Scientific Research Publishing

Journal of Computer and Communications, 2020, 8, 60-74
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

Improve Image Decoding in Lightweight Environment Using a Coroutines Based Approach

Rodrigue Saoungoumi-Sourpele[1*], Jean Michel Nlong[2], Jean-Robert Kala Kamdjoug[3], Glen Vernyuy Yufui[4]

[1]Department of Mathematics and Computer Science, ENSAI, University of Ngaoundere, Ngaoundere, Cameroon
[2]Department of Mathematics and Computer Science, FS, University of Ngaoundere, Ngaoundere, Cameroon
[3]GRIAGES, Catholic University of Central Africa, Yaounde, Cameroon
[4]Department of Computer Engineering, IUT, University of Ngaoundere, Ngaoundere, Cameroon
Email: *rsaoungoumi@univ-ndere.cm

How to cite this paper: Saoungoumi-Sourpele, R., Nlong, J.M., Kamdjoug, J.-R.K. and Yufui, G.V. (2020) Improve Image Decoding in Lightweight Environment Using a Coroutines Based Approach. *Journal of Computer and Communications*, **8**, 60-74.
https://doi.org/10.4236/jcc.2020.810007

Received: September 24, 2020
Accepted: October 27, 2020
Published: October 30, 2020

Abstract

The JPEG2000 still image compression standard, while providing a remedy for the many shortcomings of its predecessor JPEG, is still slow to establish itself on the Internet. This problem is mainly due to the complexity of the COder-DECoder (CODEC) which implies its non-adoption by large firms and platforms in the field of image acquisition, processing and transmission. Indeed, the encoding and decoding process consumes a lot of CPU, memory and energy resources and takes a lot of computing time. The objective of this paper is to propose a model for decoding jpeg2000 on lightweight devices running on the Android mobile operating system. This implementation uses coroutines, which are a lightweight process model with reduced resource consumption costs compared to conventional AsyncTask threadsets. The model minimizes decoding time while minimizing CPU and memory usage, resulting in a fast and energetically economical decoded image. The results of integrating the coroutines from the main thread into the decoding process instead of the AsyncTask from the main thread produced better performance in terms of computation time, CPU and memory utilization. Indeed, the use of our model has led to a gain of around 23.41% in execution time, 9.8% in CPU utilization rate and 18.56% in memory utilization rate, compared to the model proposed in the literature which uses the threads.

Keywords

Coroutines, JPEG2000, Android, Thread, Asynctask

1. Introduction

The requirements for operating lightweight equipment in their diversity and wireless networks with limited and dynamic bandwidth to view images can be very diverse. This makes it necessary to set up an adaptation process for file encoding, delivery, reception and decoding. Optimization at each of these stages has an impact on the entire process. As a still image format, JPEG is widely used as the preferred image format on the Internet and on digital cameras. But the limitations displayed by this compression standard were among the motivations behind the development of the JPEG2000 system. But it is worth noting that this new standard was not only intended to provide superior compression efficiency compared to the basic JPEG system. Rather, it was intended to provide a new image representation with a rich set of features, all supported in the same compressed bitstream, that can accommodate a variety of existing and emerging compression applications. In particular, Part 1 of the standard addresses some of the shortcomings of basic JPEG by supporting the following feature set:

- Improved compression efficiency;
- Lossy or lossless compression;
- Multi-resolution representation;
- Built-in bitstream, allowing progressive decoding and scalability according to the signal-to-noise ratio (SNR);
- Tiling of the image;
- Coding of regions of interest (ROI);
- Resistance to errors;
- Random access and processing of data flows;
- Improved performance in case of compression/decompression cycle;
- A more flexible file format.

Despite all these advantages, the JPEG 2000 algorithm requires a lot of resources and calculations. The proportion of the requirements for each part is 70% for Embedded Block Coding with Optimized Truncation (EBCOT), 20% for wavelet transformation and the remaining 10% for all the other calculations [1]. The first part of the implementation concerns the MQ-Decode procedure [2]. This procedure is the basic element of the entropy decoder. It is used one to several times in each decoding pass. There is a lot of research into optimizing the parameters that can speed up the COder-DECoder (CODEC) process while minimizing the use of resources in the execution environment. In general, authors propose implementations based on Field Programmable Gate Arrays (FPGA)-type circuits [3] [4], Very High Speed Integrated Circuit Hardware Description Language (VHDL) [5] and Very-Large-Scale Integration (VLSI) [6] dedicated solely to the JPEG2000 codec. As far as software implementations are concerned, the standard's reference applications[1] are designed for heavy execution environments with enough resources for the CODEC. Some attempts to import on

[1]JPEG 2000 Software https://jpeg.org/jpeg2000/software.html.

light environments (Smartphones and tablets) on which the Android operating system runs, recommend using parallel processes in order to speed up the most time-consuming computing steps. In the latter case, common implementations use threads in the form of asynchronous tasks. Threads implemented on Android are inherited from the implementation made on the Linux kernel which was not originally designed for equipment with limited resources.

In this paper we propose a model for decoding jpeg2000 on lightweight devices running on the Android mobile operating system. This implementation uses coroutines, which are a lightweight process model with reduced resource consumption costs compared to conventional AsyncTask threadsets. The model minimizes decoding time while minimizing CPU and memory usage, resulting in a fast and energetically economical decoded image.

The rest of the paper is organised as follows: Section 2 is devoted to a review of JPEG2000; Section 3 is a review on coroutines; Section 4 presents our decoding model which is available in several versions. This is followed by a presentation of the results and analysis; Section 5 closes the paper with the conclusion and prospects for future work.

2. JPEG2000 Overview

The JPEG-2000 project was motivated by Ricoh's submission of the CREW algorithm [7] [8]. The fundamentals of a JPEG2000 encoder are shown in **Figure 1**. These components include pre-processing, Discrete Wavelet Transform (DWT), quantization, arithmetic coding (tier 1 coding) and bitstream organization (tier 2 coding). Each of these components is discussed in more detail below.

2.1. Preprocessing

The first step of the preprocessing consists in dividing the input image into rectangular, non-overlapping tiles of equal size. Then, the unsigned sample values in each component are level shifted (DC offset) by subtracting a fixed value of $2^B - 1$ from each sample to make its value symmetrical around zero. Finally, the level-shifted values can be subjected to a point intercomponent transformation to decorrelate the color data.

2.2. Discrete Wavelet Transform (DWT)

The block DCT transformation in the basic JPEG has been replaced by full image DWT [9] in JPEG2000. DWT decomposition provides a natural solution for the multi-resolution requirement of the JPEG2000 standard. The lowest resolution

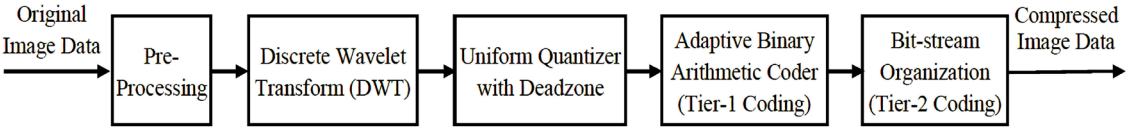

Figure 1. Fundamental elements of a JPEG2000 encoder.

at which the image can be reconstructed is called zero resolution. For example, referring to Figure 2, the 3LL sub-band would correspond to zero resolution for a 3-level decomposition. For a decomposition at N_L[2]-levels, the image can be reconstructed at N_L+1 resolutions. In general, to reconstruct an image at r resolution ($r>0$), the sub-bands $(N_L-r+1)HL$, $(N_L-r+1)LH$ and $(N_L-r+1)HH$ must be combined with the image at $(r-1)$ resolution. These sub-bands are designated as belonging to the r resolution: Resolution zero includes only the $N_L LL$ band. If the sub-bands are encoded independently, the image can be reconstructed at any resolution level by simply decoding the parts of the code stream that contain the sub-bands corresponding to that resolution and all previous resolutions. For example, referring to Figure 2, the image can be reconstructed at resolution two by combining the resolution one image and the three sub-bands labeled $2HL$, $2LH$ and $2HH$.

2.3. Quantization

It has been shown in [10] that the optimal R-D quantizer for a continuous signal with a Laplacian probability density (such as DCT or wavelet coefficients) is a uniform quantizer with a central dead zone [11]. The first part of the JPEG2000 standard adopted the dead zone with twice the step size due to its optimal integrated structure. In short, this means that if a quantization index of M_b bits resulting from a step size of Δ_b is transmitted progressively starting with the most significant bit (MSB) and progressing to the least significant bit (LSB), the resulting index after decoding only N_b bits is identical to that obtained using a similar quantizer with a step size of $\Delta_b 2^{M_b-N_b}$.

2.4. Entropy Coding

The quantization indices corresponding to the quantized wavelet coefficients in each sub-band are entropy-coded to create the compressed bitstream. The choice

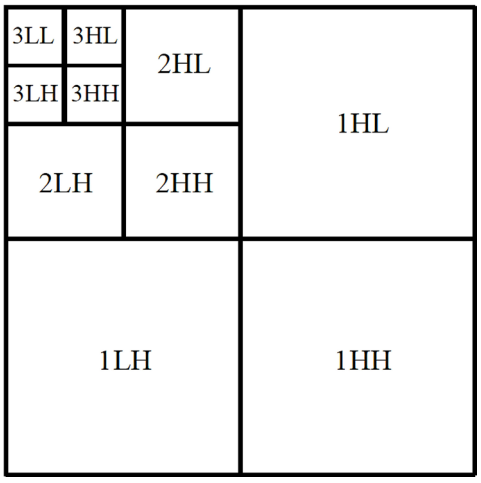

Figure 2. 2D wavelet decomposition at 3 levels.

[2]N_L is the notation used in the JPEG2000 document to indicate the number of resolution levels, although the L index can be somewhat confusing as it seems to indicate a variable.

of the entropy encoder in JPEG2000 is motivated by several factors. The first is the need to create an integrated bitstream, which is made possible by bit-plane coding of the quantization indices. Bit-plane coding of the wavelet coefficients has been used by several well-known integrated wavelet encoders such as Embedded Zero-tree Wavelet (EZW) [12] and Set Partitioning in Hierarchical Trees (SPIHT) [13]. However, these encoders use coding models that exploit the correlation between sub-bands to improve coding efficiency. Unfortunately, this has a negative impact on error resilience and severely limits the flexibility of an encoder to organize the bitstream in an arbitrary progression order. In JPEG2000, each sub-band is encoded independently of the other sub-bands. In addition, JPEG2000 uses a block coding paradigm in the wavelet domain as in the EBCOT [14] integrated block coding algorithm, where each sub-band is divided into small rectangular blocks, called code blocks, and each code block is coded independently. The nominal dimensions of a code block are free parameters specified by the encoder but are subject to the following constraints: they must be an integer power of two; the total number of coefficients in a code block may not exceed 4096; and the height of the code block may not be less than four. The quantized coefficients in a code block are bit-plane coded independently of all other code blocks when creating an embedded bitstream. Instead of encoding the entire bitplane in a single coding pass, each bitplane is encoded in three passes of sub-bitplanes with the ability to truncate the bitstream at the end of each coding pass. One of the main advantages of this approach is near-optimal integration, where the information that results in the greatest reduction in distortion for the smallest increase in file size is encoded first. In addition, a large number of potential truncation points facilitates an optimal rate control strategy where a target bit rate is achieved by including coding passes that minimize total distortion.

3. Overview of Coroutines

The idea of the coroutines goes back to the work of Erdwinn and Conway on a tape-based Cobol compiler and its separability into modules [15]. Although the original use case is no longer relevant, other use cases have emerged. Coroutines have been studied many times, and initially appeared in languages such as Modula-2 [16], Simula [17] and BCPL [18]. A detailed classification of coroutines is given by Moura and Ierusalimschy [19], as well as a formalization of asymmetric coroutins by an operational semantics. Moura and Ierusalimschy observed that stacked asymmetric first class coroutines have an expressive power equal to that of point continuations, but did not study snapshots, which make coroutines equivalent to complete continuations. Anton and Thiemann showed that it is possible to derive type systems for symmetrical and asymmetrical coroutines automatically by converting their reduction semantics into equivalent functional implementations and then applying the existing type systems for programs with continuations. James and Sabry identified the input and output types of corou-

tines [20], where the output type corresponds to the output type described in this paper. The input type assigns the past value to the coroutine when it is taken over. As a design compromise, we chose not to have explicit input values in our model. First, the input type increases the verbosity of the coroutine type, which may have practical consequences. Second, as indicated in [21], the input type can be simulated with the return type of another coroutine, which gives a writable location, and returns its value when taken back. Fischer *et al.* proposed a coroutine based programming model for the Java programming language, as well as the corresponding formal extension of Featherweight Java [22].

3.1. Coroutines and the Kotlin Language on Android

A coroutine is a simultaneity design model that you can use on Android to simplify code that runs asynchronously. Coroutines were added to Kotlin in version 1.3 and are based on concepts established in other languages.

Features

The coroutine system is the solution that Google recommends for asynchronous programming on Android. Among the remarkable features are the following:

- **Light:** You can run multiple coroutines on a single thread using the suspension bracket, which does not block the thread where the coroutine is run. Suspension saves memory compared to blocking while supporting many simultaneous operations;
- **Memory Leakage Reduction:** Use structured concurrency to perform operations within a given scope;
- **Integrated Cancellation Support:** Undo propagates automatically through the running coroutine hierarchy;
- **Jetpack integration:** Many Jetpack libraries include extensions that provide full support for the routines. Some libraries also provide their own coroutine scope that can be used for structured competition.

4. Optimization of the JPEG2000 Decoding Process on Android

4.1. Model Statement

In this section we introduce our models for optimizing the decoding process. In addition to using Threads to parallelize the execution of certain parts of the decoding that can be done, we use coroutines for the most resource-intensive processes.

4.1.1. First Approach: A General Decoding Coroutine

The classical decoding model of jpeg2000 files proposed for android[3] by Thales Group[4] implements a thread in addition to the main thread for the execution of the different steps of the decoding process, which produces the representation given in **Figure 3**. In the first approach of our model, we propose the integration

[3]JP2 for Android: https://github.com/gemalto/JP2ForAndroid.
[4]https://www.thalesgroup.com/en.

of a coroutine in charge of managing all the decoding process. This choice was made in order to demonstrate the efficiency of coroutines instead of the asynchronous tasks classically used on Android implemented with threads. The motivation of the choice also comes from the fact that coroutines have a light thread property. In a lightweight environment, execution optimization must take into

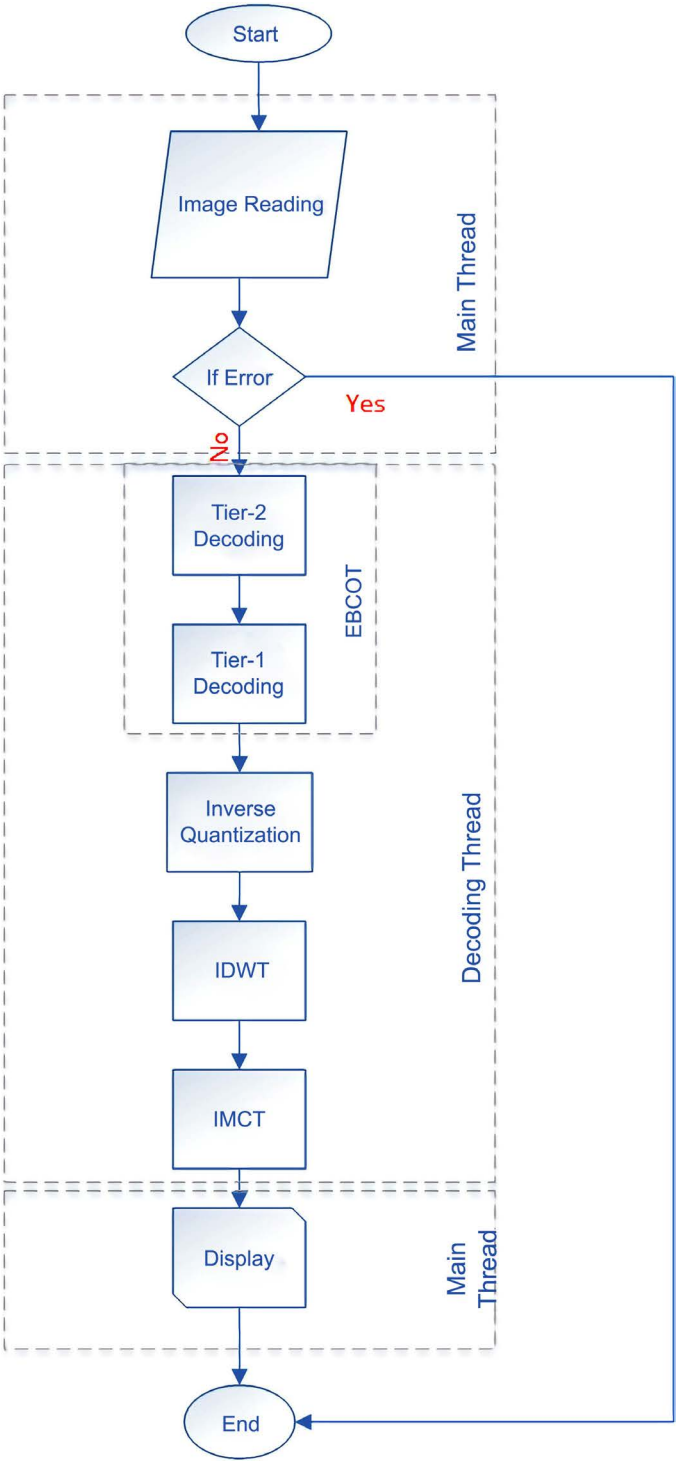

Figure 3. Classical decoding process of JPEG2000 on Android.

account resources (CPU, memory, battery). The proposed model is shown in the Figure 4. Given that the decoding process is computationally intensive, therefore, will mobilize a lot the process, we make the choice of a dispacher.Default(). The latter is adapted for this type of process.

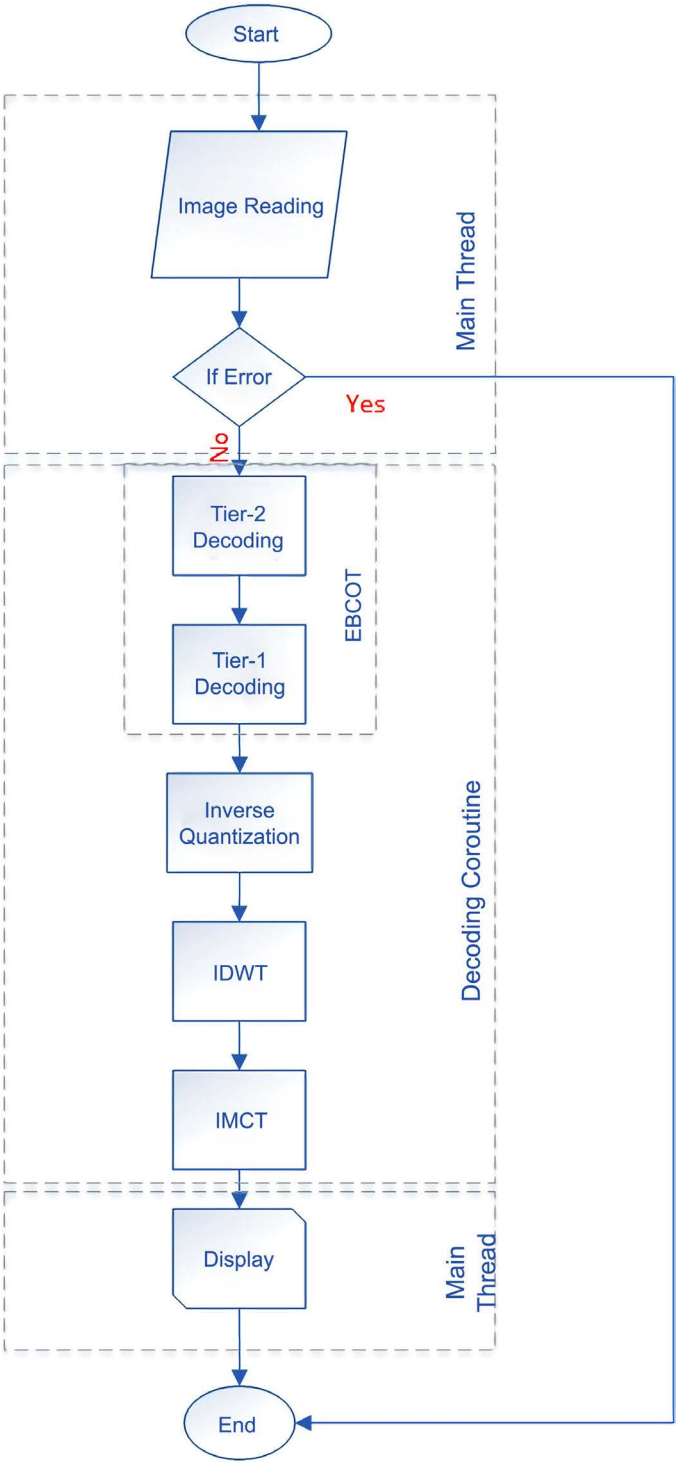

Figure 4. Single coroutine decoding process of JPEG2000 on Android.

4.1.2. Second Approach: Micro Network of Coroutines

The entire JPEG 2000 algorithm requires a lot of resources and calculations. The proportion of requirements for each part is 70% for EBCOT, 20% for wavelet transformation and the remaining 10% for all the other calculations [1].

In this second approach, we propose the integration of coroutins in the decoding process at the block level as shown in Figure 5. This produces 4 coroutines,

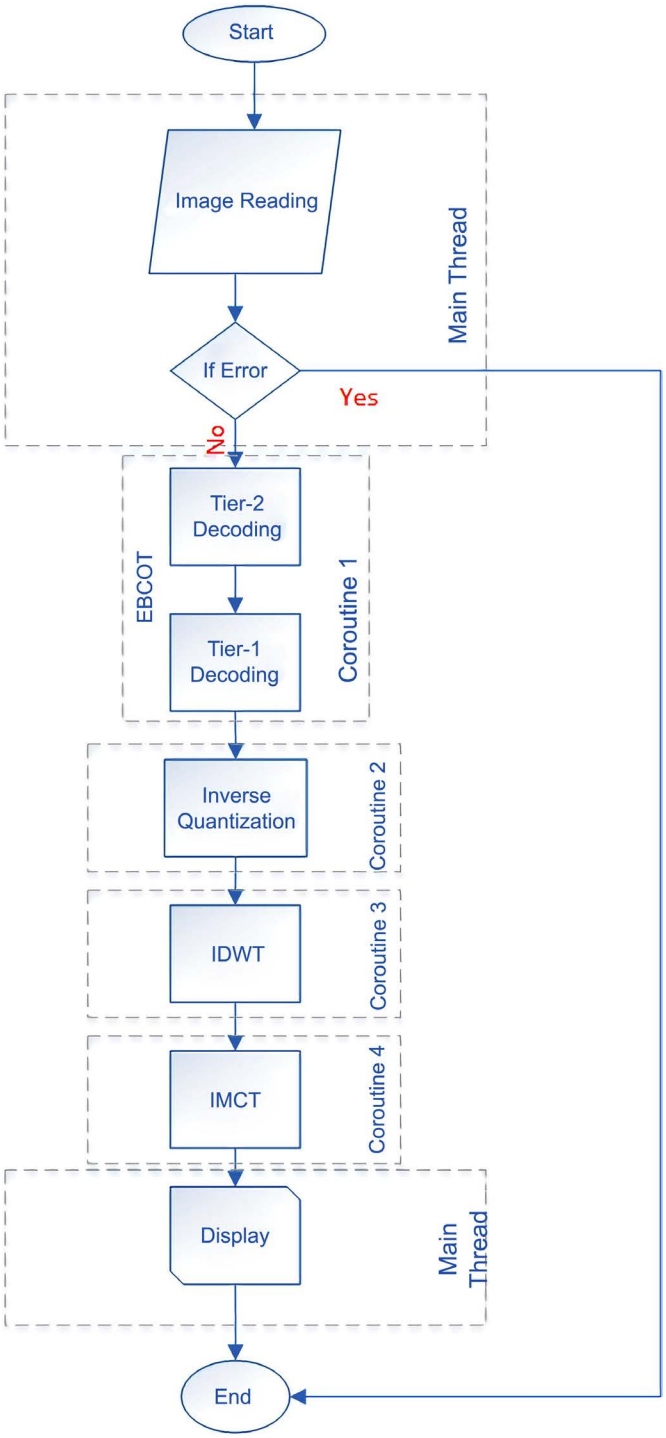

Figure 5. Micro network coroutines decoding process of JPEG2000 on Android.

one per decoding block. This approach can have a variant in 5 coroutines. This is done by splitting coroutine 1 into 2 coroutines, respectively in charge of executing tier-2 and tier-1 decoding. This variant is illustrated in the Figure 6.

4.1.3. Third Approach: Macro Network of Coroutines

In this approach, we propose to keep without change, apart from the EBCOT block, the whole system of the previous coroutine microarray. In EBCOT, which consumes more than 70% of the global computation time, we integrate as many

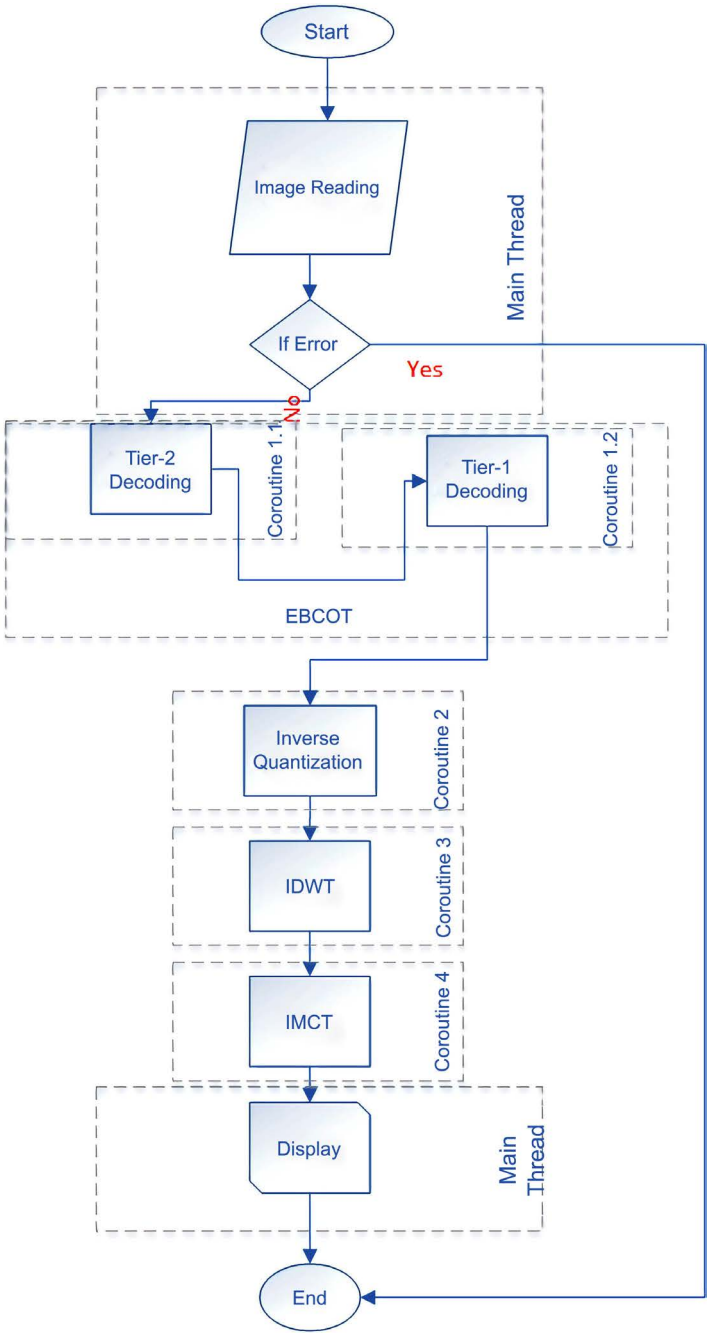

Figure 6. Micro network coroutines decoding process of JPEG2000 on Android: small zoom on EBCOT.

coroutines as packets in Tier 2 and as many coroutines as code blocks in Tier 1. Each coroutine in Tier 2 performs two tasks: packet decoding and arithmetic decoding to produce the bit planes and contexts necessary for decoding the bit planes in Tier 1. This model is illustrated by the flowchart in Figure 7.

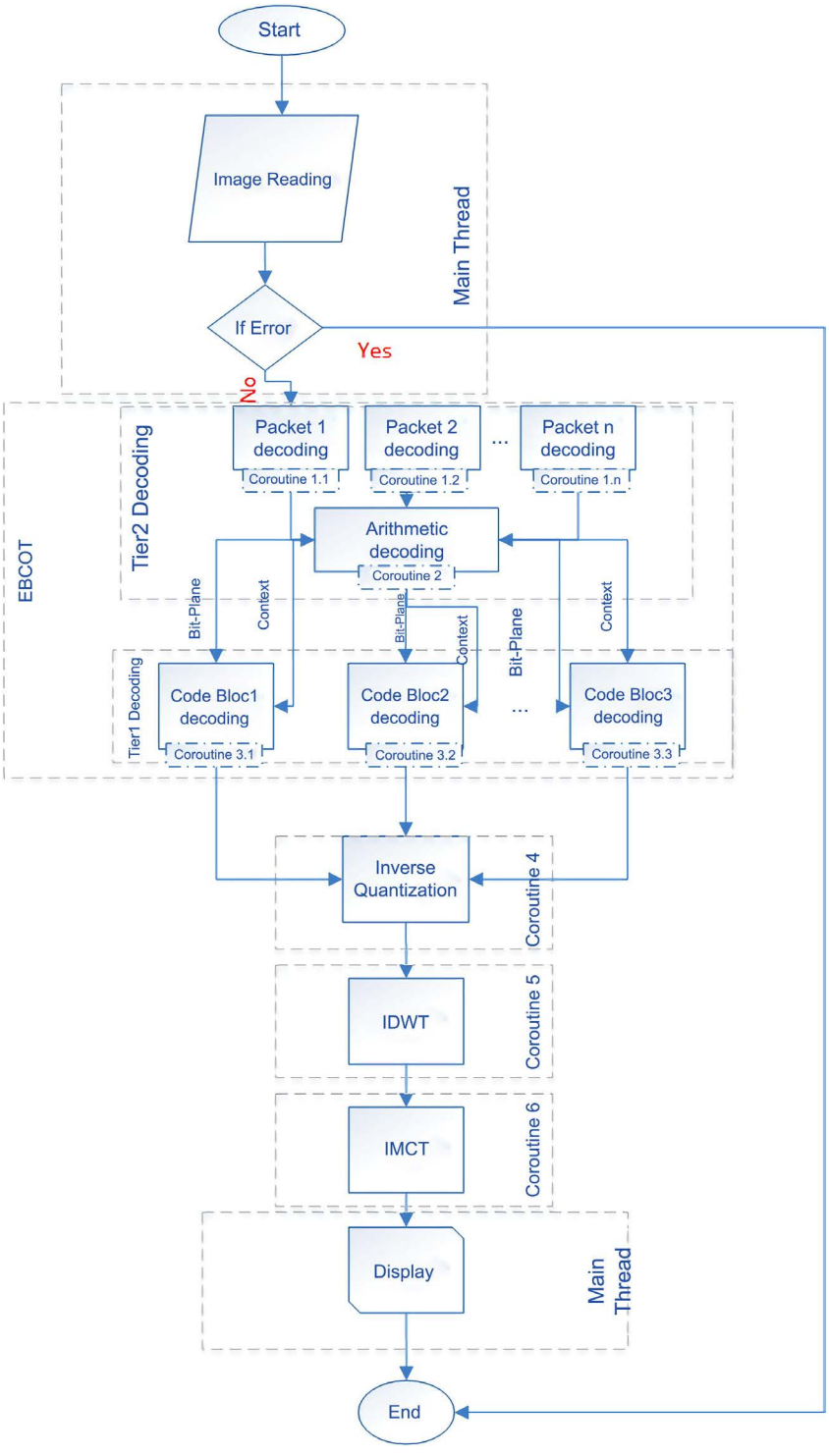

Figure 7. Android JPEG2000 coroutine macro array decoding process on Android: large zoom on EBCOT.

4.2. Results and Discussion

Implementation environment: The implementation environment is Android Studio on a machine with a Windows 10 operating system.

Execution environments: The tests were performed on a TECNO DP7CPRO phone with 1Gb RAM, Quad-core processor, with Android 5.1. The Android Studio profiler allowed us to retrieve information about the runtime environment.

Images and tests: We performed several decoding operations on the balloon files in color and High Definition (2717 × 3701), grayscale lena with a resolution of 512 × 512 and color lena with the same resolution.

Computation time: Figure 8 gives us the results in computing time for decoding and displaying "Balloon" and "Lena-grey" images. The result of this experiment shows that the use of coroutines results in an acceleration of the decoding process. The saving in time is about 23.41% on average.

CPU utilization rate: Figure 9 gives us the CPU usage rates for decoding and displaying "Balloon" and "Lena-grey" images in their entirety. An analysis of two curves clearly shows the efficiency of the use of coroutine which consumes less CPU resources to do the same work as with the "AsyncTask". Given that

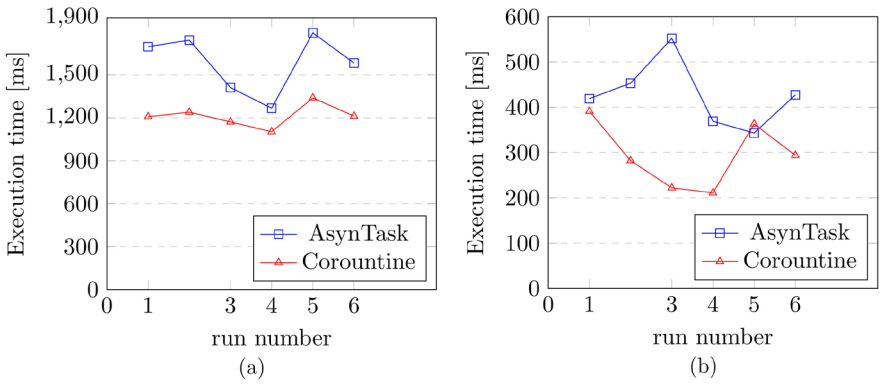

Figure 8. Execution time in milliseconds for decoding with AsyncTask and coroutines. (a) Balloon.jp2 (resolution of 2717 × 3701); (b) Lena-grey.jp2 (resolution of 512 × 512).

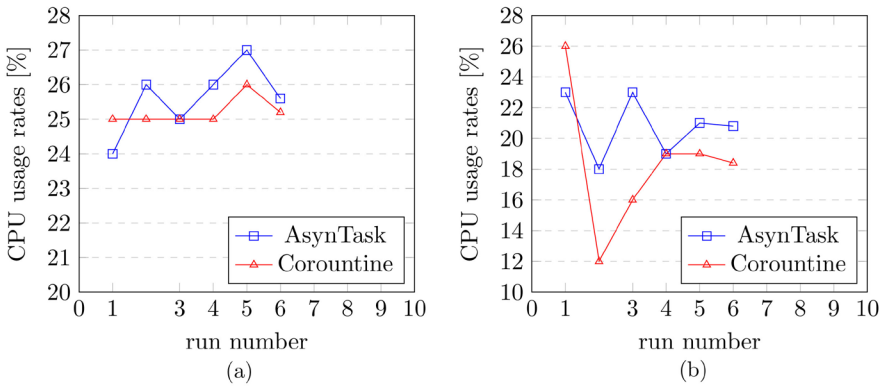

Figure 9. CPU usage ratese (%) during decoding with AsyncTask and coroutines. (a) Balloon.jp2 (resolution of 2717 × 3701); (b) Lena-grey.jp2 (resolution of 512 × 512.

the use of the CPU has an impact on the reactivity of the entire system as well as on energy consumption, we can say that the use of coroutine has a beneficial effect on the system. The average gain here is around 9.8%.

Memory space used: Figure 10 gives us the memory usage rates for decoding and displaying "Balloon" and "Lena-grey" images in their entirety. This result shows once again that coroutines use fewer resources than AsynTask. Here in terms of memory occupied during the decoding operation, we see that the average gain is around 18.56%.

Discussion

The results we obtained and previously compared to the model integrating the AsyncTask under Android show the better character of our model. These results, obtained in an environment similar to the one presented in [23], which carries out an integration test of the JPEG2000 encoder/decoder under Android in terms of execution time, show that: for an image with a resolution of 512 × 512 (in this case that of LENA for its example and ours as well), we have an average decoding time of 0.3s whereas in [23] the same image is decoded in 1s. We observe that our model obtains a time saving of around 70% compared to the result of [23]. Still in the same vein, the gain in calculation time for an image with a size of 1024 × 1024 is of the order of 24% (2.5s for [23] and 1.8s for our model), still in favour of our model. As demonstrated by [23], the power of the CPUs has a great influence on the speed of encoding and decoding images. We also know that with the rapid development of mobile equipment (smartphones in particular), processors will become increasingly powerful. This increase in power will therefore have a positive impact on image decoding time. This time must tend to be less than 1/24th of a second (41.6 ms) to hope to see JPEG2000 adopted as a still image form for videos with an isochronism of 24 frames per second. These results are a contribution to the adoption of the JPEG2000 file format in limited environments such as smartphones and tablets. This adoption will have a number of benefits, among which, the decrease in power consumption during the image decoding process will have a positive impact on battery life. In addition,

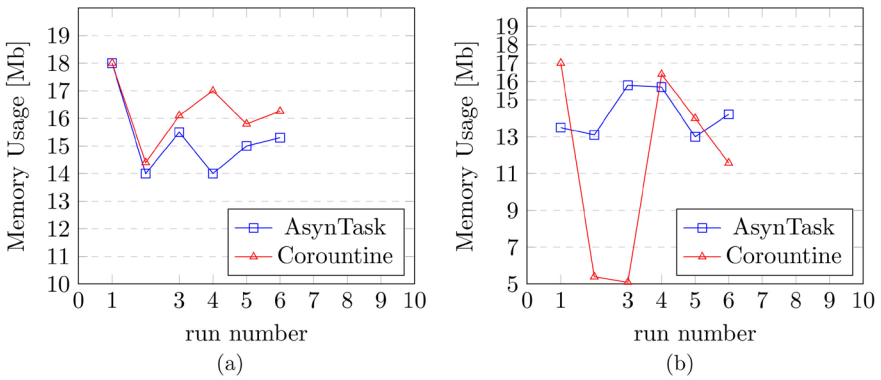

Figure 10. Memory space consumed (in Mb) during decoding with AsyncTask and Coroutines. (a) Balloon.jp2 (resolution of 2717 × 3701); (b) Lena-grey.jp2 (resolution of 512 × 512.

the consumption of memory, which is also a critical resource in limited environments, will be reduced.

5. Conclusion

In this paper, we have reviewed the process of encoding/decoding images in JPEG2000 format, as well as the corountin-type processes which, according to Google, perform better than Threads. We have performed decoding tests using .jp2 files on Android with Threads and coroutines. The results obtained show us that coroutines run faster than threads, while consuming less processor resources. The tests carried out made use of a coroutine that managed the whole decoding process. In perspective, we believe that the implementation of a network of coroutines, each of which has to execute a part of the decoding process, may have better results.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Nicholson, D., Martinez, P., Iregui, M. and Corral, J. (2000) Evaluation de la complexité algorithmique de JPEG2000. *Proceedings CORESA*, Poitiers.

[2] Fages, J.-M. (2000) JPEG2000: Principes, implémentation et évaluation. PhD Thesis, Cnam.

[3] Iskander, I.F., Al-Rawi, D.R. and Al-Tikriti, M.N. (2006) Design and Implementation of JPEG 2000 Image Compression Using FPGA. *Journal of Engineering and Sustainable Development*, **10**, 151-162.

[4] Descampe, A., Devaux, F., Rouvroy, G., Macq, B. and Legat, J.-D. (2004) An Efficient FPGA Implementation of a Flexible JPEG2000 Decoder for Digital Cinema. 2004 12*th IEEE European Signal Processing Conference*, Vienna, 6-10 September 2004, 2019-2022. https://doi.org/10.1109/MELCON.2004.1348272

[5] Wagh, K.H., Dakhole, P.K. and Vinod, G.A. (2008) Design & Implementation of JPEG2000 Encoder Using VHDL. *Proceedings of the World Congress on Engineering*, Vol. 1, 2-4.

[6] Huang, Q., Zhou, R. and Hong, Z. (2004) Low Memory and Low Complexity VLSI Implementation of JPEG2000 Codec. *IEEE Transactions on Consumer Electronics*, **50**, 638-646. https://doi.org/10.1109/TCE.2004.1309443

[7] Zandi, A., Allen, J.D., Schwartz, E.L. and Boliek, M. (1995) Crew: Compression with Reversible Embedded Wavelets. *Proceedings DCC*'95 *Data Compression Conference*, Snowbird, 212-221. https://doi.org/10.1109/DCC.1995.515511

[8] Boliek, M.P., Gormisch, M.J., Schwartz, E.L. and Keith, A.F. (1998) Decoding Compression with Reversible Embedded Wavelets (Crew) Code-Streams. *Journal of Electronic Imaging*, **7**, 402-410. https://doi.org/10.1117/1.482653

[9] Lian, C.-J., Chen, K.-F., Chen, H.-H. and Chen, L.-G. (2001) Lifting Based Discrete Wavelet Transform Architecture for JPEG2000. *The* 2001 *IEEE International Symposium on Circuits and Systems*, Vol. 2, 445-448.

[10] Sullivan, G.J. (1996) Efficient Scalar Quantization of Exponential and Laplacian

Random Variables. *IEEE Transactions on Information Theory*, **42**, 1365-1374. https://doi.org/10.1109/18.532878

[11] Christopoulos, C., Skodras, A. and Ebrahimi, T. (2000) The JPEG2000 Still Image Coding System: An Overview. *IEEE Transactions on Consumer Electronics*, **46**, 1103-1127. https://doi.org/10.1109/30.920468

[12] Shapiro, J.M. (1993) Embedded Image Coding Using Zerotrees of Wavelet Coefficients. *IEEE Transactions on Signal Processing*, **41**, 3445-3462. https://doi.org/10.1109/78.258085

[13] Said, A. and Pearlman, W.A. (1996) A New, Fast, and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees. *IEEE Transactions on Circuits and Systems for Video Technology*, **6**, 243-250. https://doi.org/10.1109/76.499834

[14] Taubman, D. (2000) High Performance Scalable Image Compression with EBCOT. *IEEE Transactions on Image Processing*, **9**, 1158-1170. https://doi.org/10.1109/83.847830

[15] Conway, M.E. (1963) Design of a Separable Transition-Diagram Compiler. *Communications of the ACM*, **6**, 396-408. https://doi.org/10.1145/366663.366704

[16] Wirth, N. (1985) Programming in Modula-2 (Texts and Monographs in Computer Science). Springer-Verlag, Berlin.

[17] Nygaard, K. and Dahl, O.-J. (1978) The Development of the Simula Languages. In: *History of Programming Languages*, ACM, New York, 439-480. https://doi.org/10.1145/800025.1198392

[18] Moody, K. and Richards, M. (1980) A Coroutine Mechanism for BCPL. *Software: Practice and Experience*, **10**, 765-771. https://doi.org/10.1002/spe.4380101002

[19] Moura, A.L.D. and Ierusalimschy, R. (2009) Revisiting Coroutines. *ACM Transactions on Programming Languages and Systems*, **31**, 1-31. https://doi.org/10.1145/1462166.1462167

[20] James, R.P. and Sabry, A. (2011) Yield: Mainstream Delimited Continuations. *First International Workshop on the Theory and Practice of Delimited Continuations*, Vol. 95, 96.

[21] Prokopec, A. and Liu, F. (2018) Theory and Practice of Coroutines with Snapshots. 32*nd European Conference on Object-Oriented Programming*, Dagstuhl, 3:1-3:32. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2018.3

[22] Fischer, J., Majumdar, R. and Millstein, T. (2007) Tasks: Language Support for Event-Driven Programming. *Proceedings of the* 2007 *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 134-143. https://doi.org/10.1145/1244381.1244403

[23] Wang, H.-Y. and You, X.-D. (2012) Study of JPEG2000 on Android. 2012 *International Conference on Wavelet Active Media Technology and Information Processing*, 57-61.