IMAGE COMPRESSION FOR A SMALL SATELITTE – LESSONS LEARNED

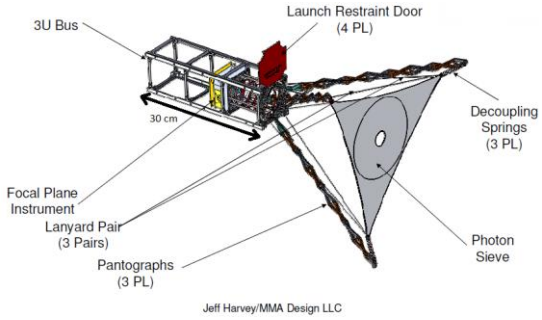
Wayne Brown
Department of Computer Science
The United States Air Force Academy
2354 Fairchild Dr., Suite 6G-101
USAF Academy, CO 80840-6208
719-333-3590
Wayne.Brown@usafa.edu

ABSTRACT

A software system for lossless compression of images was developed for a micro satellite with limited RAM memory. Valuable lessons learned during the development of the software are presented, along with some implications for computer science education. The results of the software implementation are also presented.

1 INTRODUCTION

A small micro satellite called FalconSAT-7 [1][2] designed to take high-fidelity images of the sun is being developed by the Physics department at the United States Air Force Academy. The camera images will be gray-scale at 10 bits per pixel. If the camera images were downloaded as raw data, only one image per day could be transmitted to the ground station due to limited communication time. Image compression, even in a limited scope of 3 to 1, could greatly enhance the number of images accessible from the satellite.



The brain of the micro satellite is an Atmel AVR32. Memory consists of 32K of SRAM, 256K of Flash memory, and a 2GB SD Card [3]. The onboard camera takes 1040x1360 pictures and stores each 10 bit pixel value in 2 bytes of memory. A single image requires 2.70 MB of memory. Scientific analysis of the images mandates that no loss of information be allowed in their storage or transmission. The paper describes the development of a software system that performs lossless compression on gray-scale images using very limited RAM.

2 PREVIOUS WORK

Image compression has been a very active research area for over 25 years. The Joint Picture Expert Group (JPEG) [4] was organized in 1986, and issued the first JPEG standard in 1992. The JPEG standard is actually a collection of compression techniques bundled into a single specification. Most software systems that implement the JPEG compression schemes do not implement the entire JPEG standard. Software that claims to implement the JPEG standard must implement a baseline set of functionality. The JPEG standard was designed to compression color images and is lossy. In 1993 a lossless scheme was added to the standard, called Lossless JPEG (JPEG-LS) [5], but it is not commonly included in JPEG implementations. Work on image compression has been ongoing, with the JPEG 2000 compression standard being published in 2000 and amended as recently as 2008 [6].

A reasonable introduction to data compression can be found in the book "Introduction to Data Compression" by Khalid Sayood [7]. A comprehensive treatment of lossless compression techniques can be found in David Adams' PhD thesis *Reversible Integer-to-Integer Wavelet Transforms for Image Coding* [8]. An investigation into image compression on portable multimedia devices which have low memory and low power issues can be found in Ranjan Kumar Senapati's PhD thesis [9].

3 IMPLEMENTATION AND RESULTS

The software to implement lossless compression on high-fidelity, gray-scale images was implemented in C for the FalconSAT-7 satellite. The compression software was designed to minimize memory usage. The decompression software runs on ground station computers using normal computer resources and does not attempt to minimize memory usage. The compression software reads raw data from the camera where each two bytes in Little Endian order represents one pixel intensity value. The output of the compression software is a unique bit-stream format designed for this application. The decompression software reads the bit-stream, recreates the original raw data and stores it in a file. The output file can easily be visualized or converted to a standard image file format using simple MATLAB commands.

The software performs an integer wavelet transform on the rows of an image and separates the even and odd values into two sub-images: one containing the "averages" and one containing the "differences." An integer wavelet transform is then performed on the columns of these two sub-images and again the even and odd values are separated to produce four sub-images. Five possible encodings of the sub-images were developed: fixed-length encoding, run-length encoding, table-lookup encoding, bit-plane encoding, and Huffman encoding. The details of each encoding can be found in the implemented software. The number of bits needed to encode a sub-image is calculated for each of the five possible encodings. Then an integer wavelet transformation is applied to each sub-image to create four new sub-images. If the four sub-images can be encoded using fewer bits than an encoding of the original sub-image, the sub-division is kept and used for the final bit-stream. Otherwise, the sub-division is ignored and the sub-image is encoded using the most efficient encoding scheme available. This is done recursively to find the best overall encoding using the least number of bits. The recursive depth is currently limited to 3 levels, which creates a maximum of 127 sub-images.

An estimate of the memory usage of the software is provided in the table below. The memory usage is calculated based on an image size of 1040 by 1360. Very deliberate choices of data types (short vs. int) and the use of parallel arrays instead of arrays of structures were used to minimize data memory usage. As the table shows, most of the memory usage is from the code itself and not the data it manipulates. Further minimization of the memory usage will have to come from reduction in code, which could be done by removing some of the encoding choices that are rarely used.

Software Component	Memory in bytes
Compression code	93,079
1D integer buffer (1360 * 4)	5,440
Sub-array structures (127 * 60)	7,620
Huffman encoding tables (50 codes max)	816
Total memory usage:	**106,955**

The software produces lossless compression ratios of approximately 2.5 to 1. The software is freely available and can be obtained by contacting the author.

4 LESSONS LEARNED

Development of the compression software for the satellite took many wrong turns. In an effort to "learn from one's mistakes", this section documents lessons that need to be remembered for future software development projects. These lessons can be applied to a broad range of software engineering projects and they should be included in any undergraduate computer science education.

Be Data Mindful

The satellite was under development and its camera specifications were known, but no images of the sun had been taken from the camera. An image of the sun was provided that was similar to what the camera would be expected to produce. Unbeknown to the author, the image was created by taking a 24 bit color image, converting it to gray-scale, and then scaling each pixel value by 4 to create 10 bits per pixel. This image was run through the OpenJPEG software with the lossless flag set to true and a compression ratio of 16-to-1 was produced for the output image. This result was reported back to the satellite team but they were cautioned against expecting a 16-to-1 compression ratio for the final version. However, 16-to-1 became the benchmark target. Bad data created an unrealistic expectation for the software.

The original JPEG compression standard is based on the Discrete Cosine Transform (DCT). The standard includes both a lossless and lossy mode of operation. The DCT is calculated using floating point math on unsigned integers but is not lossy. JPEG becomes lossy when the quantization phase throws out high frequency data that the human eye typically can't see. The JPEG 2000 standard is based on wavelets and includes both a lossless and lossy mode of operation. The wavelets are also calculated using floating point math on integers, but wavelets are fundamentally lossy; it is not possible to recover the original integer data after a typical wavelet transform. The author was very familiar with DCT compression, but decided to implement wavelets because the literature claims that wavelets give approximately 5 to 10% better compression ratios over DCT based compression. Having a good understanding of DCT compression, the author made an incorrect assumption that JPEG 2000 did lossless compression using wavelets without quantization, just as the original JPEG scheme can do with DCT. Regrettably the bad data used for software testing was happy to confirm the wrong assumption. After much development work and then testing on a wider range of images the approach proved to be fundamentally flawed.

Lesson learned: bad data created unrealistic expectations for the software project and wasted a huge amount of development time implementing an invalid solution. As a teacher, having good data at the beginning of a software project is something I have not emphasized enough to my students.

Don't use sample data to verify an algorithm

As discussed in the previous section, bad data was used in the initial develop of the software for test cases to validate the algorithms that were implemented. The algorithms produced correct results for the data which lead to fundamental misunderstandings of how wavelet functions operate. The fundamental misunderstandings were not discovered until late in the software development cycle, which resulting in much wasted time and effort.

Lesson learned: rigorous test cases are crucial at the beginning of the software development cycle.

Unit Testing is Crucial

In order to save time, the author began implementing the required software pieces without adequate testing along the way. While the pieces were reasonably straightforward, their combination into a system was not. The result was wasted time on debugging and integration testing. The author eventually started from "square one" and implemented "unit tests" for each sub-system. In the long run, the "unit tests" saved huge amounts of time and effort.

Lesson learned: "unit testing" and software engineering practices such as "extreme programming" are not just academic issues that should be taught to students. They are critical to the success of software system development.

Random data makes poor test cases

Creating "unit tests' for all of the special cases an algorithm must properly handle is tedious and time consuming. It is easier to create random data sets and hope they cover all the special cases. This is a bad idea.

Lesson learned: Take the time required to create robust "unit tests" that verify every special case an algorithm must correctly handle. It is time well spent.

Read code, not papers

The JPEG standards for image compression have been widely researched and widely published. However, finding published papers that present JPEG information that is helpful for implementation is very challenging. The papers seemed more interested in confiscating their work than in defining clear ways to replicate it. The author found reverse engineering of the OpenJpeg code library (which implements the JPEG 2000 algorithms) to be much more valuable than reading scholarly articles.

Lesson learned: If an implementation of a software system is available, read the code before reading the published paper.

To illustrate this lesson, the following discussion describes the basic ideas behind lossless compression in JPEG 2000 – ideas that were gained from reverse engineering software instead of clear explanations from articles. Lossless compression in JPEG 2000 is performed using integer wavelets that can be reversed with no loss of data. Integer wavelets have interesting properties and they can be explained in simple terms that most students can grasp. The following discussion presents integer wavelets in layman's terms without obfuscating jargon.

The big idea behind integer wavelets[1] is to transform data in such a way that it requires minimal bits to encode. A simple example will demonstrate this idea. Suppose you have two numbers, 35 and 42, and you want to remember these values using smaller numbers. The average of 35 and 42 is 38.5 and their difference is 7. You can remember 35 and 42 by storing 38.5 and 7. Since 7 is much smaller than the original values, it has the potential to be encoded using fewer bits.

[1] Floating point wavelets perform a similar transformation, but they also divide the data into high and low frequency signals. The human eye can't detect the high frequency data and some of the high frequency data is thrown away to produce lossy compression.

An observant reader will immediately recognize that there is a problem. While the initial two values were integers, the average value is floating point. Is it possible to calculate the average and difference values using only integer math? In turns out you can because of a simple arithmetic fact: The sum and difference of any two integer values will either both be even or both be odd. If the difference value is even, you know the sum of the two numbers was even. If the difference value is odd, you know the sum of the two numbers was odd. Therefore, when you divide the sum by 2 to calculate the average, you can throw away any fractional part because the difference value tells you whether there is a .0 or .5 fractional part to the average value. Using the previous example, 35 and 42, if we stored 38 and 7 to remember the two numbers, we know that the average is actually 38.5 because their difference, 7, is odd.

This idea of storing transformed values instead of the original data is a powerful one and many schemes for transforming the data have been proposed [8]. Any transformation where the original data can be reclaimed without loss of precision is called a "reversible integer to integer" transform. If the transformation produces averages and difference values, it is a wavelet transform. The JPEG 2000 lossless compression transformation is elegant in the way it captures the averages and differences between pixel values and retains all precision. The algorithm is presented in example format to minimize mathematical notation. It is important to recognize that the algorithm is performed in stages, where each stage uses data from the previous stage. The algorithm cannot be understood as formulas, but rather must be understood as a process. To explain the process, it will be performed on the following example data:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Data	18	18	18	23	24	67	45	46

Stage one of the lossless JPEG compression process is to calculate the difference between a pixel and the average of its surrounding pixels. This is done on pixels in odd indexed array slots and typically reduces the magnitude of the values stored in these locations because the pixels in adjacent locations are often similar in magnitude. The formula takes the average of the two pixels on either side of the pixel and subtracts it from its current value. All calculations are performed using integer math, which truncates any fractional results. Notice that for slot 5, the calculated average is 34.5, but 34 is subtracted from the pixel's value. For index 7, the value in slot 6 is used twice.

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Data	18	18 → 0	18	23 → 2	24	67 → 33	45	46 → 1
		18 – ⌊(18+18)/2⌋		23 – ⌊(18+24)/2⌋		67 – ⌊(24+45)/2⌋		46 – ⌊(45+45)/2⌋

In stage two of the process, each even indexed pixel value is increased to remember half of the average of its surrounding pixel values. To compensate for truncation, ½ is added into the calculation. The result of stage two on the example data is shown below. For index 0, the value in slot 1 is used twice.

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Data	18 → 18	0	18 → 19	2	24 → 33	33	45 → 54	1
	18 + ⌊((0+0)/2)/2 +1/2 ⌋		18 + ⌊((0+2)/2)/2 +1/2 ⌋		24 + ⌊((2+33)/2)/2 +1/2 ⌋		45 + ⌊((33+1)/2)/2 +1/2 ⌋	

The two stages perform a transformation of the original pixel values into 2 distinct sequences. The odd indexed values, [0, 2, 33, 1], describe the fine details between adjacent pixels. The even indexed values, [18, 19, 33, 54], describe the overall nature of the image. Notice that the odd indexed values are small compared to the original data. Depending on the type of image being compressed, these values can often be efficiently encoded using schemes such as Huffman encoding, run-length encoding, or bit-wise encoding. By carefully selecting an appropriate encoding, image compression can be achieved.

Although it may not be obvious, the JPEG 2000 lossless transformation is a "reversible integer to integer" transformation. This is demonstrated without explanation in the following two decoding stages which reverse the transform.

Decoding stage one:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Data	18 → 18	0	19 → 18	2	33 → 24	33	54 → 45	1
	18 - ⌊((0+0)/2)/2 +1/2 ⌋		19 - ⌊((0+2)/2)/2 +1/2 ⌋		33 - ⌊((2+33)/2)/2 +1/2 ⌋		54 - ⌊((33+1)/2)/2 +1/2 ⌋	

Decoding stage two produces the original data:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Data	18	0 → 18	18	2 → 23	24	33 → 67	45	1 → 46
		0 + ⌊(18+18)/2⌋		2 + ⌊(18+24)/2⌋		33 + ⌊(24+45)/2⌋		1 + ⌊(45+45)/2⌋

CONCLUSIONS

Appropriate application data, appropriate software engineering processes, and thorough literature analysis are necessary for successful implementation of complex software systems.

REFERENCES

[1] http://www.usafa.edu/df/dfe/dfer/centers/lorc/docs/FalconSAT07.pdf.
[2] http://www.amostech.com/TechnicalPapers/2013/Space-Based_Assets/ANDERSEN.pdf.
[3] http://www.atmel.com/images/doc32000.pdf.
[4] http://en.wikipedia.org/wiki/JPEG
[5] http://en.wikipedia.org/wiki/JPEG-LS#JPEG-LS.
[6] http://en.wikipedia.org/wiki/JPEG_2000.
[7] Sayood, Khalid, *Introduction to Data Compression*, Morgan Kaufmann Publishers, 2000, 636 pages.
[8] Adams, Michael David, *Reversible Integer-to-Integer Wavelet Transforms for Image Coding*, PhD thesis, http://www.ece.uvic.ca/~frodo/publications/phdthesis.pdf.
[9] Senapati, Ranjan Kumar, *Development of Novel Image Compression Algorithms for Portable Multimedia Applications*, PhD thesis, http://ethesis.nitrkl.ac.in/5485/1/thesis.pdf.