How Code Performance Was Measured

Code performance was measured using the VisualVM profiling tool. This was achieved by running the code through the Eclipse IDE(Integrated Development Environment) and starting up the VisualVM CPU profiler simultaneously. Each test was run with a predetermined test image and a halftone dot size of 10 (meaning that the image was drawn using dots of a maximum size of 10). In order to profile the code effectively, a breakpoint was placed on the start of the code (the first line as soon as the application began running), and then the application was run in debug mode. While the application's execution was sitting on this line, VisualVM was launched and CPU profiling was turned on. The application was then resumed, ensuring that the application ran from start to finish through the profiling tool. At the end of the profiling process, a snapshot as seen below was obtained:

Call Tree – Method	Total Time [%]	Total Time	Invocations
main		1,326 ms (100%)	1
com.halftone.original.Image.<init> (String, String, int)		640 ms (48.3%)	1
com.halftone.original.Image.initialiseImage ()		471 ms (35.6%)	1
com.halftone.original.Image.convertToGrayscale ()		387 ms (29.2%)	1
com.halftone.original.Image.squashToFit ()		84.1 ms (6.3%)	1
Self time		0.024 ms (0%)	1
Self time		168 ms (12.7%)	1
com.halftone.original.ImageUtils.calculateHypotenuse (int, int)		0.078 ms (0%)	1
com.halftone.original.Image.write ()		561 ms (42.4%)	1
com.halftone.original.Image.convertToHalftone ()		123 ms (9.3%)	1
Self time		87.9 ms (6.6%)	1
com.halftone.original.ImageUtils.calculateAverage (java.util.ArrayList, int, int,		30.6 ms (2.3%)	11926
com.halftone.original.ImageUtils.calculateCircleDiameter (double, double, dou		5.1 ms (0.4%)	11926
com.halftone.original.Image.crop (int, int)		0.079 ms (0%)	1
Thread-0		0.328 ms (100%)	1
java.util.logging.LogManager$Cleaner.run ()		0.328 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.run ()		0.000 ms (0%)	1

Each snapshot, as seen above, contained the total running time of each of the methods within the application and how they contributed to the overall running time of the application. Several different types of image files were input into the application and ran using the profiling tool in order to obtain a realistic measurement of code performance.

Input Data for Profiling

The input data used for profiling was a 3 : 2 aspect ratio 1.2 megapixel image, a 4 : 3 aspect ratio 1.2 megapixel image, and a 4 : 3 aspect ratio 13 megapixel image. All images were photographs of people's faces as the types of images that will be halftoned are likely to be images of people.

The two 1.2 megapixel images were selected for use in testing because the target device (Nexus 7) has a 1.2 megapixel front facing camera. The aspect ratios of 3 : 2 and 4 : 3 were chosen because these two aspect ratios are most commonly used for outputting images from any image capturing device. Thus these two images were selected in an attempt to emulate the "realistic" types of input that will be given to the application when run on the Nexus 7.

For good measure, an additional 13 megapixel image with aspect ratio 4 : 3 was included as test input as the highest megapixel camera out on the market attached to an Android smartphone is a 13 megapixel camera. It was decided to test this image because it is quite possible that another user may send a photo of themselves from their phone to a user who has the Ye Olde Times app, and there is no guarantee what type of phone that user might have. Thus, in order to cover the worst possible case, it was decided to include this image in the profiling conducted.

The Changes Made and Why They Were Made

It was decided to make changes to the convertToGrayscale() method in the Image class as it can be seen from the above snapshot that this method was always the method which took the longest (disregarding the write method as it did nothing more than use the Java ImageIO.write() method which is necessary to output the image), irrespective of the size and aspect ratio of image being fed into the application. It was decided to change the way in which the grayscale conversion was performed as in the original version of the code, the image passed in was being converted to grayscale by using complicated luminosity formula weightings which aim to get the "true" grey colour out of a single pixel. As multiplication takes more CPU clock cycles than addition, it was decided to change the way that this

formula was being calculated to use no multiplication and instead, addition of the three colours along with a single division operation (to divide by the number of colours) which ideally should be more efficient. It was also decided to remove the hash map that was being used to store all of the grey colours to avoid having to create duplicate Color objects where there were many pixels with the same grey colour. It was decided to make this change because it was deduced that it took a large amount of time to create the hash map in memory and then perform reads and writes to memory every time a grey colour was created. The original code is provided below to demonstrate how grayscale conversion was performed prior to any modifications being made:

```
public void convertToGrayscale()
        {
                // Loop over each pixel in the image
                for (int i = 0; i < height; i++)
                {
                        pixels.add(new ArrayList<Integer>());
                      for (int j = 0; j < width; j++)
                      {
                        // Obtain the pixel's red blue and green values
                        Color pixelRGB = new Color(image.getRGB(j,i));

                        // Obtain grey colour through luminosity formula weightings
                        int grey =
   (int)((pixelRGB.getRed()*0.2126)+(pixelRGB.getGreen()*0.7152)+(pixelRGB.getBlue()*0.0722));

                       /* If the grey color has not been encountered before, put it into the
                        * greyVarients hashmap so it can be reused if the same
                        * grey is encountered again
                        */
                        Color greyColor;
                        if(greyVarients.get(grey)!= null)
                                greyColor = greyVarients.get(grey);
                        else
                        {
                                greyColor = new Color(grey, grey, grey);
                                greyVarients.put(grey, greyColor);
                        }

                        // Update image to grayscale
                        image.setRGB(j, i, greyColor.getRGB());

                        // Store pixel data so we can later process averages of cells without
                        overlapping circles from the halftone effect interfering
                        pixels.get(i).add(grey);
                      }
                }
        }
```

It can be seen from the above code that the integer variable “grey” was being obtained by multiplying each red, green and blue component of a single pixel by separate weightings and then adding those values together in order to get the average grey colour. Additionally, there was a hash map (greyVarients) being used in order to store all of the grey Color objects the first time that a certain type of grey was encountered such that there would be no need to recreate the grey colour every time the same grey value was encountered later on in the image.

The improved code is provided below, which shows modifications which perform a simpler method of averaging red, green and blue values to find the appropriate grey colour and the removal of the hash map for storing the grey Color objects.

```
public void convertToGrayscale()
        {
                // Loop over each pixel in the image
                for (int i = 0; i < height; i++)
                {
                        pixels.add(new ArrayList<Integer>());
                      for (int j = 0; j < width; j++)
                      {
                        // Obtain the pixel's red blue and green values
                        Color pixelRGB = new Color(image.getRGB(j,i));

                        // Obtain grey colour through luminosity formula weightings
                        int grey =
                (int)(((pixelRGB.getRed())+(pixelRGB.getGreen())+(pixelRGB.getBlue()))/3);
```

```
                    // Update image to grayscale
                    image.setRGB(j, i, new Color(grey, grey, grey).getRGB());

                    // Store pixel data so we can later process averages of cells without
                    overlapping circles from the halftone effect interfering
                    pixels.get(i).add(grey);
                }
            }
        }
```

It can be seen in the above modified code that the average grey colour is now being evaluated by simply adding all of the red, green and blue values for the pixel and then dividing them by 3. Then, instead of creating a new instance of the grey colour inside of a variable and then evaluating whether the grey was in the hash map or not, the hash map has been completely removed. The variable holding the "Color" object for the grey colour was also removed as there was no longer any need to keep it once the hash map was removed. Instead, the Color object is created when it is passed into the setRGB method which also saves time.

Implications of these Changes

The implications of making these changes was that firstly, the change to the calculation of the grey colour for each pixel made the image look less well blended (the gradients of grey throughout the image showed sharper transitions than before). This was because the calculation of the grey colour was a lot less accurate when calculated as an addition of all three red green and blue values divided by the number of colours. Below are two images, one where half-toning was performed prior to making the changes to improve code efficiency and the other where half-toning was performed after making the changes. It can be seen in the area encircled in green that there is a visible difference in the harshness of transition from one grey level to another in terms of the gradient in that portion of the image.

Before Modification

After Modification

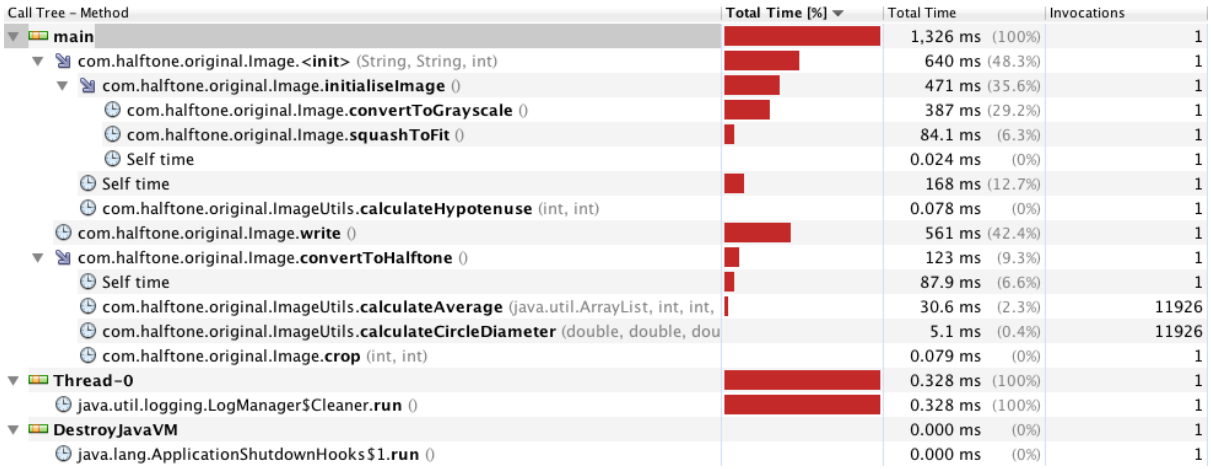
However an additional implication was that the application's ability to efficiently perform garbage collection was now being relied upon because the Color objects were now being instantiated every single time the setRGB method was being called. However, these trade-offs were determined to be reasonable as the improvement in speed as a result of making these changes was relatively significant.

The Results

Before Modification

Below are the snapshots taken in VisualVM upon performing CPU profiling on the code before the aforementioned improvements were made. The code was tested with the 3 images of differing aspect ratios (3 : 2 and 4 : 3) and megapixels (1.2 and 13) and the total time spent executing the methods in the code was recorded in the snapshots. Below are the snapshots along with brief explanations interpreting the results of the profiling conducted.

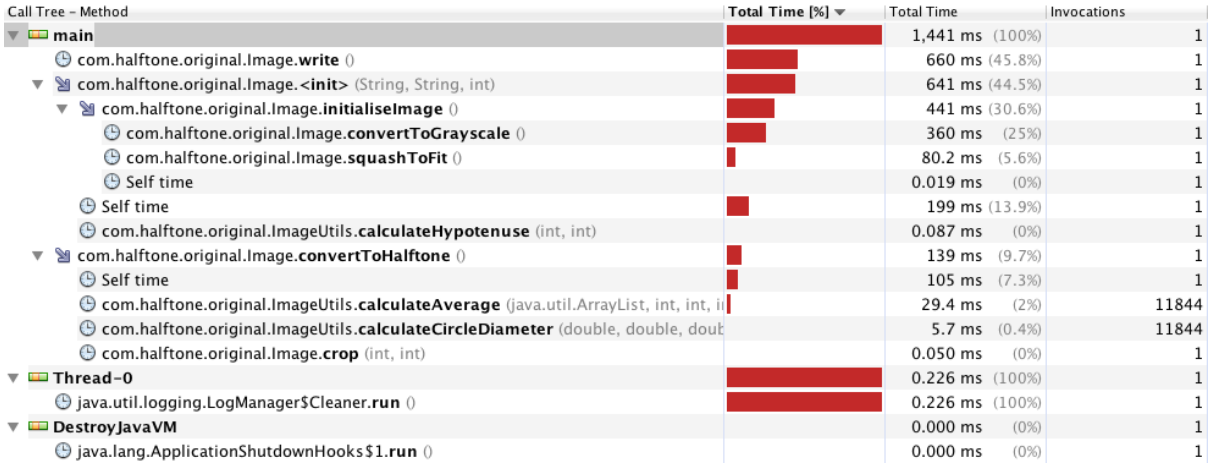
3 : 2 aspect ratio 1.2 megapixel image



Call Tree – Method	Total Time [%]	Total Time	Invocations
main		1,326 ms (100%)	1
com.halftone.original.Image.<init> (String, String, int)		640 ms (48.3%)	1
com.halftone.original.Image.initialiseImage ()		471 ms (35.6%)	1
com.halftone.original.Image.convertToGrayscale ()		387 ms (29.2%)	1
com.halftone.original.Image.squashToFit ()		84.1 ms (6.3%)	1
Self time		0.024 ms (0%)	1
Self time		168 ms (12.7%)	1
com.halftone.original.ImageUtils.calculateHypotenuse (int, int)		0.078 ms (0%)	1
com.halftone.original.Image.write ()		561 ms (42.4%)	1
com.halftone.original.Image.convertToHalftone ()		123 ms (9.3%)	1
Self time		87.9 ms (6.6%)	1
com.halftone.original.ImageUtils.calculateAverage (java.util.ArrayList, int, int,		30.6 ms (2.3%)	11926
com.halftone.original.ImageUtils.calculateCircleDiameter (double, double, dou		5.1 ms (0.4%)	11926
com.halftone.original.Image.crop (int, int)		0.079 ms (0%)	1
Thread-0		0.328 ms (100%)	1
java.util.logging.LogManager$Cleaner.run ()		0.328 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.run ()		0.000 ms (0%)	1

The total time taken running the code on an image with a 3 : 2 aspect ratio and 1.2 megapixels was 1326ms. Write() took the longest time during the process, taking 561ms in total. The second longest method was convertToGrayscale() (inside of the initialiseImage() method) which took 387ms. In terms of overall methods encompassing other methods, initialiseImage took the longest however, as it comprised of a call to both convertToGrayscale and squashToFit(), this was unsurprising.

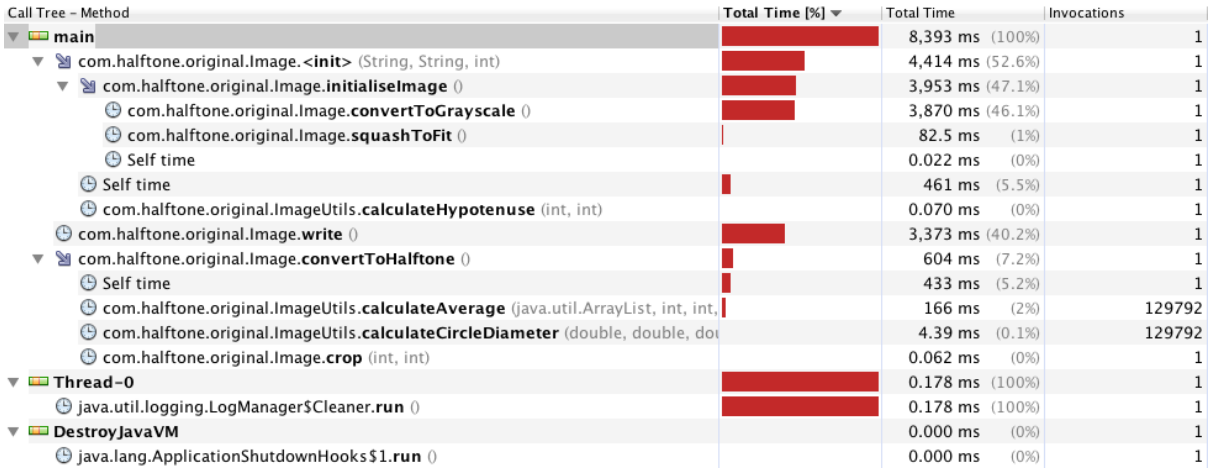
4 : 3 aspect ratio 1.2 megapixel image



Call Tree – Method	Total Time [%]	Total Time	Invocations
main		1,441 ms (100%)	1
com.halftone.original.Image.write ()		660 ms (45.8%)	1
com.halftone.original.Image.<init> (String, String, int)		641 ms (44.5%)	1
com.halftone.original.Image.initialiseImage ()		441 ms (30.6%)	1
com.halftone.original.Image.convertToGrayscale ()		360 ms (25%)	1
com.halftone.original.Image.squashToFit ()		80.2 ms (5.6%)	1
Self time		0.019 ms (0%)	1
Self time		199 ms (13.9%)	1
com.halftone.original.ImageUtils.calculateHypotenuse (int, int)		0.087 ms (0%)	1
com.halftone.original.Image.convertToHalftone ()		139 ms (9.7%)	1
Self time		105 ms (7.3%)	1
com.halftone.original.ImageUtils.calculateAverage (java.util.ArrayList, int, int, i		29.4 ms (2%)	11844
com.halftone.original.ImageUtils.calculateCircleDiameter (double, double, dout		5.7 ms (0.4%)	11844
com.halftone.original.Image.crop (int, int)		0.050 ms (0%)	1
Thread-0		0.226 ms (100%)	1
java.util.logging.LogManager$Cleaner.run ()		0.226 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.run ()		0.000 ms (0%)	1

The total time taken to process the image with a 4 : 3 aspect ratio 1.2 megapixels was 1441ms. Write() took 660ms during the process and convertToGrayscale() took 360ms. By comparison to the 3 : 2 aspect ratio 1.2 megapixel image, it was expected that initialiseImage() would take longer because the 4 : 3 aspect ratio image is slightly larger. However, this was not reflected in the result as initialiseImage() took 441ms by comparison to 471ms when the 3 : 2 aspect ratio image was profiled. This is not shocking however, as it could inherently have to do with the type of image used (perhaps it had less of a range of grey colours that needed to be generated to convert the image to greyscale and perhaps less scaling was required to get it down to the size such that it could be halftoned with 10 x 10 sized halftone dots). Some discrepancies in values are also expected to occur due to different loads being placed on the computer at the time of profiling. For example, it could be possible that another process started up in the background of the system while profiling was being conducted, causing a slowdown in the profiling process. Overall however, the results are very close to that of the 3 : 2 aspect ratio image, with convertToGrayscale taking the second longest amount of time after write();

4 : 3 aspect ratio 13 megapixel image



Call Tree – Method	Total Time [%]	Total Time	Invocations
main		8,393 ms (100%)	1
com.halftone.original.Image.<init> (String, String, int)		4,414 ms (52.6%)	1
com.halftone.original.Image.initialiseImage ()		3,953 ms (47.1%)	1
com.halftone.original.Image.convertToGrayscale ()		3,870 ms (46.1%)	1
com.halftone.original.Image.squashToFit ()		82.5 ms (1%)	1
Self time		0.022 ms (0%)	1
Self time		461 ms (5.5%)	1
com.halftone.original.ImageUtils.calculateHypotenuse (int, int)		0.070 ms (0%)	1
com.halftone.original.Image.write ()		3,373 ms (40.2%)	1
com.halftone.original.Image.convertToHalftone ()		604 ms (7.2%)	1
Self time		433 ms (5.2%)	1
com.halftone.original.ImageUtils.calculateAverage (java.util.ArrayList, int, int,		166 ms (2%)	129792
com.halftone.original.ImageUtils.calculateCircleDiameter (double, double, dou		4.39 ms (0.1%)	129792
com.halftone.original.Image.crop (int, int)		0.062 ms (0%)	1
Thread-0		0.178 ms (100%)	1
java.util.logging.LogManager$Cleaner.run ()		0.178 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.run ()		0.000 ms (0%)	1

The profiling was then conducted again using the 13 megapixel image with a ratio of 4 : 3. The total time taken to convert the larger image to halftone was 8393ms. From the snapshot above, the convertToGrayscale() method in fact took longer than the write() method, with convertToGrayscale() taking 3870ms and write() taking 3373ms respectively. This was indeed a surprising result. What can be deduced from this is that as the image size increases, the time taken to run the convertToGrayscale() method increases rapidly. Furthermore, the effect of the squashToFit() method on the overall time taken to call initialiseImage() was completely negligible by comparison to the convertToGrayscale() method. This finding was a large contributor to the decision to refactor the convertToGrayscale() method to make it more efficient.

Findings Based On Initial Profiling

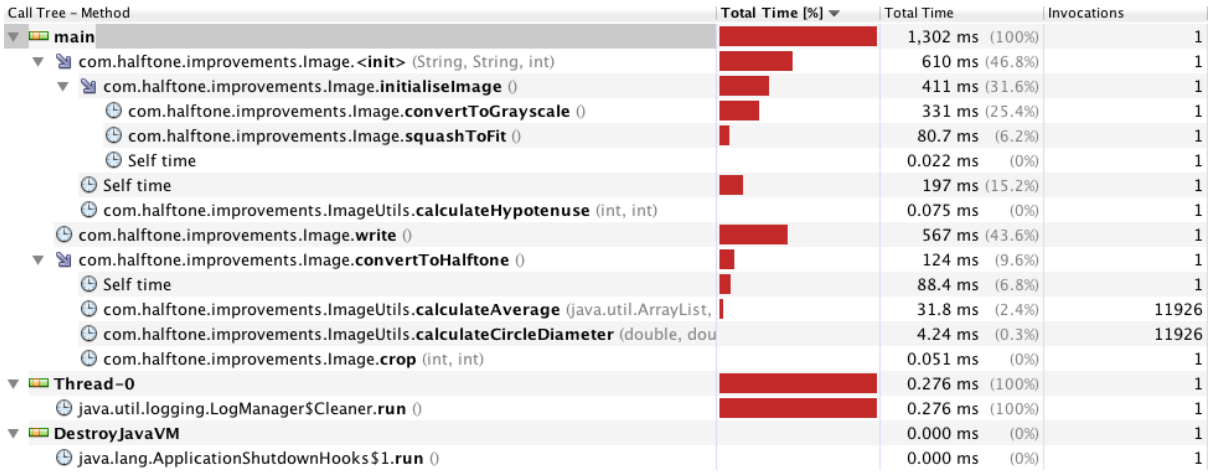
It can be seen that upon profiling the code in its unmodified state, the hotspots in the code were definitely the write() and the convertToGrayscale() methods across all snapshots. As the image fed into the application was increased in size, the speed of the convertToGrayscale() method in fact became more prevalent and succeeded the write() method in being the longest taking in duration. The code would probably not be fast enough in its current state to be used to convert images on the Nexus7 device as it is estimated that it will take between 5.80 seconds (as determined by summing the time taken to call initialiseImage() and convertToHalftone() for the 4 : 3 aspect ratio 1.2 megapixel image and multiplying it by 10) and 5.94 seconds (as determined by summing up the time taken to call initialiseImage() and convertToHalftone() for the 3 : 2 aspect ratio 1.2 megapixel image and multiplying it by 10). This was determined based on the premise that tablet and smartphone devices operate at $1/10^{th}$ of the speed of a standard PC. Memory consumption was determined not to be an issue as all of the data structures used to store image data were light-weight (e.g. a hash map containing the pixels corresponding to the original image, a hash map containing the grey Color objects used to convert the image to grayscale such that it can later be halftoned and an unavoidable BufferedImage to store the image for reading and writing).

Thus, after profiling the code in its current state, it was decided to make modifications to the convertToGrayscale method as it was obvious that it was having a significant impact on the overall runtime of the code through identifying that as the image size increased, convertToGrayscale took longer to run than the write() method.

After Modification

After comparing the snapshots before making any modification to the original code, it was found that the convertToGrayscale() and write() methods contributed most to the time taken to produce a halftone image. As the write() method uses its own built-in method (Java's ImageIO.write()), it was decided to focus on improving the run time of the convertToGrayscale() method. The snapshots below were taken after modifications were made to the convertToGrayscale() method and CPU profiling was run using the same images again.

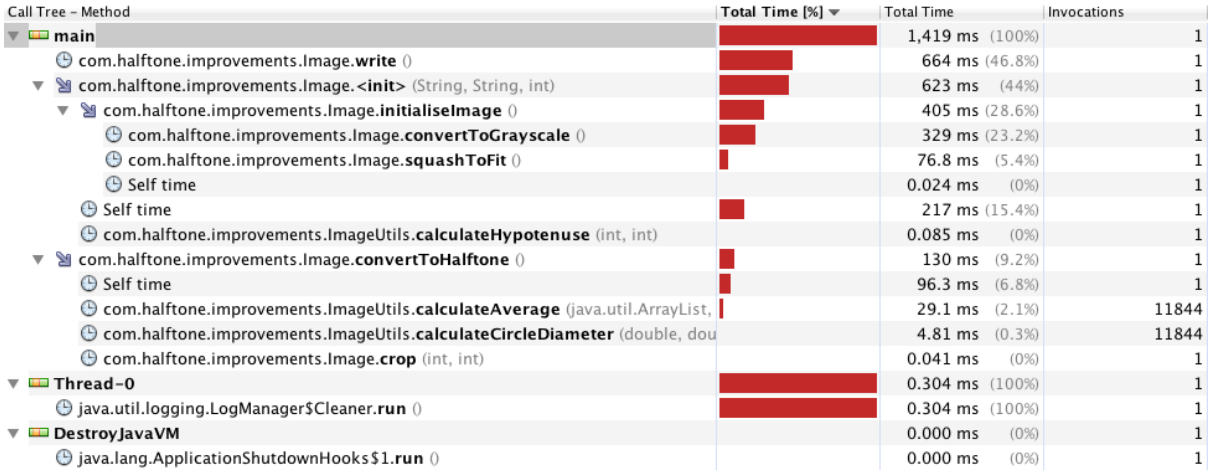
3 : 2 aspect ratio 1.2 megapixel image



Call Tree – Method	Total Time [%]	Total Time	Invocations
main		1,302 ms (100%)	1
com.halftone.improvements.Image.<init> (String, String, int)		610 ms (46.8%)	1
com.halftone.improvements.Image.initialiseImage ()		411 ms (31.6%)	1
com.halftone.improvements.Image.convertToGrayscale ()		331 ms (25.4%)	1
com.halftone.improvements.Image.squashToFit ()		80.7 ms (6.2%)	1
Self time		0.022 ms (0%)	1
Self time		197 ms (15.2%)	1
com.halftone.improvements.ImageUtils.calculateHypotenuse (int, int)		0.075 ms (0%)	1
com.halftone.improvements.Image.write ()		567 ms (43.6%)	1
com.halftone.improvements.Image.convertToHalftone ()		124 ms (9.6%)	1
Self time		88.4 ms (6.8%)	1
com.halftone.improvements.ImageUtils.calculateAverage (java.util.ArrayList,		31.8 ms (2.4%)	11926
com.halftone.improvements.ImageUtils.calculateCircleDiameter (double, dou		4.24 ms (0.3%)	11926
com.halftone.improvements.Image.crop (int, int)		0.051 ms (0%)	1
Thread-0		0.276 ms (100%)	1
java.util.logging.LogManager$Cleaner.run ()		0.276 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.run ()		0.000 ms (0%)	1

After modifying the convertToGrayscale() method, the total time taken to convert the 3 : 2 aspect ratio 1.2 megapixel image was 1302ms when it originally took 1326ms using the same image. The convertToGrayscale() method runs slightly faster, now taking 331ms while previously it took 387ms.

4 : 3 aspect ratio 1.2 megapixel image



Call Tree – Method	Total Time [%]	Total Time	Invocations
main		1,419 ms (100%)	1
com.halftone.improvements.Image.write ()		664 ms (46.8%)	1
com.halftone.improvements.Image.<init> (String, String, int)		623 ms (44%)	1
com.halftone.improvements.Image.initialiseImage ()		405 ms (28.6%)	1
com.halftone.improvements.Image.convertToGrayscale ()		329 ms (23.2%)	1
com.halftone.improvements.Image.squashToFit ()		76.8 ms (5.4%)	1
Self time		0.024 ms (0%)	1
Self time		217 ms (15.4%)	1
com.halftone.improvements.ImageUtils.calculateHypotenuse (int, int)		0.085 ms (0%)	1
com.halftone.improvements.Image.convertToHalftone ()		130 ms (9.2%)	1
Self time		96.3 ms (6.8%)	1
com.halftone.improvements.ImageUtils.calculateAverage (java.util.ArrayList,		29.1 ms (2.1%)	11844
com.halftone.improvements.ImageUtils.calculateCircleDiameter (double, dou		4.81 ms (0.3%)	11844
com.halftone.improvements.Image.crop (int, int)		0.041 ms (0%)	1
Thread-0		0.304 ms (100%)	1
java.util.logging.LogManager$Cleaner.run ()		0.304 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.run ()		0.000 ms (0%)	1

After the modifications were made to the convertToGrayscale() method, the time taken for processing the 4 : 3 aspect ratio 1.2 megapixel image was 1419ms while previously it took 1441. This was a very small improvement but again could be to do with the fact that the image had little variance in its determined grey values. The convertToGrayscale method took 329ms this time by comparison to 360ms before the modifications were made. This is once again a small, but considerable improvement on the original time that it took to run the method.

4 : 3 aspect ratio 13 megapixel image

Call Tree – Method	Total Time [%]	Total Time	Invocations
main		8,039 ms (100%)	1
com.halftone.improvements.Image.<init> (String, String, int)		4,200 ms (52.2%)	1
com.halftone.improvements.Image.initialiseImage ()		3,735 ms (46.5%)	1
com.halftone.improvements.Image.convertToGrayscale ()		3,652 ms (45.4%)	1
com.halftone.improvements.Image.squashToFit ()		82.3 ms (1%)	1
Self time		0.026 ms (0%)	1
Self time		465 ms (5.8%)	1
com.halftone.improvements.ImageUtils.calculateHypotenuse (int, int)		0.074 ms (0%)	1
com.halftone.improvements.Image.write ()		3,335 ms (41.5%)	1
com.halftone.improvements.Image.convertToHalftone ()		503 ms (6.3%)	1
Self time		348 ms (4.3%)	1
com.halftone.improvements.ImageUtils.calculateAverage (java.util.ArrayList, i		151 ms (1.9%)	129792
com.halftone.improvements.ImageUtils.calculateCircleDiameter (double, doub		2.96 ms (0%)	129792
com.halftone.improvements.Image.crop (int, int)		0.064 ms (0%)	1
Thread-0		0.229 ms (100%)	1
java.util.logging.LogManager$Cleaner.run ()		0.229 ms (100%)	1
DestroyJavaVM		0.000 ms (0%)	1
java.lang.ApplicationShutdownHooks$1.run ()		0.000 ms (0%)	1

Before the modifications were made to the convertToGrayscale() method, the 13 megapixel image with a 4 : 3 aspect ratio took 8393ms in total to be converted to halftone. However, after the modifications were performed, the total time taken was 8039ms. In terms of the convertToGrayscale() method, the time taken was 3652ms after the modifications were made while it was 3870ms originally. This verifies that the changes made in fact had a considerable impact on the overall computation time of the code base. As the difference in speed was noticeably improved upon running the modified code on the larger image, it can be deduced that as larger images are fed into the program, the speed up as a result of calling the new modified code over the old inefficient code will increase.

Conclusions and Recommendations for the Future

Ultimately, by modifying the convertToGrayscale() method, it was possible to improve the efficiency and performance of the overall code base. Instead of determining grey pixels using weightings from red, green and blue pixels, calculating the grey pixels by averaging the red, green and blue pixels was less computationally intensive, and the removal of the Colour hash map further improved the efficiency of the code. Given the changes now made to the speed of the code, it is estimated that it should take around 5350ms (5.35 seconds) (as calculated by adding the time taken to initialise a 3 : 2 aspect ratio 1.2 megapixel image : 411ms plus the time taken to perform the halftoning itself : 124ms which equals 535 and then multiplying it by 10 as smartphones and tablets run at one tenth of the speed of a PC. Surprisingly, the time taken to initialise a 4 : 3 aspect ratio 1.2 image : 405ms plus the time taken to perform halftoning on it : 130 also added to the same total.) to convert a standard image taken by the device's camera to a halftone image.

For future implementations, it is recommended that the image size be further compressed before converting to a grayscale image as it was seen that the algorithm ran extremely fast on smaller images. It may also be worthwhile to look at further optimising the convertToHalftone() method as that method is the next longest taking method in terms of computational time.