Rodeh, O., & Teperman, A. (2003). zFS-A scalable distributed file system using object disks. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies* (pp. 207–218). Washington, DC: IEEE Conference Publications. http://dx.doi.org/10.1109/MASS.2003.1194858

Roy, S., Kumar, R., & Prvulovic, M. (2001). Improving system performance with compressed memory. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium* (p. 66). Washington, DC: IEEE Conference Publications.

SquashFS (2008). *SquashFS*. Retrieved from http://squashfs.sourceforge.net.

Tuduce, I. C., & Gross, T. (2005). Adaptive main memory compression. In *Proceedings of the USENIX 2005 Annual Technical Conference* (pp. 237–250). Berkeley, CA: USENIX Association.

Wiseman, Y., & Feitelson, D. G.. (2003). Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, *14*(6), 581–592. http://dx.doi.org/10.1109/TPDS.2003.1206505

Wiseman, Y., Isaacson, J., & Lubovsky, E. (2008). Eliminating the threat of kernel stack overflows. In *Proceedings of the IEEE International Conference on Information Reuse and Integration* (pp. 116–121). Washington, DC: IEEE Conference Publications. http://dx.doi.org/10.1109/IRI.2008.4583015

Wiseman, Y., Schwan, K., & Widener, P. (2004). Efficient end to end data exchange using configurable compression. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems* (pp. 228–235). Washington, DC: IEEE Conference Publications. http://dx.doi.org/10.1109/ICDCS.2004.1281587

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* (pp. 189–194). New York: Association for Computing Machinery. http://dx.doi.org/10.1145/1065983.1066026

Xu, X., Clarke, C. T., and Jones, S. R. (2004). High performance code compression architecture for the embedded ARM/Thumb processor.In, S. Vassiliadis, J. L. Gaudiot, & V. Piuri, Eds., *Proceedings of the 1st Conference on Computing Frontiers* (pp. 451–456). New York: Association for Computing Machinery. http://dx.doi.org/10.1145/977091.977154

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

Yang, L., Dick, R. P., Lekatsas, H. and Chakradhar, S. (2010). Online memory compression for embedded systems. *ACM Transactions on Embedded Computing Systems*, *9*(3), 1–29. http://dx.doi.org/10.1145/1698772.1698785

Zhang, C. S. (2012). Design of real-time lossless compression system based on DSP and FPGA, In C. S. Zhang, Ed., *Materials Science and Information Technology*, (pp. 4173–4177). Durnten-Zurich, Switzerland: Trans Tech Publications, Inc.