DIMINISHING FLIGHT DATA RECORDER SIZE

Yair Wiseman and Alon Barkai, Bar-Ilan University, Ramat-Gan, Israel
wiseman@cs.biu.ac.il

Abstract

Flight Data Recorders produce data that is stored on an embedded memory device. A widespread problem with these devices is that the embedded memory device runs out of space. The concern of getting to this problematic situation causes the software of the flight data recorder to work in a careful manner - it constantly makes efforts to minimize the used memory space; otherwise a larger flight data recorder will be required that will also cost more money. In this paper we propose using a compressed file system with the aim of having a better cost-effective memory usage. The main significant disadvantage that sometimes prevents flight data recorders from using compressed file systems is the intense overload of data compression when writing the data to the memory device, which puts an intolerable overburden on the processor and as a result it harms the system performance. This paper proposes a file system that instantaneously compresses the data of the flight data recorder while taking care that the other tasks' response time will not be harmed.

1. Introduction

A Flight Data Recorder is a small embedded computer device employed in aircraft. Its function is recording any 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 a high speed and the heat of an extreme temperature.

An ordinary difficulty is that Flight Data Recorders run out of space in our hard disk. Our concern of getting into this difficulty leads us to act in a careful manner, a constant attempt to reduce the used data space [1]. In addition, working with nearly full disk 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 [2].

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 we will not be using the old versions [3].

Signals are often processed by embedded systems. Unlike the controversy in the personal computers world, in the embedded computing world and especially in Flight Data Recorders everyone agrees that the problem is significant. Storage area is hundreds of times less than 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 gives us barely 1-4GB of space for the system files. Usually it is not possible to add additional storage space such as Hard Disk Drive or even SD reader because of hardware constraints, system constraints, size constraints, and power consumption constraints [4].

As we know, we cannot install a full operation system environment which includes a compilation chain (Tool Chain), GUI (X Server) in such a small storage space. For the purpose of illustration, a basic installation of 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. While installing window operation system takes much more than that.

The easiest solution for this is removing features, installing only the essentials, and developing lighter applications for the embedded cards of Flight Data Recorders.

More profitable solutions would be the use of disk data compression [5,6]. Other Embedded devices can use compression of rarely used data, or compression of all data, and uncompressing it when needed in run time; whereas Flight Data Recorder can assume all the data as rarely used. Compressing the data will directly give us more storage space, without losing any information. But of course it 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 not to harm valuable system performance.

With the aim of solving this problem and get the best of both worlds – we offer a decision algorithm which decides at runtime, according to current available system resources, if a file should be compressed and if so which method of compression will be used when saving this file to disk. How does it work? In runtime, the file system decides whether to compress the file or not and if so then which compression algorithm and strength to use according to available system resources at that moment. So only in the worst case that 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 algorithm.

Given a file system like this, there is no reason not to use it. Since the worst case is the case that you have today, when no files are being compressed at all. This means, using this new file system can only improve today's Flight Data Recorders!

2. Related Work

This chapter 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 always being too small because of attempts to reduce product's cost and size. Several aspects were investigated where real-time compression can provide a significant improvement:

- 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.
- 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.

2.1 Hardware Based Memory Compression

Benini and Bruni [5] proposed to introduce a compression / decompression element between the RAM and the Cache, so that any information in the RAM would be saved at a compressed format and all data in the cache would be non-compressed.

Kjelso, Gooch and Jones [7,8] proposed a hardware-based compression for memory. Their algorithm X-match, using a dictionary of words that were used recently, is designed for hardware implementation.

2.2 Software Based Memory Compression

Yang, Dick, Lekatsas and Chakradhar [9] showed that using On-Line 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 [10,11,12] compresses pages that were evacuated from the memory and keeps them in compact form in software cache which is also located in RAM.

Cortes, Eles, and Peng [12] also investigated the implementation of the Swap Compression mechanism in the Linux kernel to improve performance and reduce memory requirements. Swap Compression can also alleviate thrashing effects in overloaded system [13].

It seems natural to assume that if the compression of the swap pages which are saved in a storage device gives a significant improvement then, for similar considerations, so would the compression of the rest of the files in the storage device.

2.3 Read Only File Systems

In embedded Linux environments, there are several options for a compressed file system that offer a solution to the problem of the small storage space that exists in these small systems. Most of the compressed file systems are read-only for the ease of implementation, and the high performance cost of run-time data compression which might hurt the performance of the applications in low-resource cases. 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 and only then he can use it.

CramFS [14] is a read-only compressed Linux file system. It uses Zlib compression for each page separately 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 [15] is a famous compressed file system in Linux environment. It uses the GZIP or LZMA algorithms for compression. But the drawback is that it Read-Only and so it is not intended for routine work with its files but it is mostly for archiving purposes.

Cloop [16] is a Linux module that allows a compressed file system to be supported by a Loopback Device [17]. 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 [18] is a Linux kernel patch that adds support for Compressed Block Device designed to reduce volumes of file systems. CBD is also read-only and works with a Block Device as in Cloop. Data written to the device is saved in memory and never being sent to the physical device. It uses the Zlib compression algorithm.

2.4 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 show some examples of such file systems:

ZFS is a file system made by Sun Microsystems. ZFS is used under Solaris operating system, and is also supported in other operating systems such as Linux, Mac OS X Server, and FreeDSD. ZFS is known for its ability to support high capacity, integrating concepts from file management and partitioning management, innovative disk structure, and a simple storage management. ZFS is an open source project [19].

One of its features is that it supports transparent compression. The compression algorithm is configurable by the user. It can be one of the following: LZJB or GZIP, or no compression.[20] Both of these algorithms 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 not to use compression at all is decided by the system administrator in advance and this choice is used in all cases.

FuseCompress [21] is a Linux file system environment which has transparent compression to compress the file's content when they are being written to the storage device and decompress the data when it is being read from the device. This is being done in a transparent way so the application doesn't know how the files were really saved, and so it can work with any application transparently. Compression is being executed On-The-Fly, and currently supports 4 compression algorithms: lzo, zlib, bzip2, 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 NTFS of Microsoft for Windows 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 in use for all compressed files and it is LZ77. There are only 2 options for a file: with or without compression [22].

DriveSpace (initially known as DoubleSpace) is a disk compression utility supplied with MS-DOS starting from version 6.0. The purpose of DriveSpace is to increase the amount of data the user could store on disks, by transparently compressing and decompressing data on-the-fly. It is primarily intended for use with hard drives, but use for floppy disks is also supported. However, DriveSpace belongs to 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 [23].

Finally, Sun Microsystems has a patent about 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 [24].

3. Adaptive Compressed File System

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

Our file system ACFS (Adaptive Compressed File System) which is being suggested here will show better performance in low resources or loaded system than any other file system. 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 follow:

Let us denote a compression type as C.

For example, known algorithms which can be used [5]:

Czip-fastest, Czip-best, Crar-fast, Crar-good, Clzw, Cnone

We refer to different compression levels as different compressions.

We use only lossless compression algorithms [25,26].

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, like BWT [27].
2. A weak compression algorithm which uses less system resources while compressing, but it also less effective in the compression rate of the data like Huffman [28].
3. An identity algorithm which does not compress at all, it will produce the exact same output as input, as so it will not take any resources while compressing.

When there are no resources available while compressing, we will want to use the 3rd algorithm. When the system is not doing anything else (idle) we will want to use the 1st 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 2nd lighter algorithm.

Let us denote by R the total available system resources, as percentages, when R=[0-100].

By the value of R, we will choose which compression algorithm to use. The value R will be calculated based on resources available at runtime, at the moment that the compression algorithm has to be chosen.

However, how can we calculate this R value? There are many different properties which can affect the R values. For example: Available CPU, available RAM, available disk space, and available DMA.

These properties do not have to be only available resources properties, they can also be more subtle properties like the number of I/O requests at the recent time, or estimation of compression of a certain file type which we are about to compress (we do not wish to use much system resources for trying to heavily compress a file which is already compressed in its nature).

We start by identifying the properties we want to take in account.

Each one 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. 2) Recalculate only when a certain amount of time has passed since the last calculation, to minimize hurting the system resources. Property information is received in 2-dimensions (each property has its value) and we need to convert it to a single dimension value. This can be done by several different ways; 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 ways could be calculating the average of all values using weights of the importance of each property. Or, a smarter choice that calculates the desired compressing level by taking into account the results of past decisions. At the end of the process we get 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.

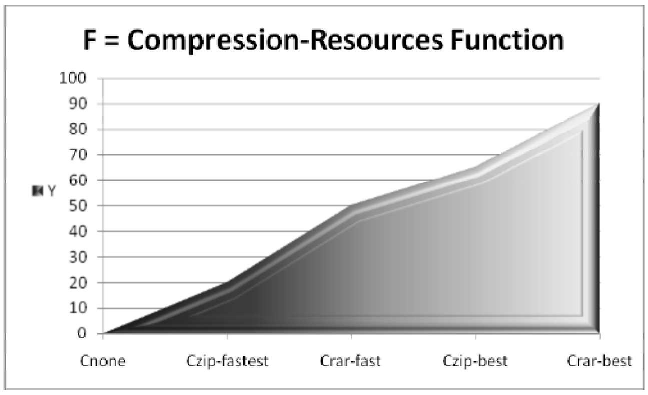

Figure 1. The Compression-Available Resource Function

We define a function F that for each instance of a group of X will give a value Y from the range 0 to 100. This Y value is the minimum value of free resources that are needed so that we can use this compression. This function does not need to be linear, but it must be a monotonically increasing function. To do so we will sort the group X by its Y values from smallest to the largest, when Cnone (no compress) will always get the value of 0. This function is depicted at Figure 1.

We define the Select function as follows:

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 basically just 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 and so we could say the complexity is O(1).

Calculating the R values of the different properties will take O(n), but because of the small number of properties and that it is not depended on the size of 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 decompressing time is not really an issue, and most of the time it is even better than reading the uncompressed data from the storage device.

There exist some special lossless compressions designed for better compression of certain file types and that perform better than general compression algorithms. For example PNG compression for BMP files.

Moreover, one can tweak some general purpose algorithms to be more effective by a change of parameters to get better results. For example, telling it that this file is a video or a text file. In addition to selecting the compressing algorithm by available resources, we could extend the ACFS algorithm and select the compression by file type too. We could define some general file types like: Text, Executable, Graphics, Other. For every file type we will define a group of compression algorithms X and an F function of its own. For example: Xtext, Ftext. For each file, we will analyze the file for file type (by [29] or [30]), and with this information in hand we 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.

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

This way we can turn the ACFS to a file system with delayed compression which will take advantage of the idle times of the system for compressing the quickly saved data 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 better 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.

We added a third extension - different compression groups for different file types. This allows increasing or lowering the selected compression algorithm within the compression algorithms group relevant to this specific file type.

A file system with these properties will be very efficient because it will change itself according to the usage of the system. It will adapt to the system and will provide close to both maximum compression and maximum performance at the same time!

4. Implementation

We do not need to implement a whole new file system from the grounds up. Instead we could take an existing open source compressed file system, understand it, and improve it by changing it to dynamic selection of compressing algorithms while evaluating available resources at run time.

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 compressing algorithm. It comes with four different compression algorithms that the user can choose from: lzo, zlib, bzip2, lzma. This file system is a good start for us and can provide the basic start for our needs. We implement our changes on top of this file system.

All of the files' operating system APIs will be redirected to our user-space application which will actually do the requested operation. It could be anything, but in this case it will return the information of the real files when browsing the directory and if some application reads data from a file in this folder we will return the uncompressed version of the data to this read API. Copying files into the virtual mount point, creating or changing file there will cause the data to get compressed on-the-fly and saved at 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.

That way any application can work with the compressed file system exactly as a normal file system without knowing about the compression. It will work on the virtual directory without knowing that actions are actually made on the real folder which holds the compressed versions of the files.

5. Evaluation

Our test set for this evaluation was the /lib directory of a standard Ubuntu 10.10 installation. Size: 233MB. The computer was with CPU: Intel Core i7 950 @ 3.07GHz (8 Cores, so host processes will not interfere with the tests), RAM: 6 GB, Operation System: Windows 7 64bit with Virtual machine of Ubuntu 10.10 32bit, Hard Disk: 60 GB Flash SSD (For OS) + 1 TB WDC SATA III (For Data). The Virtual machine can help us to virtually have just one CPU when needed. A parallel approach can be also applied as we suggested at [31,32].

Firstly, we executed a normal folder copy command "cp -r" to copy the test-set from its original location to a different location which is not inside a compressed file system mount point and it took 17 seconds. Then we executed the same command but the destination location is in the compressed file system virtual folder. And it took 53 seconds. That is 312% of the original time because heavy compression calculation had taken place while writing the files to the destination directory.

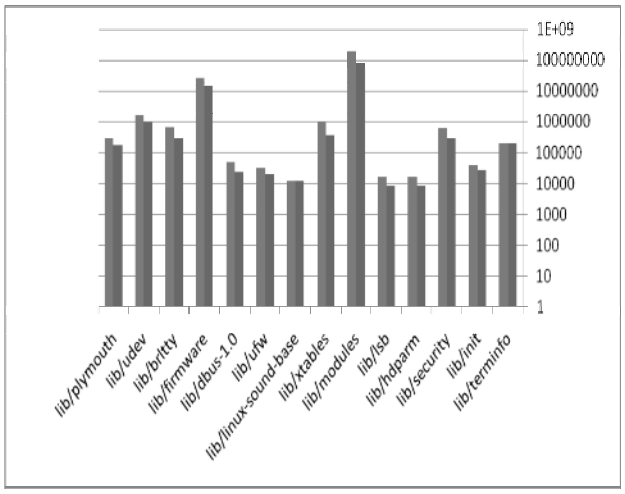

Figure 2. Compression Ratio in logarithmic scale.

To evaluate the decompression time we copy the files from the compressed location to uncompressed location. Thus, reading the files to make the decompression happen. The copying time reduced from 17 seconds to 14 seconds. That is 82% or the original time. It happened because less data has to be read from the slow hard disk. That shows another benefit of using a compressed file system – less I/O is involved! Faster reads from the disks, which makes out of the added decompression calculations time when reading the files.

5.1 Real-Time File System Compression

We evaluated the effectiveness of a real-time 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 2 shows the results in logarithmic scale. Bright Bars – Original folders, Dark Bars – Compressed Folders.

Then we test the effect on system resources while compressing. When a Compression was invoked the CPU was loaded 96.0% in user mode, 4.0% in kernel mode and 0.0% idle. The memory usage was 502196k used. 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 498720k.

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.

We can see that while compressing, 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 default in all computer systems. If we can eliminate this possible 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 computers systems. We can only 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. We call this constant set of instructions "Frame". The process counted the number of frames that had been executed and each second prints 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 3 scenarios. In the test #1 we invoked the CPU-Eater test process alone, when no other operation takes place in the system. In 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 which takes high amounts of CPU (> 90%) ACFS will decide not to use compression at all, just like normal file copying operation.

We measured the performance by the average FPS (Frames Per Second) the innocent application (the CPU-Eater process) generates and by average %CPU. The results are an average of 50 executions and are shown in Figures 3 and 4.

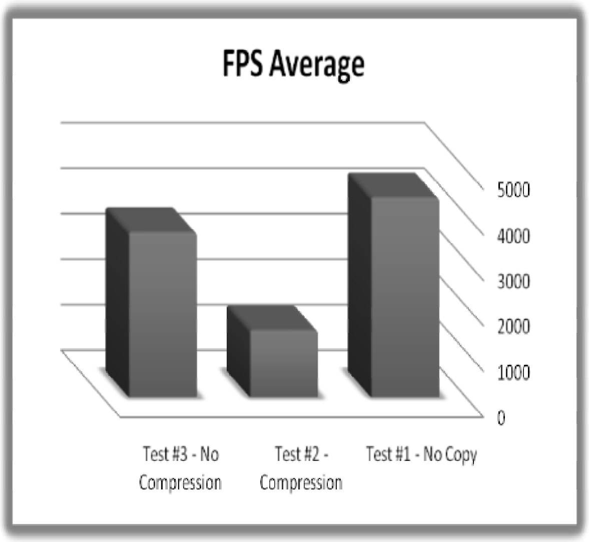

Figure 3. Average of FPS

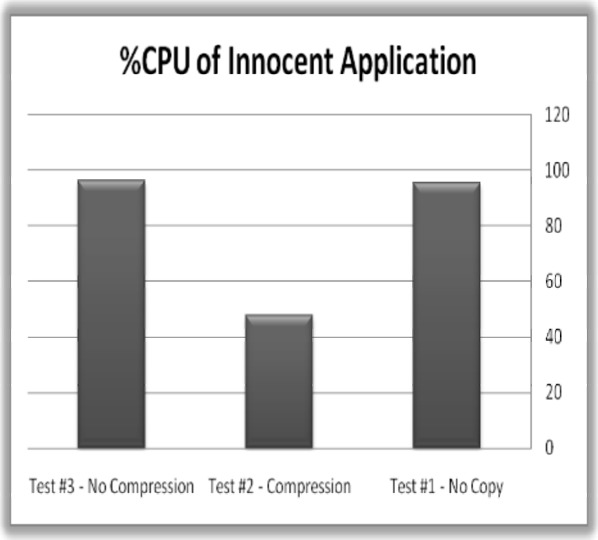

Figure 4. %CPU of the CPU-Eater process

Test #1 is a reference case; since no file operation was taken place in the time of the test it means these values cannot get any higher. In test #2 the FPS reduced to 33.3% and the %CPU 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.

This is the main reason why users do not use compressed file systems at all. Our goal is creating a file system that does not harm the performance of the other processes.

We can see that the %CPU of the innocent application noticeably reduced when compressing was used to compress the files on the on-the-fly compressed file system, whereas it got back to normal when copying files with no compression.

This means that by using ACFS applications will not suffer from 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.

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

5.2 Priorities

We also tested different priorities on both the user's process and the compression process. The user task for this test is the CPUEater program from the previous tests. This program takes as much as available CPU (~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 is running. We executed this test multiple times with different nice values and we took the average of each nice value. The results are shown in Table 1.

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

Table 1. Priority effect on FPS and %CPU

In these results we can clearly see that the division of the CPU cycles between the user's process and the compression process is strongly affected by the nice 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 nice value, so this explains why the %CPU is nearly linear in the nice value.

We can also learn from these results that there exists a setting (-20) that a user's task will have almost no reduction of performance. Although in this setting a compression will be very long, but in these situations 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 nice level to any priority with each of the different user process nice levels gives almost the same results. Unlike the user's process, the compression process adapts itself to employ only free CPU cycles.

On 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. So we took another test with different levels of CPU consumption. The CPU eater process from the previous tests was slightly modified so it would only use a certain amount of %CPU as can be manually set. The change is actually that the process sleeps a certain part of each second. All the tests here run with a -20 nice setting so the user task will have a minimal impact. The results are shown in Table 1. In this figure "Alone" means when only the user process is executed with no compression. The values are average values.

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

table 2. Different levels of CPU consumption

We can clearly see here that on each and every tested CPU consumption, there was nearly no reduced performance of the innocent user's process while copying files to a compressed file system, and the compression algorithm used only unclaimed CPU cycles.

6. Conclusions

Avionic Systems' memory should be handled efficiently [33] despite the fact that the memory device of embedded computer system is typically small [34,35]. In particular, Flight Data Recorders have small memory devices and in addition Flight Data Recorders just write the memory device and almost never read them; therefore a compression can be beneficial for such systems.. The ACFS suggests a way of using a compressed file system while making sure that the innocent other tasks will have no reduced performance. The compression algorithm (whichever algorithm will be 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 at real time based on available CPU resource, so the application that waits for the file operation to be completed will not wait too long.

References

[1] Wu J. C., Banachowski S. and Brandt S. A., Hierarchical disk sharing for multimedia systems, Proceedings of the international workshop on Network and operating systems support for digital audio and video, pp. 189-194, 2005.

[2] Ng, S.W., Advances in disk technology: Performance issues, IEEE Computer, Vol. 31(5), pp. 75-81, 1998.

[3] K. Muniswamy-Reddy, C. P. Wright, A. Himmer and E. Zadok, A Versatile and User-Oriented Versioning File System, in Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004), pp. 115-128, San Francisco, California, March 31-April 2, 2004.

[4] Yaghmour K., Masters J., Gerum P. and Ben-Yossef G., Building embedded linux systems, O'Reilly Media, Inc., 2008.

[5] Benini, L., Bruni, D., Macii, A., and Macii, E., Hardware-assisted data compression for energy minimization in systems with embedded processors. In Proc. Design, Automation &Test in Europe Conf., 2002.

[6] Roy, S., Kumar, R., and Prvulovic, M., Improving system performance with compressed memory. In Proc. Parallel & Distributed Processing Symp., 2001.

[7] Kjelso, M., Gooch, M., and Jones, S., Design and performance of a main memory hardware data compressor. In Proc. Euromicro Conf. 423–430., 1996.

[8] Kjelso, M., Gooch, M., and Jones, S., Performance evaluation of computer architectures with main memory data compression. In J. Systems Architecture. Vol. 45. 571–590, 1999.

[9] Yang Lei, Dick Robert P., Lekatsas Haris and Chakradhar Srimat, "Online memory compression for embedded systems", ACM Transactions on Embedded Computing Systems, volume 9(3), pp. 1-29, 2010.

[10] Tuduce, I. C. and Gross, T., Adaptive main memory compression. In Proc. USENIX Conference, 2005.

[11] Rizzo, L.. A very fast algorithm for RAM compression. Operating Systems Review 31, 2, (Apr.), pp. 36–45, 1997.

[12] Cortes, T., Becerra, Y., and Cervera, R., Swap compression: Resurrecting old ideas.Software-Practice and Experience Journal 30 (June), 567–587, 2000.

[13] Reuven M. and Wiseman Y., Medium-Term Scheduler as a Solution for the Thrashing Effect, The Computer Journal, Oxford University Press, Swindon, UK, Vol. 49(3), pp. 297-309, 2006.

[14] Cramfs. Cramfs: Cram a filesystem onto a small ROM. http://sourceforge.net/projects/cramfs

[15] SquashFS - http://squashfs.sourceforge.net

[16] Cloop. Cloop: Compressed loopback device. http://www.knoppix.net/docs/index.php/cloop

[17] Lekatsas, H., Henkel, J., and Wolf, W., Code compression for low power embedded system design. In Proc. Design Automation Conf. 294–299. 2000.

[18] CBD. CBD compressed block device. http://lwn.net/Articles/168725

[19] ZFS - http://en.wikipedia.org/wiki/ZFS

[20] Oracle Solaris ZFS Administration Guide. Chapter 6 -Managing Oracle Solaris ZFS File Systems. http://docs.sun.com/app/docs/doc/819-5461/gavwq?a=view

[21] FuseComress File System http://miio.net/wordpress/projects/fusecompress

[22] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, “Using Transparent Compression to Improve SSD-based I/O Caches.”,

ACM/SIGOPS European Conference on ComputerSystems (EuroSys), 2010.
[23] Qualls, J. H., The PC Corner, BUSINESS ECONOMICS, VOL 32; NUMBER 3, pp. 70-72, 1997.
[24] Madany; Peter W. (Fremont, CA), Nelson; Michael N., (San Carlos, CA) Wong; Thomas K. (Pleasanton, CA). Patent #5774715 - File system level compression using holes. Application: 08/623,907, Assignee: Sun Microsystems, Inc. (Palo Alto, CA). U.S Patent Documents: 5155484; 5237675; 5481701; 5551020; 5652857.
[25] Cai Suo Zhang, Design of Real-Time Lossless Compression System Based on DSP and FPGA, Materials Science and Information Technology, 4173-4177, 2012.
[26] Arnold, R., & Bell, T., A corpus for the evaluation of lossless compression algorithms. In Designs, Codes and Cryptography, pp. 201-210, 1997.
[27] Burrows, M. & Wheeler, D. Block sorting Lossless Data Compression Algorithm, System research center, research report 124, Digital System Research Center, Palo Alto, CA, 1994.
[28] Huffman, D., A method for the Construction of Minimum Redundancy Codes Proc. of the IRE Vol. 40, pp. 1098-1101, 1952.
[29] FileType – File-Type detection system, open source project - http://pldaniels.com/filetype
[30] Mason McDaniel, M. Hossain Heydari, "Content Based File Type Detection Algorithms," hicss, vol. 9, pp.332a, 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, 2003 http://www.computer.org/portal/web/csdl/doi/10.1109/HICSS.2003.1174905.
[31] Shmuel T. Klein & Yair Wiseman, Parallel Huffman Decoding with Applications to JPEG Files, The Computer Journal, Oxford University Press, Swindon, UK, Vol. 46(5), pp. 487-497, 2003.
[32] Shmuel T. Klein & Yair Wiseman, Parallel Lempel Ziv Coding, Journal of Discrete Applied Mathematics, Vol. 146(2), pp. 180-191, 2005.
[33] Weisberg P. and Wiseman Y., Efficient Memory Control for Avionics and Embedded Systems, To Appear, International Journal of Embedded Systems, 2013.
[34] Yang, L., Dick, R. P., Lekatsas, H., and Chakradhar, S., CRAMES: Compressed RAM for embedded systems. In Proc. Int. Conf. Hardware/Software Codesign and System Synthesis. 2009.
[35] Xu, X. H., Clarke, C. T., and Jones, S. R., High performance code compression architecture for the embedded ARM/Thumb processor. In Proc. Conf. Computing Frontiers. 451–456, 2004.